CBM: A Cooperative Buffer Management for SSD

Qingsong Wei
Data Storage Institute, A-STAR
Singapore
WEI_Qingsong@dsi.a-star.edu.sg

Cheng Chen
Data Storage Institute, A-STAR
Singapore
CHEN_Cheng@dsi.a-star.edu.sg

Jun Yang
Data Storage Institute, A-STAR
Singapore
YANG_Jun@dsi.a-star.edu.sg

Abstract— **Random writes significantly limit the application of Solid State Drive (SSD) in the I/O intensive applications such as scientific computing, Web services, and database. While several buffer management algorithms are proposed to reduce random writes, their ability to deal with workloads mixed with sequential and random accesses is limited. In this paper, we propose a cooperative buffer management scheme referred to as CBM, which coordinates write buffer and read cache to fully exploit temporal and spatial localities among I/O intensive workload. To improve both buffer hit rate and destage sequentiality, CBM divides write buffer space into *Page Region* and *Block Region*. Randomly written data is put in the *Page Region* at page granularity, while sequentially written data is stored in the *Block Region* at block granularity. CBM leverages threshold-based migration to dynamically classify random write from sequential writes. When a block is evicted from write buffer, CBM merges the dirty pages in write buffer and the clean pages in read cache belonging to the evicted block to maximize the possibility of forming full block write.**

CBM has been extensively evaluated with simulation and real implementation on OpenSSD. Our testing results conclusively demonstrate that CBM can achieve up to 84% performance improvement and 85% garbage collection overhead reduction compared to existing buffer management schemes.

Keywords- flash memory; cooperative buffer management; buffer hit ratio; write sequentiality

I. INTRODUCTION

Flash memory technology has been changing the paradigm of storage system over the past few years, receiving strong interest in both academia and industry [1-6]. Flash memory has been traditionally used in portable devices. As price drops and capacity increases, this technology has made huge strides into enterprise storage space in the form of SSD. As alternative for hard disk dirve (HDD), SSD outperforms its counterpart by orders of magnitude with even lower power consumption.

Although SSD shows its attractive worthiness especially on improving random read performance due to no mechanical characteristic, however, it could suffer from random write when it is applied in the I/O intensive applications. Writes on SSD are highly correlated with access patterns. The properties of NAND flash memory result in random writes being much slower than the sequential writes. Further, the durability issue of SSD is a more serious problem [28,30]. NAND flash memory can incur only a finite number of erases for a given block. Therefore, increased erase operations due to random writes shortens the lifetime of a SSD. Experiments in [9] show that random write workload could make SSD wear out over hundred times faster than sequential write workload. Reducing random writes is desirable to both performance and lifetime.

For the past decade, Non-volatile Memory (NVM) technologies have been under active development, such as *phase-change memory (PCM)* [7], *spin torque transfer magnetic RAM (STT-MRAM)* [8] and *Resistive RAM (RRAM)* [38]. They are persistent, fast and byte-addressable. For example, PCM chip with 2Gbits capacity is available. A commercial 64Mbits STT-MRAM chip with DDR3 interface was announced in 2013. Though currently unfit for large scale application due to cost, it is feasible to use small amount of NVM in SSD as write buffer to reduce latency and reshape access pattern to provide sequential writes for flash.

In particular, many block-level algorithms that manage NVM write buffer have been proposed, i.e. FAB[20], BPLRU[14], and LB-CLOCK[21]. These algorithms exploit spatial locality and group resident pages in write buffer so as to flush block as sequential as possible. However, they suffer from buffer pollution and early eviction issues due to coarse management granularity, resulting in low buffer utilization and low hit ratio. To overcome this issues, a hybrid buffer management BPAC[26] is proposed to leverage both temporal and spatial localities. However, it is not efficient in handling changing workloads. Forming sequential write is becoming increasingly difficult because of following two facts. First, widely used Multi-Level Cell (MLC) flash memory block can contain up to 256 or 512 pages. Increasing block size results in difficulty in forming full block write. Second, I/O intensive workload mixed with random and sequential writes aggravates this difficulty. The ability of existing write buffer management schemes to deal with I/O intensive workloads is limited.

Read and write are always mixed in the real applications. Besides NVM write buffer, SSD is also equipped with a read cache (i.e. DRAM) to speed up read performance. Usage patterns exhibit block-level temporal locality: the pages in the same logical block are likely to be accessed (read/write) again in the near future. However, above mentioned algorithms BPLRU, LB-LOCK and BPAC are only designed for write buffer, ignoring localities among read requests. Current write buffer and read cache work separately without cooperation.

In this paper, we propose a cooperative buffer management scheme referred to as CBM, which coordinates write buffer and read cache to fully exploit temporal and spatial localities. To improve both buffer hit rate and destage sequentiality, CBM divides write buffer into *Page Region* and *Block Region*. Randomly written data is put in the *Page Region* at page granularity, while sequentially written data is stored in the *Block Region* at block granularity. We give preference to pages with high temporal locality for staying in the *Page Region*, while blocks with high spatial locality in the *Block Region* are replaced first. Buffered pages in the *Page Region* are dynamically migrated to the *Block Region* if the number of pages of a block reaches a threshold, whose value can be adaptively adjusted for different workloads. When a block is evicted from write buffer, CBM merges the dirty pages in write buffer and the clean pages in read cache belonging to the evicted block to maximize the possibility of forming full block write. Our experiments on simulation and real OpenSSD show that proposed CBM not only improves performance, but also reduces block erasure, thus extends SSD lifetime.

We made the following contributions in the paper.

- We design a buffer management scheme to make full use of both temporal and spatial locality by coordinating write buffer and read cache.
- A hybrid write buffer management is designed to improve buffer hit and destage sequentiality by managing random writes at page level and sequential writes at block level.
- Dynamic threshold-based migration and workload classification is proposed to classify random and sequential writes for changing workloads.
- We have implemented and evaluated CBM on real OpenSSD platform. Benchmark results show that proposed CBM can achieve up to 84% performance improvement and 85% garbage collection overhead (block erasure) reduction, compared with the state-of-the-art buffer management schemes.

The rest of this paper is organized as follows. Section II provides an overview of background and related works in the literature. In Section III, we present details of cooperative buffer management scheme (called CBM). Section IV presents simulation results. Evaluation on real OpenSSD is presented in Section V. Conclusion is summarized in Section VI.

