CAWR: Buffer Replacement with Channel-Aware Write Reordering Mechanism for SSDs

Ronghui Wang, Zhiguang Chen, Nong Xiao, Minxuan Zhang, and Weihua Dong

A typical solid-state drive contains several independent channels that can be operated in parallel. To exploit this channel-level parallelism, a variety of works proposed to split consecutive write sequences into small segments and schedule them to different channels. This scheme exploits the parallelism but breaks the spatial locality of write traffic; thus, it is able to significantly degrade the efficiency of garbage collection. This paper proposes a channel-aware write reordering (CAWR) mechanism to schedule write requests to different channels more intelligently. The novel mechanism encapsulates correlated pages into a cluster beforehand. All pages belonging to a cluster are scheduled to the same channels to exploit spatial locality, while different clusters are scheduled to different channels to exploit the parallelism. As CAWR covers both garbage collection and I/O performance, it outperforms existing schemes significantly. Trace-driven simulation results demonstrate that the CAWR mechanism reduces the average response time by 26% on average and decreases the valid page copies by 10% on average, while achieving a similar hit ratio to that of existing mechanisms.

Keywords: SSD, multichannel, buffer replacement, parallelism, garbage collection, write reordering.

I. Introduction

Due to the improved bandwidth and random I/O performance, NAND flash–based solid-state drives (SSDs) are replacing hard disk drives (HDDs) in high-end enterprise-scale storage systems and high-performance computing (HPC) environments. However, flash memory exhibits some peculiarities that are incompatible with existing software stacks. Accordingly, SSDs internally employ a flash translation layer (FTL) to hide the idiographic characteristic of flash memory and to mimic themselves as block devices. FTL provides an address mapping between the logical addresses used by the host and the physical addresses used in flash memory. Besides this, FTL internally issues extra read, write, or erase operations to efficiently manage the storage space, and the number of those extra operations depends both on the data access pattern from the upper layer and the algorithm adopted by the address mapping.

As Fig. 1 shows, the modern SSD uses an on-disk buffer in-between the host interface and the FTL. The buffer stores data from the host first and then writes the data to the NAND flash memory afterwards. The replacement policy employed by the write buffer should take both the write sequence and the FTL algorithm into account. Furthermore, as the SSD usually contains several independent channels, the replacement policy is also responsible for scheduling write requests among these channels to exploit parallelism. Most existing buffer replacement policies do not consider hardware parallelism, and their evicting sequence contains quite a number of consecutive pages even from a page-level buffer. When these buffer policies are applied to independent channels, directly scheduling the consecutive sequence to separate channels does help to exploit parallelism; however, this will increase the

Manuscript received Jan. 13, 2014; revised May 20, 2014; accepted June 16, 2014.

Ronghui Wang (corresponding author, ronghuiw@gmail.com), Zhiguang Chen (chengzhiguanghit@gmail.com), Nong Xiao (nongxiao@nudt.edu.cn), and Minxuan Zhang (mxzhang@nudt.edu.cn) are with the School of Computer, National University of Defense Technology, Hunan, China

Weihua Dong (ttinywolf@gmail.com) is with the Department of Software, the State Key Laboratory of Astronautic Dynamics, Hangzhou, China.

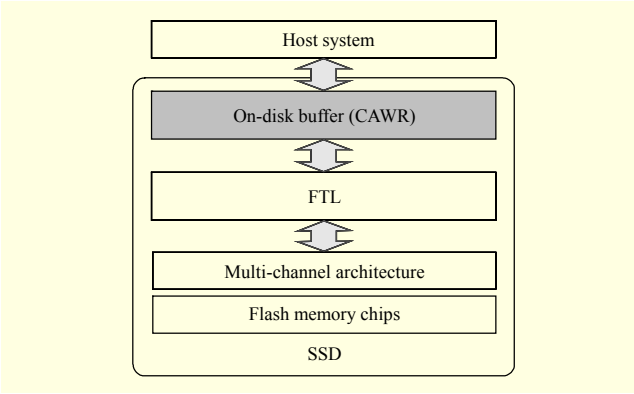

Fig. 1. SSD overview: proposed CAWR is applied to the buffer inside the SSD.

overhead of garbage collection significantly, since scheduling a consecutive write sequence to multiple channels will break the spatial locality of write traffic. There are only two buffer policies designed for independent channels. Chang and others [1] proposed a gating buffer to collect requests and instantaneously flush a page to every channel in parallel, but they did not designate the write pattern. CAVE [2] is another buffer policy that makes use of a multiple eviction technique capable of maintaining the sequential pattern of a block-level buffer. However, its round-robin channel allocation manner may break the sequential order of the block; moreover, it cannot address the consecutive pages with page-level buffer replacement policies.

This paper proposes a channel-aware write reordering (CAWR) mechanism to recognize the patterns of pages to be evicted from the buffer. In this work, the buffer is logically partitioned into two regions. The first region is managed by the buffer replacement policy in the traditional manner. Pages evicted from the first region are delivered to the second region. In this region, correlated pages are encapsulated into a cluster. When buffer replacement is required, the CAWR mechanism schedules a cluster to each channel, guaranteeing that all channels are operated in parallel. Furthermore, as all pages in a given cluster are correlated, scheduling the entire cluster to a given channel guarantees that the spatial locality of write traffic remains intact. In conclusion, the CAWR mechanism aggressively exploits the parallelism among channels and thus achieves a higher I/O performance. On the other hand, as correlated pages are scheduled to the same channel and even written to the same physical block, the overhead of garbage collection is significantly reduced.