II. BACKGROUND AND RELATED WORK

A. SSD and Flash Translation Layer

SSDs use NAND flash memory as storage medium (See Fig. 1). NAND flash memory can be classified into Single-Level Cell (SLC) and Multi-Level Cell (MLC) flash. A SLC flash memory cell stores only one bit, while a MLC flash memory cell can store two bits or even more. NAND Flash memory is organized as blocks. Each block consists of 64 to 256 pages. Each page has a 2KB or 4KB data area and a metadata area (e.g. 128 bytes). Flash memory performs read and write in the unit of page and erase in block unit. Blocks must be erased before they can be reused. In addition, each block can be erased only a finite number of times. A typical MLC flash memory has around 10,000 erase cycles, while a SLC flash memory has around 100,000 erase cycles[11,13].

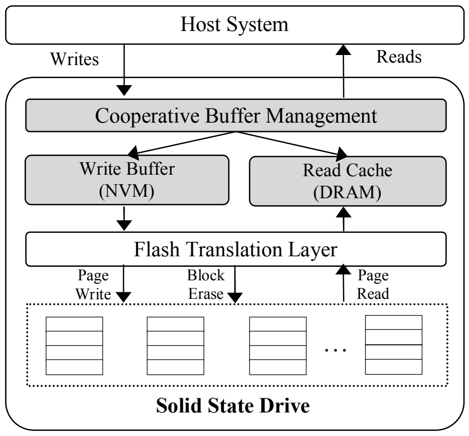


Fig. 1. The proposed CBM is applied inside SSD

SSD controller uses NVM as write buffer to improve performance. The write buffer can delay the write requests which directly operate on flash memories, such that the response time of operations could be reduced. Additionally, it also can reorder the write request stream to make the sequential write flushed first when the synchronized write is necessary. SSD also uses DRAM as read cache to speed up read accesses.

Flash Translation Layer (FTL)[10] allows hosts to access flash memory in the same way as conventional disk drives. The FTL performs address mapping, garbage collection and wear leveling. Hybrid FTL is widely used FTL, such as BAST[31], FAST[32] and LAST[10]. It uses a block-level mapping to manage most data blocks and uses a page-level mapping to manage a small set of log blocks, which works as a buffer to accept writing requests [10]. When there is no available log block, It selects a victim log block and merge it with its corresponding data block(s)—a *merge* operation. While executing the merge operation, multiple page copy operations (to copy valid pages from both the data block and log block to a new data block) and erase operations (to create a free block) are invoked. Therefore, merge operations seriously degrade the performance of the SSD because of these extra operations. There are three types of merge operations in hybrid FTL: *switch merge*, *partial merge*, and *full merge*. Among them, *switch merge* is the most economical merge operation, while *full merge* is the least economical merge operation. Hybrid FTLs incur expensive full merges under I/O intensive workloads. Since the buffer memory is positioned at a level higher than FTL and receives requests directly from host (See Fig. 1), reducing random writes through efficient buffer management is desirable to eliminate expensive full merges for hybrid FTL.

B. Workload Characteristics

Figure 2 shows the distribution of request size over ten real I/O intensive workloads which are made available by Storage Network Information Association (SNIA) [22]. The figure shows the cumulative percentage (shown on Y-axis) of requests whose sizes are less than a certain value (shown on X-axis). As shown in the Fig. 2, these workloads are mixed with small and sequential I/O. Most of requests are smaller than 64K, and few requests are bigger than 128K. Small request is much more popular than big sequential request. Our workload

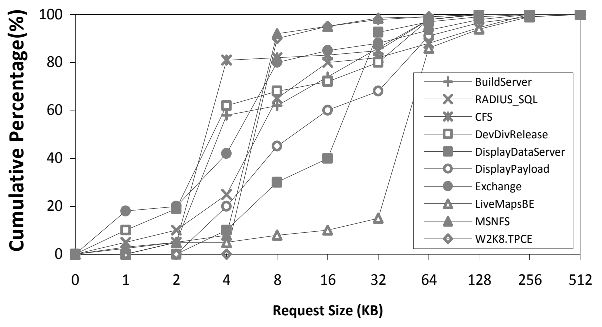


Fig. 2. I/O intensive workload characteristics

characterization study showed the prevalence of small random requests in I/O intensive application. Research works in [33,34] also highlighted that small random I/O dominates in modern scientific HPC applications.

C. Random Writes Issues

Figure 3 shows the write performance of Intel® X25-E Extreme SATA SSD by using IOmeter. We can see that sequential write is much faster than random write. For 4 KB request, random write speed is only 12.6 MB/second while sequential write is 52 MB/second. Further, performance of mixed workload with sequential and random writes (e.g. 80:20) is worse than that of pure random write. The result reflects that random writes consume significant resources and limit SSD performance.

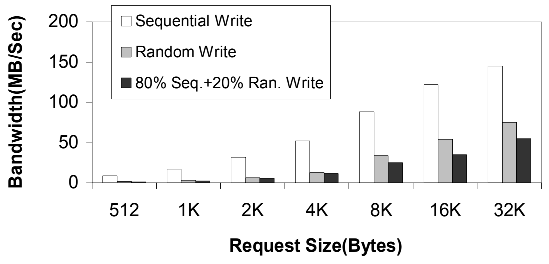


Fig. 3. Write performance of SSD

In addition, random write has following issues.

— *Shorten SSD Lifetime*. Increased erase due to random writes make a SSD wear out much faster than sequential writes.

— *High Garbage Collection Overhead*. Random writes are most likely to trigger garbage collection and result in higher overhead of garbage collection than sequential writes.

— *Internal Fragmentation*. If the incoming writes are randomly distributed over the logical block address space, all physical flash memory blocks will be fragmented [12].

— *Reduced Parallelism*. While striping and interleaving can improve performance for sequential writes, its ability to deal with random write is very limited [13].

As competing with HDD, SSD must retain its performance advantages even with low cost configurations, such as those with a small built-in memory (NVM and DRAM) and using MLC NAND. The increased density of the MLC flash significantly reduces the cost and increases the capacity of SSD. However, the write speed of MLC flash is more than three times slower than that of the SLC flash, making write a further expensive operation for the MLC flash. In addition, erase cycle of MLC flash is ten times less than that of the SLC flash. Therefore, the impact of random write on MLC flash is more significant than on SLC. Further, SSD's internal garbage collection can produce additional write operations. To this end, random writes should be minimized through efficient buffer management with high buffer space utilization.

D. Related Work

In current practice, there are several major efforts in buffer management, which can be classified either into a *page-based* or a *block-based* buffer management scheme.