CAWR bridges the information gap between the buffer replacement policy and the hardware architecture. The existing multiple eviction techniques need the buffer to identify the page pattern; otherwise, the sequential order will be totally lost. However, if the working cache organizes the cached data into clusters, because of the possibility of mixing cold data and hot data inside a cluster, the cold pages will stay in the cache with the hot ones if they belong to the same cluster, causing a waste of cache space and a degradation of the hit ratio. In addition, page-level mapping FTLs remove the block merge, releasing the constraint of block borders; thus, the advantage of a cluster-organized buffer to create larger sequential writes is lost. CAWR aims at small sequential pages naturally generated from the cache. It organizes the page clusters at the end of the buffer, only for the pages that have already been identified as cold data to be evicted; thus, the working cache region need only focus on the hit ratio and read-write asymmetry of the flash memory.

We implement the CAWR with a page-level replacement policy and simulate it in a flash-based SSD simulator with realistic workloads. The experimental results show that CAWR does not critically affect the hit ratio of an on-disk buffer. By reordering the page sequence, the number of reclaiming operations is reduced, and by evicting multiple pages simultaneously, the performance is improved. Compared to the CAVE, with the same replacement policy, our method reduces the garbage collection overhead, and as a consequence of doing so, slightly improves the performance.

II. Background

NAND flash memory consists of a number of blocks, each of which consists of the same number of pages. There are three basic operations for a NAND flash memory: read, write (program), and erase. A block is a basic unit of erase operations, while a page is a basic unit of read and write operations. A write operation is much slower than a read, while an erase operation is even slower than a write. If a page has been written, it cannot be overwritten until the block that the page belongs to is erased; that is, the erase-before-write characteristic. Therefore, flash memory uses "out-of-place write" rather than "in-place write" in HDDs. Furthermore, the number of erasures that each block can survive is limited — 10,000 times for multi-level cell or 100,000 times for single-level cell flash memory.

Flash storage devices internally employ an FTL to hide the characteristic of flash memory and emulate it as a block device. The most important function of the FTL is to maintain a mapping between the logical block addresses (LBAs) used by the host and the physical block and page addresses used in flash memory. This mapping can either be at the page level, block level, or hybrid level. Although, as far as we know, most current commercial SSD products employ the hybrid-mapping FTL schemes, these FTLs usually perform, arguably, more poorly than page-mapping FTLs. The only weakness of a

page-mapping FTL is that it consumes a prohibitively large RAM space to store the page-mapping information. With techniques for optimizing the huge demand on the RAM, such as DFTL [3] and HAT [4], page-mapping FTLs are a promising alternative to the hybrid schemes. Another basic function of an FTL is that it cleans blocks for reuse. This function is performed by a garbage collector, which recycles blocks after all the valid pages within them have been migrated elsewhere. Moreover, for the limited program-erase cycle count, an FTL adopts a wear-leveling technique to maximize the endurance of the flash memory, keeping as many writable blocks as possible.

Modern NAND flash–based SSDs consist of multiple channels, where each channel has multiple NAND flash memory chips. The multiple channels of SSDs can be organized as synchronized channels or independent channels. Synchronized channels have all channels perform the same flash command at the same flash address. The pages and blocks of the same flash addresses in all channels form a super page and a super block, which scales up the unit sizes of read-write and erase by the total number of channels. It provides high I/O throughput but dramatically increases the read-modify-write overhead [1] and garbage collection overhead [2]. Independent channels carry out flash operations on their own data, commands, and addresses. It is more flexible, but the problem of how to maintain high channel utilization is a big design obstacle. An FTL should take the hardware architecture into account for independent channels. Our method is designed for independent channels too, exploiting the channel-level parallelism from the upper layer.

In addition to the array of flash chips and the FTL, SSD has an on-disk buffer. The buffer holds the metadata of the FTL and also works as a cache to improve performance. The replacement policy employed by the buffer should take both the characteristics of the flash memory and the FTL algorithm into account. Since currently commercial SSDs employ a hybrid-mapping FTL, some buffer replacement policies try to decrease the number of merge operations by clustering pages in the same block and destaging them at the same time. With a page-mapping FTL, since the page can be placed anywhere, buffer management policies need not consider the block borders.

III. Related Work

According to the replacement granularity, the flash-aware buffer replacement policies can be classified into two types: page level and block level. The Least Recently Used (LRU) policy is the basis for most of these policies. Block-level replacement policies organize the buffered data in the unit of block. When a replacement is needed, they replace data of a whole block or several pages that belong to one block. For example, FAB [5] maintains an LRU block list: the pages that belong to the same physical block of flash memory are grouped together, and a group is moved to the Most Recently Used (MRU) position of the list when the group reads, updates, or inserts a page. FAB selects the block with the most number of pages to produce larger sequential writes. BPLRU [6] targets to random write patterns and uses the page padding technique to change the fragmented write patterns to sequential ones. CLC [7] selects a cold large cluster as a victim to increase the hit ratio. Block-level replacement policies produce large sequential writes; however, it is hard for them to treat read and write operations differently; thus, some of them work only as a write buffer and simply redirect the non-cached read requests to the beneath layer.

Page-level replacement policies replace data in the unit of page. Most existing page-level algorithms (for example, CFLRU [8] and LRU-WSR [9]) focus on the asymmetric latencies between read and write in flash-based storage systems, trying to give a higher priority towards evicting clean pages rather than dirty pages (that is, a clean-first policy). CFLRU and LRU-WSR do not consider the access frequencies of data. They merely keep cold dirty data and evict hot clean data; thus, they may degrade the overall I/O performance. CCF-LRU [10] further refines the idea of LRU-WSR by distinguishing cold-clean from hot-clean pages. Cold pages are distinguished from hot pages using the second chance algorithm. They define four types of eviction costs: cold-clean, cold-dirty, hot-clean, and hot-dirty, with increasing priority. The aforementioned page-level policies do not address the write patterns. REF [11] chooses a page having the same logical block number that is recently evicted as a victim for reducing a block merge number and associativity; however, it does not distinguish between the clean and dirty states of pages.

Existing buffer replacement policies do not consider the parallelism exhibited by the multichannel architecture of modern SSDs. Chang and others [1] proposed a gating buffer to collect requests and instantaneously flush a page to every channel in parallel. CAVE [2] shares the same idea but is more specific. CAVE is claimed to work well with block-level and page-level replacement policies: it only considers the eviction rate, while replacement policies only focus on the eviction order. Figure 2 gives examples of CAVE cooperating with block-level and page-level policies. There are some problems with CAVE: (a) when CAVE cooperates with block-level replacement policies, since it uses a round-robin scheme for allocating the channel number to a block, sequential pages belonging to a given block will be written to different channels; (b) when CAVE cooperates with page-level replacement

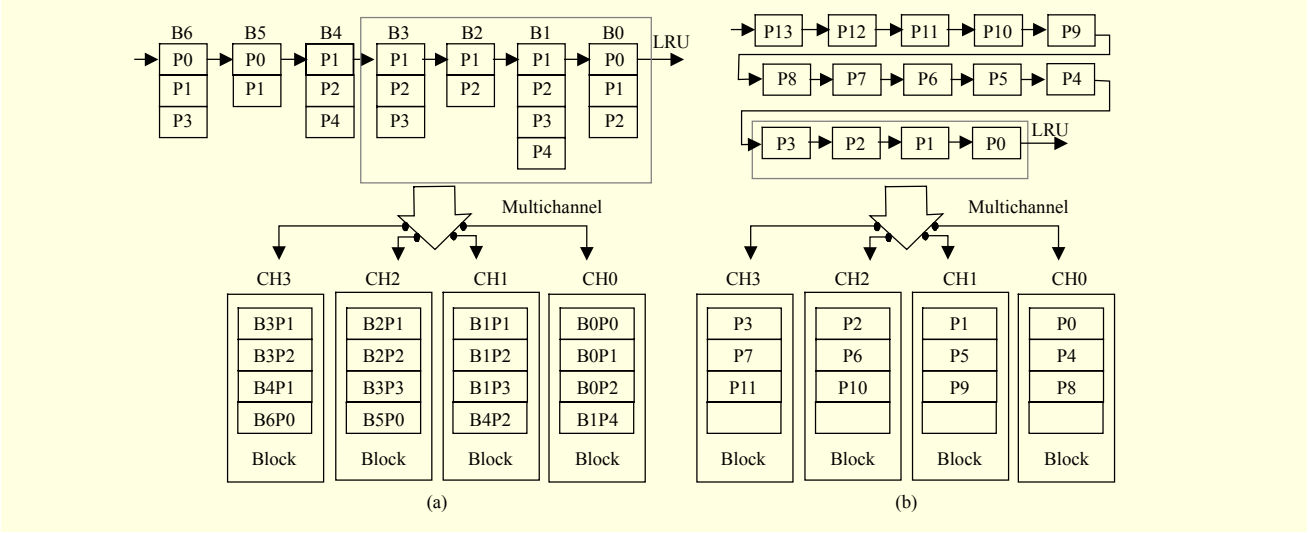

Fig. 2. Examples of CAVE with buffer replacement policies: (a) block-level + CAVE and (b) page-level + CAVE.

policies, it does not consider the write pattern of the evicted pages and thus simply flushes multiple victims to different parallel units. However, by recording the evicted page sequences of some real-world workload traces, we find that even page-level replacement policies flush quite a number of sequential writes, on purpose or unconsciously. Striping these sequential writes also breaks the spatial locality; and (c) CAVE is only applied to a write buffer involving no caching for read operations. Our proposed method applies itself well to a read-write buffer and also solves the first two problems of CAVE.

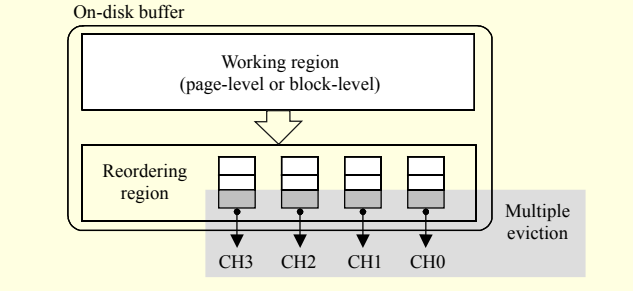

Fig. 3. Structure of CAWR buffer.

IV. CAWR

1. Motivation and Two-Region Buffer

As we have mentioned, existing buffer replacement policies do not exploit I/O parallelism, while gating buffer [1] and CAVE [2] only consider the eviction rate. CAVE can partially keep the sequential write pattern for block-level replacement policies, but for page-level policies the pattern information is totally lost. However, from some experiments of real-world traces, we find that the pages flushed from the page-level buffer can still be clustered, as can be the case with the original access sequence. To evict multiple victims for increasing I/O parallelism and to keep the spatial locality of sequential writes inside a physical block, we propose the CAWR to bridge the information gap between the buffer management policy and the multichannel architecture.

We logically partition the buffer into two parts, as Fig. 3 shows. The first part is the working region that is managed by a traditional cache replacement policy. The other is the reordering region that is managed by CAWR. The sizes of the two regions are dynamically changed, and only pages to be replaced are logically partitioned into the reordering region. The working region can adopt both block-level and page-level replacement policies, obtaining the block-level CAWR and page-level CAWR. For a page-level CAWR, cold pages are retired from the working region, and then they enter into the reordering region instead of being directly flushed into the flash memory. In the reordering region, CAWR clusters correlated pages for each channel, guaranteeing the spatial locality to write traffic. Then when a buffer replacement is required, CAWR evicts pages from each cluster to each channel simultaneously, guaranteeing that all channels are operated in parallel.