Page-based buffer management organizes and replaces buffered data at page granularity. There are a large number of page-based disk buffer algorithms proposed such as LRU, CLOCK [23], WOW[29], 2Q [19], ARC [18] and DULO [15]. All these algorithms focus only on how to better utilize temporal locality to minimize page fault rate. However, direct application of these algorithms is inappropriate for SSD because spatial locality is unfortunately ignored. Clean first LRU (CFLRU) [24] is a page-based buffer management scheme for flash storage. It divides the host buffer space into *working region* and *eviction region*. Victim buffer pages are selected from the eviction region. To exploit the asymmetric performance of flash read and write operations, it chooses a clean page as a victim rather than dirty pages. CFLRU reduces number of writes by performing more reads. However, CFLRU does not optimize the sequentiality of evicted pages. In addition, it is proposed to be deployed in host, not in SSD.

Block-based buffer management such as FAB[20], BPLRU[14], PDU-LRU[27] and LB-CLOCK[21] exploits spatial locality to reshape the access pattern and provides sequential writes for flash memory. Resident pages in the buffer are grouped on the basis of their logical block associations. The flash aware buffer policy (FAB) [20] evicts a block having the largest number of cached pages. In case of a tie, it considers the LRU order. All the dirty pages in the victim group are flushed, and all the clean pages in it are discarded. This policy may results in internal fragmentation. The main application of FAB is in portable media players in which access pattern of write is sequential. Block Padding LRU (BPLRU) [14] uses block-level LRU, page padding, and LRU compensation to establish a desirable write pattern with RAM buffering. However it does not consider read requests. For completely random workload, BPLRU incurs a large number of additional reads caused by page padding, which significantly impact the overall performance. Large Block CLOCK (LB-CLOCK) [21] algorithm considers recency and block space utilization metrics to make cache management decisions. LB-CLOCK dynamically varies the priority between these two metrics to adapt to changes in workload characteristics.

These buffer schemes suffer buffer pollution and early eviction problems due to coarse granularity. In fact, they order the block just based on one or several hot pages, resulting in buffer pollution (the block may contain a large number of cold pages). When replacement happens, all pages of the victim block are removed simultaneously. However, some hot pages of the victim block may be accessed in the near future, resulting in early eviction. Increasing block size worsens the problems, especially for I/O intensive workloads.

Hybrid buffer management is proposed to combine both page-based and block based buffer managements. Block Page Adaptive Cache (BPAC) [26] is a hybrid write buffer management for SSD [9]. BPAC proposes dual-list to partition buffer space into a page-based list and a block-based list and exploits size-dependent and size-independent region to differentiate the low spatial locality cluster from high spatial locality ones. This mechanism uses PIRD to partition the page-list and block-list, and BIRD to partition size-dependent region and size-independent region. However, the proposed PIRD and BIRD are not efficient in handling fast changing workloads.

All these algorithms are only designed for write buffer, ignoring the localities among read operations.

III. COOPERATIVE BUFFER MANAGEMENT

We now present a novel buffer management scheme with cooperation between write buffer and read cache.

A. Overview

To fully utilize both temporal and spatial localities among workloads, the write buffer and read cache are managed in cooperative way as shown in Fig. 4. Read cache uses existing page-based algorithms such as LRU, 2Q and ARC to order the clean pages and decide which page to be evicted. Our proposed write buffer algorithm manages dirty pages in NVM. When NVM evicts pages, we merge the clean pages in DRAM to cooperatively flush pages as sequential as possible. Note we do not change the behaviors of read cache. We focus on presenting our design on write buffer in Section B.

B. Write Buffer Management

1) Hybrid space management

Because NVM is expensive, buffer algorithm for managing NVM should pursue high hit rate to save cost and absorb repeated random writes. Reordering small writes into sequential write is also an efficient way to reduce small writes. So sequentiality of evicted pages is also important. To improve hit ratio and destage sequentiality of write buffer, we divide NVM space into *Page Region* to store random writes with high temporal locality at page granularity and *Block Region* to store sequential writes with high spatial locality at block granularity (See Fig. 4). Buffered pages in the *Page Region* are dynamically migrated to the *Block Region* if the dirty page counter of a block reaches a threshold, whose value can be dynamically adjusted for different workloads.

A dirty page is either in *Page Region* or in *Block Region*. Both regions serve writing requests. Pages in *Page region* are organized as page LRU list (Note that any existing algorithm such as 2Q and ARC can be used to manage page region). When a page buffered in the *Page Region* is written, this page is placed at the head of the page LRU list.

Blocks in *Block Region* are organized and sorted on the basis of *Block Popularity* which is defined as block access frequency including writing of any pages of the block. When a

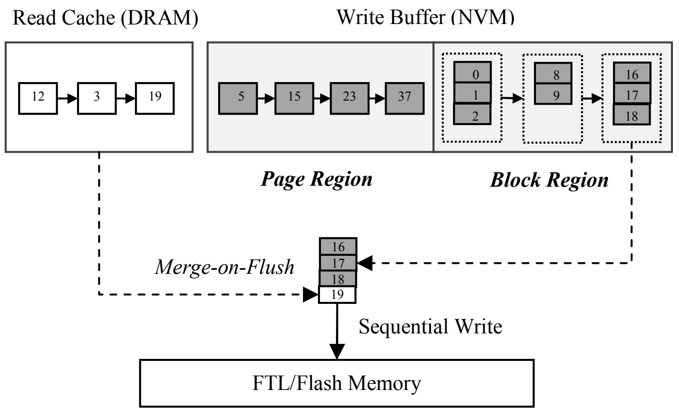


Fig. 4 Overview of CBM

page of a block is written, we increase the block popularity by one. Sequentially writing multiple pages of a block is treated as one block access instead of multiple accesses. Thus, block with sequential writes will have low popularity value, while block with random writes has high popularity value. The advantage of block popularity is its ability to fully capture block-level temporal and spatial localities. Note that it is different from block-LFU and block-LRU. The blocks in the *Block Region* are organized as Block Popularity List (BPL, see Fig. 5). The BPL list is sorted on the basis of block popularity, and dirty page counter. Block popularity is primary criterion to decide the position of a block in the BPL list. The most popular block stays at the BPL head, while the least popular block is put at the BPL tail. The blocks with same popularity value are further sorted based on the dirty page counter. Block with bigger dirty page counter will stay behind in the BPL list.