When the working region employs a block-level policy, the underlying FTL layer must apply a block-mapping or hybrid-mapping FTL. The block-level CAWR should keep the block borders and works almost like CAVE; multiple blocks are destaged from the working region and enter into the reordering region for multiple pages eviction. During the entering, cleaning pages in those blocks are first evicted, so only dirty pages remain in the channel clusters. Since the pages already have been clustered in the working region, there is no need to perform the reordering step. Because when a block enters into one cluster the channel number has already been assigned and

firmly fixed; thus, CAWR solves the problem that is caused by round-robin allocating the channel number to a block. However, because of the relatively lower hit ratio of block-level buffer policies, page-level policies are more preferable for CAWR. The following discussions mostly focus on the page-level policy.

2. Write-Reordering and Multi-eviction Scheme

In the reordering region, CAWR uses multiple cold dirty lists (CDLs), which will be illustrated in the next subsection to store the pages to be flushed to flash memories. The number of lists is equal to the number of channels in the SSD. When a page is evicted by the working region, it is arranged into one of the CDLs in the reordering region according to the replacement sequence. Algorithm 1 describes how to arrange the pages, and Fig. 4 gives an example of a multiple-pages eviction of a page-level CAWR. Notice that this page-level CAWR is not designed for a block-mapping or hybrid-mapping FTL, it only organizes the cold dirty pages into page clusters without designating the block borders. Otherwise, if the underlying FTL applies a block or hybrid mapping, then the *isSequence* condition (line 6 in Algorithm 1) to arrange pages should be changed to a new condition that judges whether two pages belong to the same block.

In Algorithm 1, the dirty page at the LRU position of the working list first tries to append to a nonempty CDL (the first FOR loop). If the page is not consecutive to any pages in all CDLs, then this tentative step fails, and the page then tries to append to an empty CDL (the second FOR loop). We also define a maximum length for each CDL, avoiding a long sequence of consecutive pages, which may result in a worse cache hit ratio and failure of multiple eviction. The length of each CDL can be between 4 and 16, because while we examine the real-world workloads, most of the write requests are less than four-times the page size (4 kB × 4 kB = 16 kB), and it is seldom that writes are greater than sixteen-times the page size (16 kB × 4 kB = 64 kB). Algorithm 1 only describes how to move a cold dirty page, and when to move is described in the next subsection by combining with the victim selection algorithm of the buffer.

In Fig. 4, where it is assumed that there are four channels in the SSD, the evicting pages are moved from the working LRU list to CDLs on a one-by-one basis in accordance with Algorithm 1. When we need to move the page 1,002, since it is not consecutive to any pages in all other CDLs and each CDL has at least a page, CAWR flushes the LRU pages (4, 309, 55, and 100) of each list. After flushing, there is an empty list of CDLs and the page 1,002 can then be moved in. Since the next page of 7 is the consecutive page to 6, it can be moved into the

Algorithm 1. to CDLs

```
1: WL: working LRU list
2: item: the LRU dirty pate in WL
3: CDLs: cold dirty LRU list organized into channels
4: moved = false;
5: for (i = 0; i < ChannelSize; i++) do
6:   if ((CDLs[i] is not empty) & (CDLs[i] is not full)
       & (isSequence(item, CDLs[i]))) then
7:       remove item from WL;
8:       move item into MRU position in CDLs[i];
9:       moved = true;
10:      break;
11:  end if
12: end for
13: if (!moved) then
14:  for (i = 0; i < ChannelSize; i++) do
15:    if (CDLs[i] is empty) then
16:      remove item from WL;
17:      move item into MRU position in CDLs[i];
18:      moved = true;
19:      break;
20:    end if
21:  end for
22: end if
23: return moved;
```

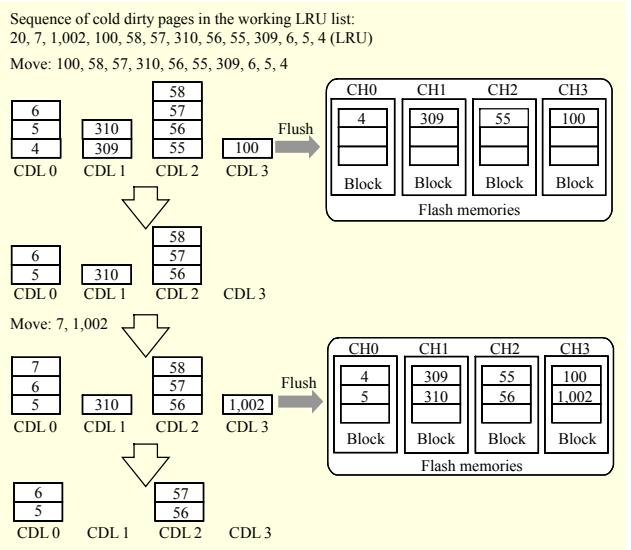

Fig. 4. Example of flushing pages of CAWR.

very list that contains page 6. As there is no place for the page of 20, CAWR again flushes the pages of 5, 310, 56, and 1,002. We can see that by flushing pages in each page cluster, CAWR changes the pattern of multiple evicted pages for a parallel write.

3. Clean-First Page-Level CAWR

In this subsection, we illustrate the overall design of a page-

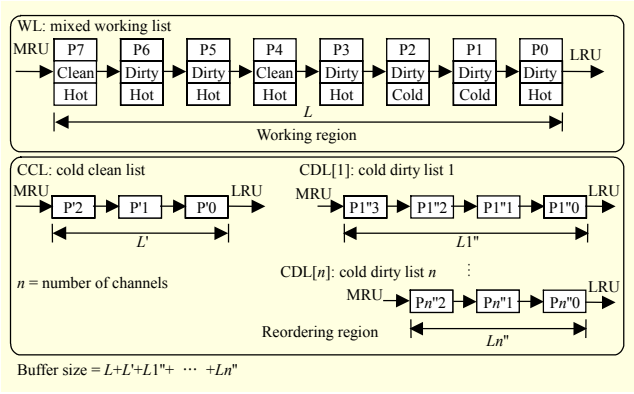

Fig. 5. Structure of page-level CAWR.

level CAWR, which cooperates with the extended CCF-LRU page-level working region. The working region organizes the buffered data in a list of pages, where each page contains a clean/dirty bit and a hot/cold bit. Figure 5 shows the structure, in which WL represents the working list of the working region. There are several CDLs and a cold clean list (CCL) in the reordering region, where the number of CDLs (n) is equal to the number of channels. As we have mentioned in the previous subsection, a maximum length for each CDL is defined; for example, four, and if the device has four channels, then the maximum size of a CDL is 4 kB × 4 kB × 4 kB = 64 kB. The sizes of the WL, CCL, and CDLs are dynamically changed, the sum of which is equal to the size of the cache.

When a replacement is needed, the victim selection starts, as illustrated in Algorithm 2. The algorithm first evicts the LRU page in the CCL (lines 5–6), and since a clean page can be directly removed from the buffer, only one victim is selected from the list at a time. If the clean list is empty, then the CDLs are checked, trying to evict multiple pages whose number is equal to that of the number of channels at any given time (lines 9–12). If the clean list and at least one dirty list are empty, then the mixed working LRU list is scanned from the LRU position, to move some cold pages to the cold lists (lines 14–29). The dirty page in the working LRU list is given a second chance; a clean page is directly moved to the CCL, while a dirty page is marked from "hot" to "cold" in the first scan and will be moved to the CDLs in the next scan. The moving between the working list and the cold lists ends when the CDLs cannot be moved in (none of the dirty lists are empty and the current cold dirty page is not consecutive to the pages in the lists). After the moving step, the selection algorithm is called again.

Algorithm 2. Victim selection

```
1: WL: working LRU list
2: CCL: cold clean LRU list
3: CDLs: cold dirty LRU list organized into channels
4: while (True) do
5:   if (CCL is not empty) then
6:     return the LRU pate in CCL;
7:   else
8:     if (All CDLs are not empty) then
9:       for (i = 0; i < ChannelSize; i++) do
10:        victims[i] = the LRU page in CDLs[i];
11:      end for
12:      return victims;
13:    else
14:      item = the LRU page in WL;
15:      while (item is Dirty) do
16:        if (cold-flag of item is set) then
17:          if (!toCDLs(item)) then
18:            break;
19:          end if
20:        else
21:          set cold-flag of item;
22:          move item to MRU position in WL;
23:        end if
24:        item = the LRU page in WL;
25:      end while
26:      if (item is not dirty) then
27:        remove item from WL;
28:        move item into MRU position in CCL;
29:      end if
30:    end if
31:  end if
32: end while
```

Table 1. Comparison of buffer management policies.

Technique	R/W asymmetry	Write pattern	Parallelism
Block-level buffer	No	Yes	No
CFLRU, LRU-WSR, CCF-LRU	Yes	No	No
REF	No	Yes	No
Gating buffer, CAVE	No	No	Yes
CAVE + block-level	No	Yes (partial)	Yes
CAVE + page-level	No	No	Yes
CAWR (proposed)	Yes	Yes	Yes

4. Comparison of Buffer Management Policies

The CAWR clusters the dirty pages to be evicted into multiple lists. Each list, in turn, corresponds to a channel of the SSD. The data to be written to flash memories is only flushed from the reordering region, such flushing only occurring when every channel has at least one outstanding page write. The CAWR flushes the cold clean page first. It then clusters the cold dirty pages and flushes them in multiples during each eviction. Table 1 compares it with previous techniques.

V. Performance Evaluation

1. Evaluation Setup

To evaluate the proposed mechanism, we use FlashSim [12] as our simulation framework, implementing an on-disk buffer with our page-level CAWR and an ideal page-level address mapping FTL that extends the FTL described in [13]. To allow parallel operations among independent channels, our FTL keeps a write point for each channel. The multiple evicted pages from the buffer are allocated to multiple write points so that they can be operated in independent channels simultaneously. The 1-channel garbage collection [2] is triggered when the number of free blocks in a channel gets to a certain threshold.

Flash memory chips are organized in four channels. Table 2 presents the parameters of the simulated flash memory. Two financial workloads provided by Storage Performance Council [14] and three MSR-series workloads [15] collected from different production servers [16] in Microsoft's data centers are employed as realistic I/O intensive traces. Financial workloads represent the random access pattern; and the others represent the sequential pattern. The size of the simulated device is fixed and referenced LBAs larger than this size are filtered. Table 3 lists the characteristics of the workloads.

An initial process is simulated to fill the device and warm up the FTL algorithm, writing all valid LBAs to the device. This process generates an aged SSD in which cleaning is more easily invoked. Statistics collection begins as the traces are loaded. As the hit ratio is the most well-known factor used to evaluate a buffer management scheme, we evaluate the hit ratios of the previously mentioned page-level buffer management schemes with and without CAWR. CAWR tries to preserve spatial locality so as to improve garbage collection efficiency. So, we adopt the total number of cleaning operations (that is, the sum of the valid page migrations and block erases) to evaluate the garbage collection efficiency. Furthermore, the average response time, which covers both the exploitation of multichannel architecture and the reduction of garbage collection overhead, is used to estimate the performance of the SSD. Finally, we calculate the coefficient of variation among the total numbers of read, write, and erase operations of each channel to analyze the problem of load balance among channels.