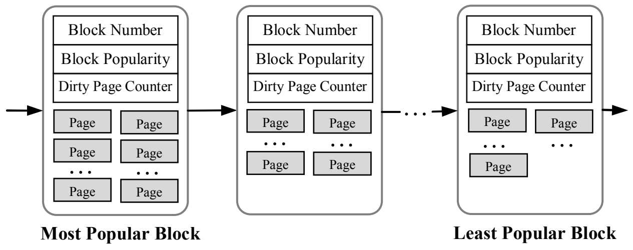


Fig. 5. Block Popularity List (BPL)

Therefore, the temporal locality among random writes and spatial locality among sequential writes can be fully exploited by *Page Region* and *Block Region* respectively.

2) Replacement and Flush Policy

This paper views negative impacts of random write on flash memory's lifetime and performance as penalty. The cost of sequential write miss is much lower than that of random write. Random writes will be frequently updated because it is more popular than sequential writes. When replacement happens, unpopular sequential data should be replaced instead of popular random data. Keeping popular data in write buffer as long as possible can minimize the penalty. For this purpose, we give preference to random written pages for staying in the *Page Region*, while sequential written pages in *Block Region* are replaced first.

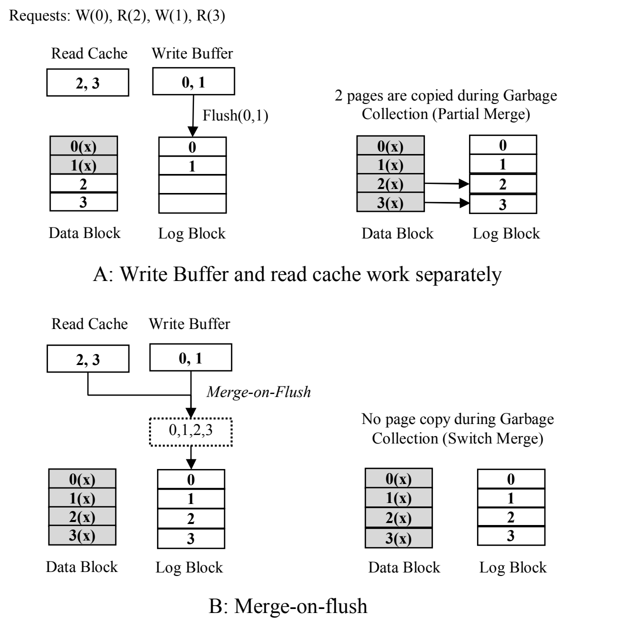

Requests: W(0), R(2), W(1), R(3)
Read Cache
Write Buffer
2, 3
0, 1
Flush(0,1)
2 pages are copied during Garbage Collection (Partial Merge)
0(x)
1(x)
2
3
0
1
2(x)
3(x)
Data Block
Log Block
A: Write Buffer and read cache work separately
Merge-on-Flush
0,1,2,3
No page copy during Garbage Collection (Switch Merge)
B: Merge-on-flush

Fig. 6. Merge-on-flush increases write sequentiality and reduces garbage collection overhead

The least popular block in the *Block Region* is selected as victim. If more than one block has the same least popularity, a block having the largest number of dirty pages is selected as a victim. Only if the *Block Region* is empty, we select the LRU page as victim from the *Page Region,* where the pages belonging to the same block will be replaced and flushed. This policy tries to avoid single page flush which has high impact on garbage collection and internal fragmentation.

Once a block is selected as victim, it is merged with clean pages in read cache before it is flushed to FTL if the dirty page count is bigger than clean page count. We call it a *merge-on-flush* operation. If the read cache contains clean pages belonging to the victim block, the dirty pages and clean pages are merged and flushed into flash memory sequentially. Note that the clean pages are still kept in read cache. This policy improves the possibility to form full block write and reduce garbage collection overhead (See Fig. 6). BPLRU uses page padding to read rest pages from flash if the victim block is not a full block. For random dominant workload, BPLRU needs to read a large number of additional pages, which impact the overall performance. Unlike BPLRU, CBM leverages block-level temporal locality among reads and writes to naturally form sequential block. Thus more sequential writes are passed to flash memory. The flash memory is then able to process the writes with stronger spatial locality more efficiently.

3) *Threshold-based Migration*

Buffer data in the *Page Region* will be migrated to the *Block Region* if the number of dirty pages in a block reaches the threshold, as shown in Figure 7. With filter effect of the threshold, random written pages will stay in the *Page Region*, while the sequential blocks reside in the *Block Region*. Therefore, temporal locality among random writes and spatial locality among sequential writes can be fully utilized in the hybrid buffer management.

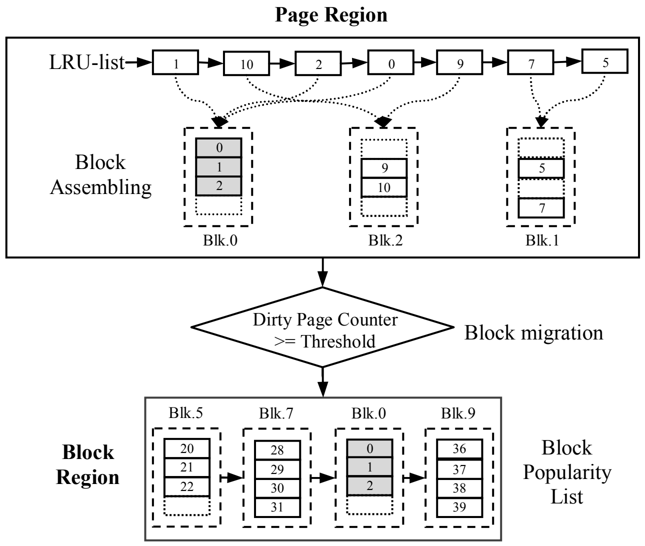


Fig. 7 Threshold-based Migration. Grey boxes denote that a block is found and migrated to Block Region.

4) *Dynamic Threshold*

Our objective is to improve destage sequentiality, at the meantime maintain high buffer space utilization. The migration balances the two objectives. In order to investigate the proper threshold value, we tested the effects of different threshold value through repetitive experiments over a set of workloads. However, we found in our experiments that it is difficult to find an average well-performed value for all types of workloads. Different threshold values should be set to achieve optimal results even for the same workload. Statically setting threshold value cannot adapt to enterprise workloads with complex features and interleaved I/O requests.

We realized that the value of threshold is highly dependent on workload features. For random dominant workload, a small value is suitable because it is difficult to form sequential blocks. For sequential dominant workload, a large value is desirable because a lot of partial blocks instead of full blocks will be migrated from page region to block region if a small threshold is set.

Dynamic threshold is achieved by using the following heuristic method. We use $THR_{migrate}$ to denote the threshold. Clearly, the value of $THR_{migrate}$ is from 1 to the total number of pages in a block. N_{block} and N_{total} represent the size of *Block Region* and total write buffer, respectively. The value of the ratio between N_{block} and N_{total} is under following constraint, which is used to control whether to enlarge or reduce the threshold.

$$\frac{N_{block}}{N_{total}} \leq \theta \qquad (1)$$

Where θ is a parameter configured based on write buffer (NVM) size. Since the *Block Region* is used to store block candidates whose pages are sequential enough for replacement and flush, the size of the *Block Region* should be much smaller than the size of the *Page Region*. We set the value of θ as 10%.

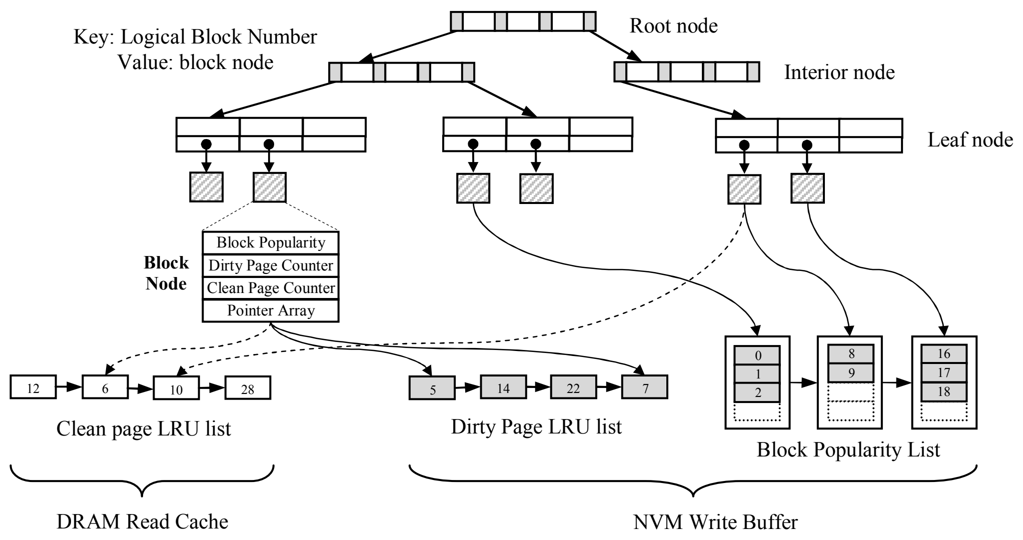


Fig. 8. Global Block B+tree

Initially, $THR_{migrate}$ is set as 2. If the value of N_{block}/N_{total} is becoming larger than θ, it indicates that the size of the *Block Region* breaks the above constraint. To reduce the size of the *Block Region*, a large value of $THR_{migrate}$ is required to increase the difficulty of migrating pages from *Page Region* to *Block Region*. Then, the value of $THR_{migrate}$ will be doubled until N_{block}/N_{total} is less than θ. On the other hand, the value of $THR_{migrate}$ will be halved if the *Block Region* is empty.

The dynamic method can properly adjust the threshold value based on different workload features. With our experiments, the dynamic threshold performs well with various buffer size imposed by different application sets.

C. *Management Data Structure*

CBM uses a *Global Block B+tree* to maintain the block association across the read cache, write buffer's *Page Region* and *Block Region*. The *Global Block B+tree* uses logical block number as key. A data structure called *Block Node* is introduced to describe a block in terms of block popularity, dirty page counter, clean page counter, and pointer array. The pointer array points to the pages belonging to the same block in the read cache, write buffer's *Page Region* and *Block Region* (See Fig. 8).

Upon arrival of an access, the logical page number is used to calculate the corresponding logical block number (the logical page number is divided by the block size). Then CBM searches the *Global Block B+tree* using the logical block number. If the block exists in the *Global Block B+tree*, we update the *Block Node* including block popularity, dirty page counters, clean page counter and pointer. Otherwise a new block node is created.

By using the *Block B+tree*, the pages belonging to the same block can be quickly searched and located. Meanwhile, the space overhead of the *Block B+tree* is limited. As shown in the Fig. 8, the *Global Block B+tree* generally includes three parts: block nodes, leaf nodes and interior nodes (including root node). In order to analyze the space overhead, we first make following assumptions: (i) Integer and pointer consumes 4 bytes, (ii) Fill factor of *Global Block B+tree* is at least 50% [25], (iii) the total size of interior nodes is half of the total size of leaf nodes (in practice, the number of interior nodes is much smaller than leaf nodes. Fan out of B+tree is 133 on average [25]), (iv) every block node consumes 16 bytes when pointer array only includes one pointer. In this case the number of the block nodes is max.

When the full length of write buffer and read cache is L pages, the size of write buffer and read cache is L*2*1024bytes (2KB/page). Accordingly, the number of block nodes is L (*assumption iv*), the total size of block nodes is L*16 (*assumption iv*), the total size of leaf nodes is L*2*8 (one unit corresponding to one block node in leaf node consumes 8 bytes), and the total size of interior nodes is L*2*4 (*assumption iii*). Therefore, the maximum space overhead of *Block B+tree* is L*40, which is less than 2% of the size of write buffer and read cache.

IV. SIMULATION EVALUATION

A. Experiment Setup

1) *Trace-driven Simulator*

We built a trace-driven buffer simulator by modifying Flashsim [35] which interfaces with DiskSim[16]. Four buffer schemes are implemented: 1) our CBM, 2) BPLRU [14], 3) FAB [20], and BPAC[26]. We use FAST as FTL and allocate 3% of the total flash memory as log-blocks [13]. We use DRAM is read cache and STT-MRAM as write buffer. Configuration values taken from [7][11] are listed in the Table I. For fair comparison, page-based LRU is used to manage read cache for proposed CBM, BPLRU, FAB and BPAC. They have exactly the same pages available in read cache. Page padding is not enabled for BPLRU.