Table 2. Parameters of simulated flash memory.

Parameter	Value
Page read to register	60 μs
Page program (write) from register	800 μs
Block erase	1.5 ms
Page size	4 kB

Table 3. Characteristics of workload traces.

	Write (%)	Average interval	Write size (%)			
			4 kB	4 kB –16 kB	16 kB –64 kB	> 64 kB
Financial1	76.84	0.0082	86.58	10.63	2.76	0.03
Financial2	17.66	0.0110	87.82	10.11	1.88	0.18
rsrch_0	92.56	0.4570	67.71	25.35	6.94	0.00
src_20	89.81	0.4486	70.12	23.06	6.83	0.00
stg_0	84.81	0.2978	73.33	18.62	9.05	0.00

2. Evaluation Results

A. Hit Ratio

In the first experiment, we compare the proposed page-level CAWR with the conventional CCF-LRU in terms of hit ratio. Experimental results are shown in Fig. 6. From the figure, we observe that the proposed method almost achieves the same hit ratios as the conventional method under different buffer sizes. For the read dominant trace of Financial2, the hit ratio of the conventional method is slightly higher than the proposed method. The reason is as follows. CAWR must collect multiple dirty pages for parallel eviction. However, under read-intensive workloads, it needs to wait for a long time to collect enough dirty pages. In this period of time, a large number of clean pages have been replaced. As a result, the hit ratios of reads are degraded.

B. Cleaning Operations

In the second experiment, we compare the proposed method with the CCF-LRU that allocates channels in a round-robin manner in terms of the number of reclaiming operations. CCF-LRU evicts one page at a time, and the subsequent pages are then written to different channels in a round-robin manner. Figure 7 shows the normalized number of valid page migrations of the proposed method, with respect to the conventional one. As the buffer size increases, the write requests are reduced, so the total number of valid page migrations of garbage collection is also reduced for both methods; here, the reduction of the proposed method is greater than the conventional one. The proposed method reduces page migration operations by 7% for a 1 MB buffer and by 10% for a 16 MB buffer, on average, compared to the conventional method with the same buffer size. For the

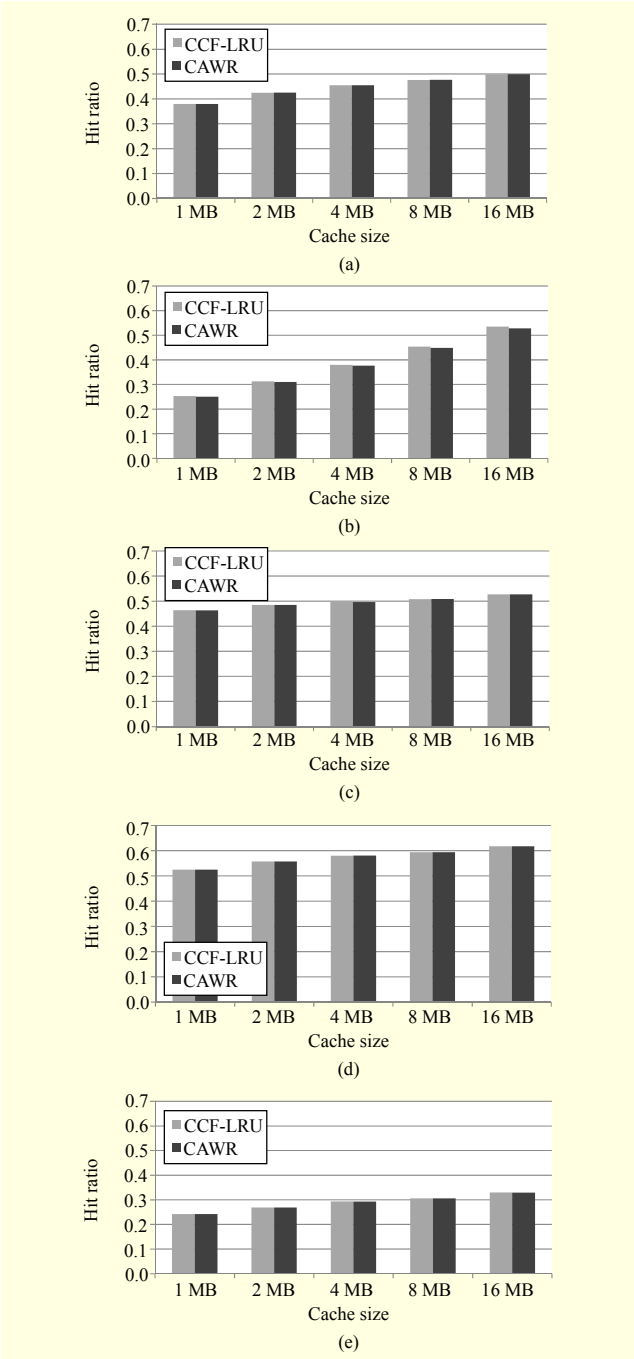

Fig. 6. Hit ratio of buffer: (a) Financial1, (b) Financial2, (c) rsrch_0, (d) src2_0, and (e) stg_0.

financial workloads with the random access pattern, the difference is small because most writes only contain one page and the reordering step does not have such a great effect. For the workloads with more sequential writes, CAWR reduces more valid page migrations, by 15% at most. There is a small reduction in the number of erase operations in the proposed method, by 5% at most; thus, the result is not illustrated as a figure here.