TABLE I. SPECIFICATION OF SSD CONFIGURATION

Page Read from Flash memory	25□s
Page Program (Write) to Flash memory	200□s
Block Erase	1.5ms
Serial Access to Register (Data bus)	100□s
Die Size	2 GB
Block Size	256 KB
Page Size	4 KB
Data Register	4 KB
Erase Cycles	100 K
SSD Capacity	320GB
NVM(STT-MRAM) write buffer read latency	32ns
NVM(STT-MRAM) write buffer write latency	40ns
DRAM read cache read latency	15ns

2) Workload Traces

We use a set of real-world traces to study the efficiency of different buffer schemes. Table II presents salient features of the workload traces. We employ a read-dominant I/O trace from an OLTP application running at a financial institution [17], henceforth referred to as Financial trace. We also employ a write-dominant I/O trace called CAMWEBDEV, which was collected by Microsoft made available by the SNIA[22]. Besides read and write dominant workloads, we intend to assess the different buffer schemes under mixed workloads. For this purpose, we use MSNFS and Exchange traces, which were collected by Microsoft made available by SNIA[22]. For all the traces, block address beyond SSD volume is ignored.

TABLE II. SPECIFICATION OF WORKLOADS

Workload	**Avg. Req. Size(KB)**	**Write (%)**	**Seq. (%)**	**Avg. Req. Inter-arrive Time(ms)**
Financial	3.89	18	0.6	11080.98
MSNFS	9.81	33	6.1	586.79
Exchange	12.01	72	10.5	3780.67
CAMWEBDEV	8.14	99	0.2	707.10

3) Evaluation Metrics

In this study, we utilize following metrics to characterize the behavior of different buffer management schemes.

- ***Average response time***, which is overall performance including read and write operations.
- ***Buffer hit ratio***, which is calculated as the ratio between the number of pages hit in the write buffer and total number of writing pages.
- ***Erase count***, which is number of blocks erased during garbage collection.
- ***Distribution of destage length***, which indicates sequentiality of blocks evicted from write buffer. Note this destage length is measured before *merge-on-flush*.

B. Experiment Results

Figures 9 through Figure 12 show the average response time, buffer hit ratio, erasure count and distribution of destage length for the four workloads when we vary write buffer size. To indicate the destage length distribution, we use CDF curves to show percentage (shown on Y-axis) of evicted pages from write buffer whose sizes are less than a certain value (shown on X-axis). Because of space limitation, we only present CDF curves for 1 MB write buffer in Figure 9 through Figure 12. The following observations are made from the results.

1) Financial trace

Figure 9 shows that CBM outperforms BPLRU, FAB and BPAC in terms of average response time, buffer hit, number of erases and number of sequential writes under the completely read dominant trace.

With 1MB write buffer, the average response time of CBM is 0.16 msecond. By contrast, the average response time of BPLRU, FAB and BPAC is 0.75, 0.51 and 0.28 msecond (see Fig. 9(a)). CBM makes 80%, 69% and 43% improvement in terms of average response time compared with BPLRU, FAB and BPAC. Accordingly, there is a 185%, 146% and 38% buffer hit improvement over BPLRU, FAB and BPAC (see Fig. 9(b)).

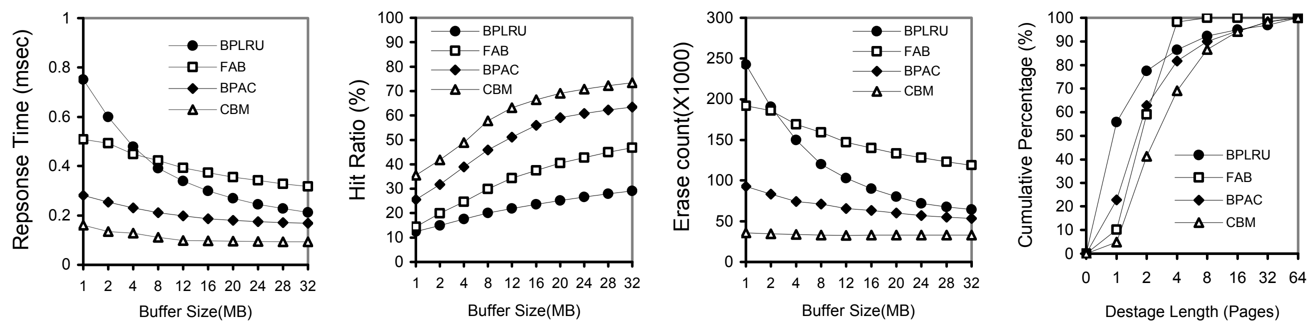
Figure 9(d) shows that the percentage of 1-page destages of BPLRU, FAB and BPAC is 56%, 10%, and 22% respectively. By contrast, CBM only has 5% small destages, better than BPLRU, FAB and BPAC. Further, CBM provides much more large destage than BPLRU, FAB and BPAC. For example, almost 32% of the destages are larger than 4 pages in size for CBM, while BPLRU, FAB and BPAC only have 14%, 2% and 18% destages larger than 4 pages. The result indicates that CBM algorithm is very efficient in increasing destage sequentiality of write buffer. Consequently CBM exhibits 85%, 82% and 45% less block erasures compared with BPLRU, FAB and BPAC for a 1 MB write buffer (see Fig. 9(c)).

The results indicate that the performance gain of CBM comes from two aspects: high buffer hit ratio and reduced garbage collection. This is because CBM exploits page and block to manage write buffer in hybrid form, taking both temporal and spatial localities into account. CBM makes its contributions through improving buffer hit ratio and increasing the portion of sequential writes.

2) MSNFS trace

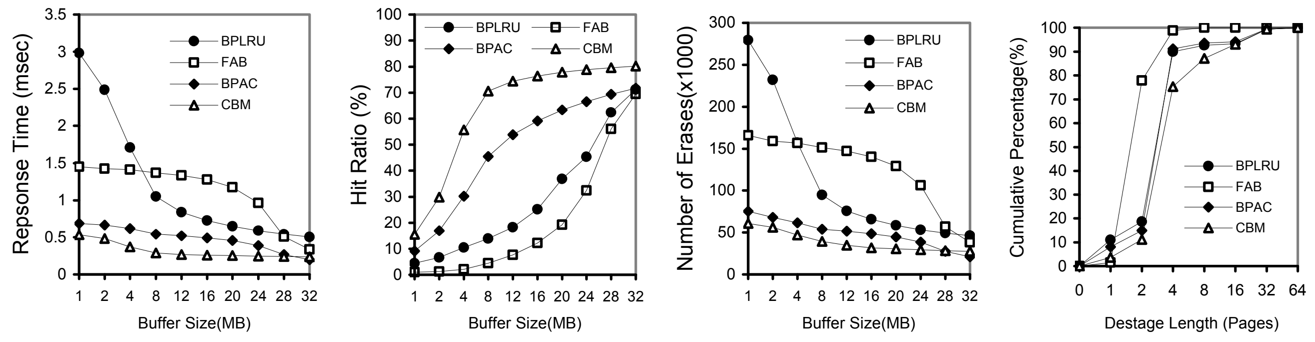
MSNFS trace is a random workload in which reads are about 34% more than writes.

Figure 10 shows that CBM exhibits up to 82% faster, 39% more buffer hits and 78% less block erases compared to BPLRU for the write buffer size up to 32 MB. CBM performs up to 63%, 63% and 308% better in terms of average response time, buffer hits ratio and erase count compared to FAB. CBM also achieves up to 28%, 66% and 20% better in terms of average response time, buffer hits ratio and erase count over BPAC. Beyond 32 MB, the advantages of CBM over BPLRU, FAB and BPAC narrow down because write buffer is large enough to accommodate most accesses. It is obvious to see that the advantage of CBM over BPLRU, FAB and BPAC is more significant for smaller write buffer configuration. This is because CBM efficiently makes use of write buffer space by improving both buffer hit ratio and destage sequentiality.



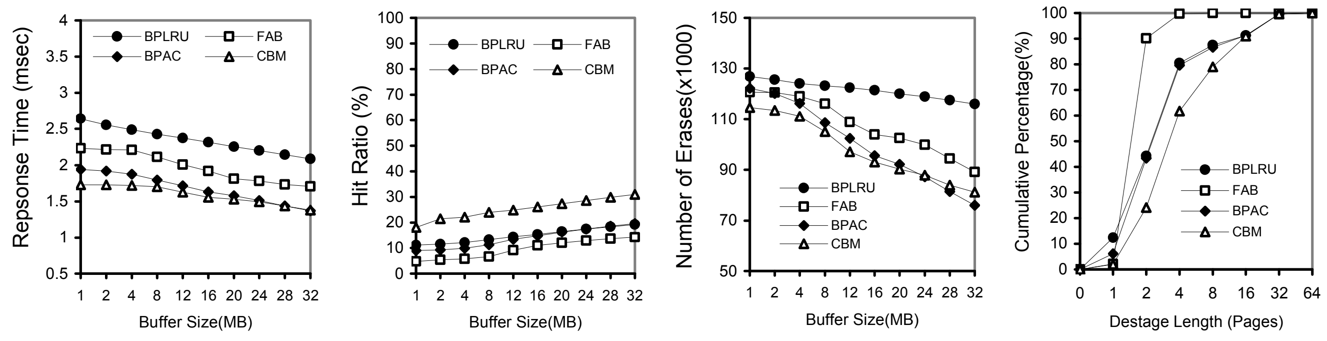
(a) Overall performance (b) Write buffer hit rate (c) Erase count (d) Destage Length

Fig. 9. Financial Trace



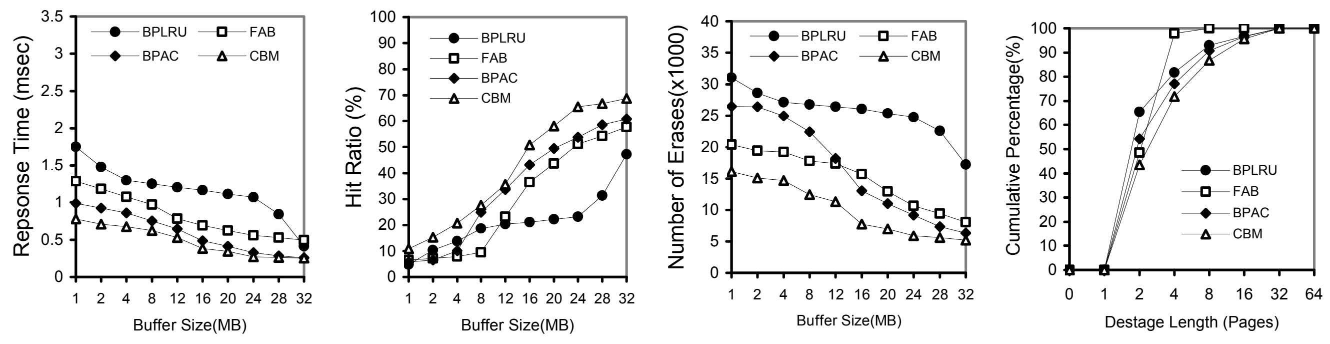
(a) Overall performance (b) Write buffer hit rate (c) Erase count (d) Destage Length

Fig. 10. MSNFS Trace



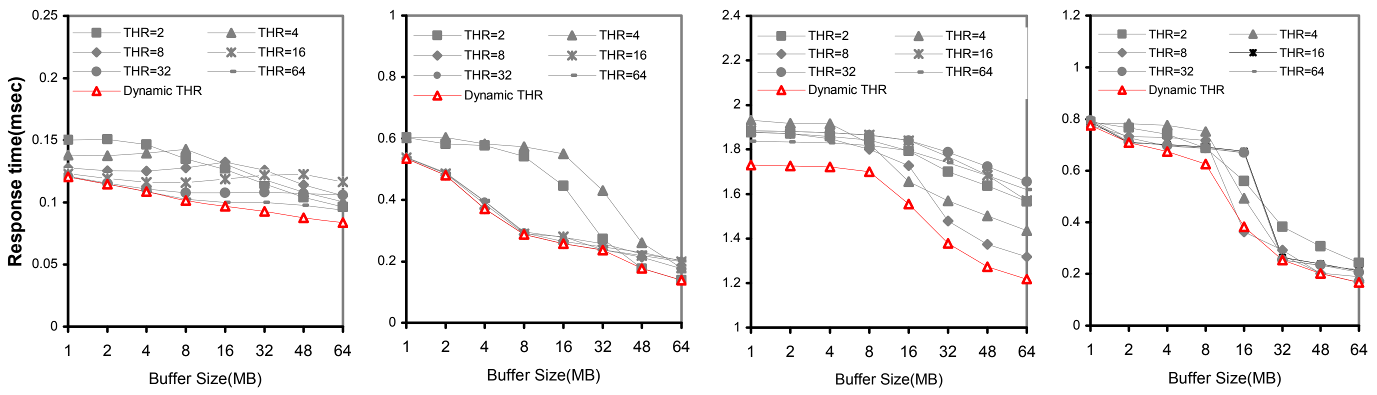
(a) Overall performance (b) Write buffer hit rate (c) Erase count (d) Destage Length