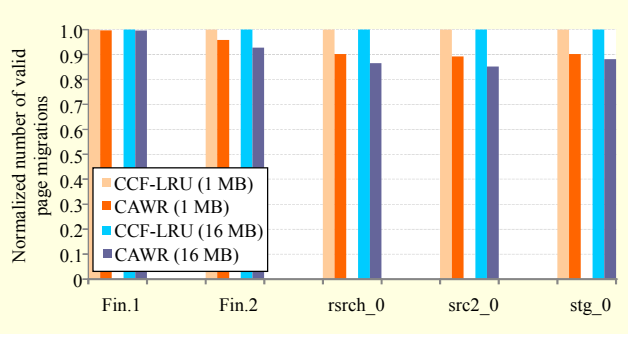

Fig. 7. Normalized number of valid page migrations.

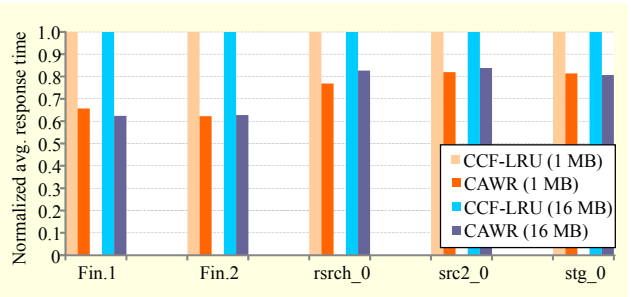

Fig. 8. Normalized average response time.

C. Average Response Time

In the experiment, we compare the average response time of the proposed method to that of the conventional CCF-LRU policy with round-robin allocation of channels. Figure 8 shows the normalized number of average response time, with respect to the conventional method. The results show that the response time of the proposed method is always lower than that of the conventional method. The achieved reduction is 38% at most for the Financial2 workload, and 26% on average. With the cache size increased, the reduction varies more or less. This improvement comes mainly from the multiple pages eviction, which utilizes the I/O parallelism.

D. Load Balance

In the experiment, we collect the numbers of read, write, and erase operations of each channel, calculating their coefficients of variation to reflect whether the loads among channels are balanced. The coefficient of variation represents the ratio of the standard deviation to the mean, and it is a useful statistic for comparing the degree of variation. In Table 4, the reads and the writes not only include host reads and writes but also the migrating reads and writes introduced by the internal garbage collection process. Both the round-robin allocation policy and the multi-eviction policy of the proposal achieve almost balanced loads for writes and erases. For read operations, the values are relatively larger but still acceptable.

Table 4. Coefficients of variation of reads, writes, and erases.

Workloads	Policies	Reads	Writes	Erases
Financial1	Round-robin	0.0095	0.0031	0.0033
	CAWR	0.0023	0.0045	0.0043
Financial2	Round-robin	0.0210	0.0132	0.0466
	CAWR	0.0304	0.0035	0.0289
rsrch_0	Round-robin	0.1263	0.0006	0.0148
	CAWR	0.1272	0.0016	0.0060
src2_0	Round-robin	0.0974	0.0051	0.0109
	CAWR	0.0980	0.0046	0.0350
stg_0	Round-robin	0.0315	0.0008	0.0032
	CAWR	0.0316	0.0026	0.0041

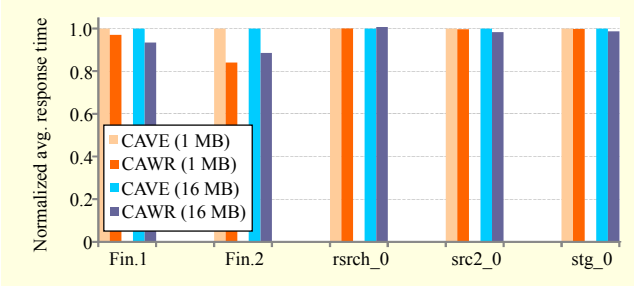

Fig. 9. Normalized average response time of CAVE and CAWR.

E. Comparison with CAVE

We also make a comparison of the proposed method with the CCF-LRU policy combined with CAVE. A small modification of CAVE is made to support a read-write buffer; when there are enough cold dirty pages for all channels, CAVE evicts. Since CCF-LRU does not detect the write pattern, CAVE allocates channels to each page in a round-robin manner, no matter whether the pages are consecutive or not. The conventional CCF-LRU policy mentioned in experiment B also uses this round-robin allocation manner, so the conventional CCF-LRU and this page-level CAVE have similar data distributions and garbage-collecting activities. The experimental results confirm that they achieve a similar number of valid-page-migration and erase operations. Therefore, the difference in garbage-collection overhead between CAVE and CAWR is similar to that between the conventional CCF-LRU and CAWR, which has been discussed in experiment B. The proposed method behaves better in terms of endurance, and it reduces page migration operations by 10% for a 16 MB buffer, on average.

The proposed method and CAVE both utilize the parallelism among channels. Figure 9 shows the normalized number of average response time of CAWR with respect to CAVE. The response time is similar except for the financial workloads. The reason is complicated. Firstly, arranging sequential write pages into the same block reduces the clean cost but may increase the latency of reads, since read pages must be fetched from the block one by one; thus, sequential read cannot take advantage of the parallelism among channels. Second, the arrival intensity of IO requests is another factor, because a higher intensity means that the response time is more sensitive to the garbage-collection overhead. The financial workloads have fewer sequential reads and higher arrival intensity; thus, the proposed method behaves better than CAVE, because the proposed method optimizes the garbage collection remarkably, while the read performance is unlikely to be impaired by the limited parallelism. For the other workloads, although the proposed CAWR reduces more valid page migrations, the proposed method ties CAVE for the increased time of sequential reads and relative lower arrival intensity. And, for the rsrch_0 workload, CAVE even slightly outperforms the proposed mechanism by 0.09% for a 1 MB buffer and 0.75% for a 16 MB buffer, for the increased read time.

VI. Conclusion

In this paper, we proposed a channel-aware write reordering (CAWR) mechanism, which reorganizes the evicting page sequence and evicts multiple victims to increase I/O parallelism, as well as preserving the spatial locality. CAWR detects the consecutive page at the end of the buffer; thus, it can be well applied even though the buffer replacement policies do not address the page pattern. The experimental results show that even though the CAWR collects and evicts more victims each time than the conventional method, this cannot critically affect the hit ratio of a buffer. By reordering the page sequence, the number of reclaiming operations is reduced, and by multiple-pages eviction, the performance is improved for SSDs with independent multichannel architectures.

Keeping a sequential write pattern increases the time of a sequential read; however, we think that it is not a good idea to sacrifice endurance for the sake of performance. Other existing methods, such as redundancy encoding, duplication, and cost-aware buffer replacement, have been earmarked by us for future work in the hope that we will be able to combine these with our scheme in an attempt to compensate the read performance.

References

[1] L.-P. Chang, Y.-H. Huang, and C.-Y. Wen, "On the Management of Multichannel Architectures of Solid-State Disks," *IEEE Symp. Embedded Syst. Real-Time Multimedia*, Taipei, Taiwan, Oct. 13–

14, 2011, pp. 37–45.
[2] S.K. Park et al., "CAVE: Channel-Aware Buffer Management Scheme for Solid State Disk," *ACM Symp. Appl. Comput.*, Taichung, Taiwan, Mar. 21–25, 2011, pp. 346–353.
[3] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings," *ACM Int. Conf. Archit. Support Programming Languages Operating Syst.*, Washington, DC, USA, Mar. 7–11, 2009, pp. 229–240.
[4] Y. Hu et al., "Achieving Page-Mapping FTL Performance at Block-Mapping FTL Cost by Hiding Address Translation," *IEEE Symp. Mass Storage Syst. Technol.*, Incline Village, NV, USA, May 3–7, 2010, pp. 1–12.
[5] H. Jo et al., "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2 , May 2006, pp. 485–493.
[6] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *USENIX Conf. File Storage Technol.*, San Jose, CA, USA, Feb. 26–29, 2008, pp. 239–252.
[7] S. Kang et al., "Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices," *IEEE Trans. Comput.*, vol. 58, no. 6, June 2009, pp. 744–758.
[8] S.-Y. Park et al., "CFLRU: A Replacement Algorithm for Flash Memory," *IEEE Int. Conf. Compilers Archit. Synthesis Embeded Syst.*, Seoul, Rep. of Korea, Oct. 23–25, 2006, pp. 234–241.
[9] H. Jung et al., "LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, Aug. 2008, pp. 1215–1223.
[10] Z. Li et al., "CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory," *IEEE Trans. Consum. Electron.*, vol. 55, no. 3, Aug. 2009, pp. 1351–1359.
[11] D. Seo and D. Shin, "Recently-Evicted-First Buffer Replacement Policy for Flash Storage Devices," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, Aug. 2008, pp. 1228–1235.
[12] Y. Kim et al., "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives," *Int. Conf. Adv. Syst. Simulation*, Porto, Portugal, Sept. 20–25, 2009, pp. 125–131.
[13] A. Birrell et al., "A Design for High-Performance Flash Disks," Microsoft Research, Silicon Valley, CA, USA, Tech. Rep. MSR-TR-2005–176, Dec. 2005.
[14] SPC, *Storage Traces form Storage Performance Council*, SPC, 2009. Accessed May 22, 2013. http://traces.cs.umass.edu/
[15] SNIA, *Block Traces from SNIA IOTTA Repository*, SNIA, 2009. Accessed May 22, 2013. http://iotta.snia.org/traces/list/BlockIO
[16] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *USENIX Conf. File Storage Technol.*, San Jose, CA, Feb. 26–29, 2008, pp. 253–267.

Ronghui Wang received her BS and MS degrees in computer science from the College of Computer, National University of Defense Technology (NUDT), Changsha, China, in 1999 and 2002, respectively. She is a senior engineer at the State Key Laboratory of Astronautic Dynamics, Xi'an, China, and she is currently enrolled in the NUDT's PhD program in electronic science. Her research interests include software/hardware co-design, mass storage architecture, and solid-state storage systems.

Zhiguang Chen received his BS degree in computer science and technology from the Harbin Institute of Technology, China, in 2007. He achieved his MS and PhD degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China, in 2009 and 2013, respectively. Now, he is an associate professor at the College of Computer, NUDT. His current research interests include distributed file systems, network storage, and solid-state storage systems.

Nong Xiao received his BS, MS, and PhD degrees in computer science from the College of Computer, National University of Defense Technology (NUDT), Changsha, China. Currently, he is a professor at the College of Computer, NUDT. His current research interests include large-scale storage systems, network computing, and computer architecture.

Minxuan Zhang received his BS, MS and PhD degrees in computer science from the College of Computer, National University of Defense Technology (NUDT), Changsha, China. Currently, he is a professor at the College of Computer, NUDT. His research interests include high-performance microprocessor design, high-performance parallel computing, and computer architecture.

Weihua Dong received his BS degree in computer science from the College of Computer, National University of Defense Technology, Changsha, China, in 1999. Currently, he is a senior engineer at the State Key Laboratory of Astronautic Dynamics, Xi'an, China. His research interests include high-performance computing systems, software architecture, and software engineering.