Fig. 11. Exchange Trace



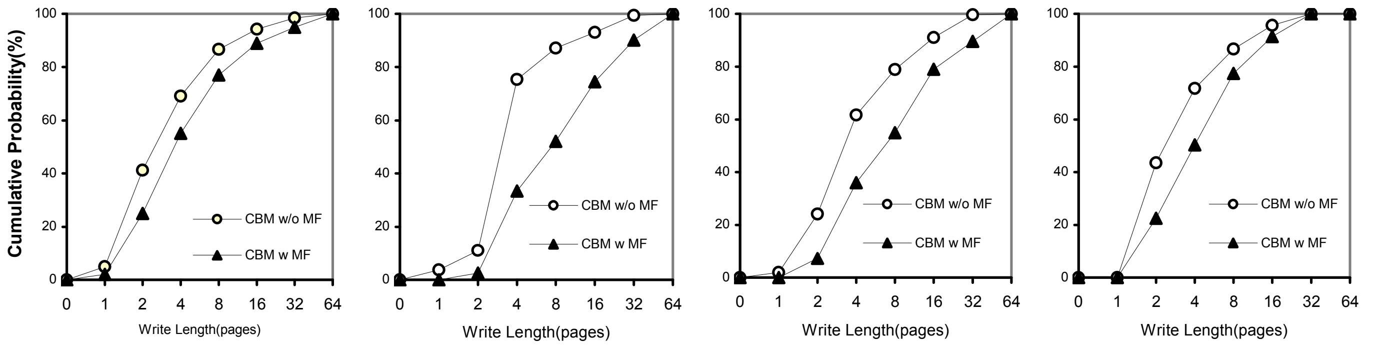
(a) Overall performance (b) Write buffer hit rate (c) Erase count (d) Destage Length

Fig. 12. CAMWEBDEV Trace



(a) Financial (b) MSNFS trace (c) Exchange (d) CAMWEBDEV

Fig. 13. Effect of migration threshold



(a) Financial (b) MSNFS trace (c) Exchange (d) CAMWEBDEV

Fig. 14. Effect of Merge-on-Flush (WF)

3) Exchange trace

Exchange trace is a random workload in which writes are about 44% more than reads.

For 1 MB write buffer, the average response time of CBM is 1.73 msecond. By contrast, the average response time of BPLRU is 2.64 msecond (see Fig. 11(a)). CBM makes 34% improvement in terms of average response time compared to BPLRU. Accordingly, there is a 62% buffer hit increase (see Fig. 11(b)) and a 10% erase reduction (see Fig. 11(c)). CBM exhibits 23% faster, 280% more buffer hits and 5% less block erasures than FAB for 1 MB write buffer. CBM also performs better than BPAC especially when write buffer is small.

The percentage of small writes (less than 2 pages) of BPLRU and FAB is 44% and 90%, respectively. By contrast, CBM has 24% small writes, better than BPLRU and FAB (see in Fig. 11(d)). Further, CBM provides much more sequential destages than BPLRU and FAB. For example, almost 38% of the destages are larger than 4 pages in size for CBM, while BPLRU and FAB only have 20% and 0.2% destages larger than 4 pages.

4) CAMWEBDEV trace

CAMWEBDEV trace is a completely random write dominant workload.

Figure 12 shows that CBM performs 56%, 40%, and 21% faster than BPLRU, FAB and BPAC for 1MB write buffer. Accordingly, there is a 128%, 65% and 98% hit ratio improvement compared with BPLRU, FAB and BPAC. There is also a 48%, 21% and 38% reduction of block erasures over BPLRU, FAB and BPAC.

CBM is also efficient in reducing the number of small writes and increasing the number of sequential writes for write intensive workload. Figure 12 (d) shows that BPLRU and FAB produce 82% and 98% small destages (less than 4 pages). By contrast, CBM has 74% small destage, which is less than BPLRU and FAB. CBM also provide 10% sequential destages, which is larger than 8 pages in size. However, BPLRU and FAB only have 7% and 0.02% large destages.

The results further show that the destage sequentiality is directly correlated to the garbage collection overhead and performance. With 1 MB write buffer, CBM is able to produce 27% destages whose sizes are larger than 4 pages, while BPLRU and FAB generates 18% and 2% destages larger than 4 pages (see Figure 12(d)). Accordingly, CBM makes 48% and 21% garbage collection overhead reduction over BPLRU and FAB (see Figure 12(c)). Consequently, performance is improved by 56% and 40% compared to BPLRU and FAB (see Figure 12(a)). The correlation clearly indicates that destage length is a critical factor affecting SSD performance and garbage collection overhead.

5) Effect of Threshold

To investigate how threshold value affects the efficiency of proposed CBM, we tested CBM with static thresholds and dynamic threshold for different traces, as shown in Figure 13.

Let's take Figure 13(c) as an example. With 16MB write buffer, the average response time of CBM is 1.79msecond, 1.65msecond and 1.73msecond when threshold value is 2, 4, and 8 respectively. By contrast, the average response time of CBM is 1.55msecond for dynamic threshold, which is much better than that of static thresholds.

We further observed that the same threshold is also unable to achieve optimal performance for different workloads. Figure 13(a) shows that with 16MB write buffer, CBM performs better when threshold is set as 64 for Financial trace, compared to other static thresholds. However, with buffer size of 16 MB and threshold of 64, the average response time of CBM is 0.67msecond for CAMWEBDEV trace, which is worse compared to threshold of 2, 4, and 8 respectively(see Figure 13(d)). By contrast, the results in Figure 13 show that dynamic threshold achieves best performance for Financial, MSNFS, Exchange and CAMWEBDEV traces respectively.

The variation in performance curves shown in the Figure 13 clearly indicates that threshold value has significant impact on efficiency of our proposed CBM. Statically setting threshold is unable to achieve optimal performance. Dynamically adjusting the threshold for enterprise workloads makes proposed CBM workload adaptive.

6) Effect of Merge-on-Flush

Figures 9 through Figure 12 show destage length of write buffer before *merge-on-flush*. Proposed CBM uses *merge-on-flush* to increase possibility of forming full block writes. Dirty blocks evicted from write buffer, together with clean pages in read cache are merged and written to flash memory. To indicate the effect of the *merge-on-flush,* we plot the distribution of write length in Figure 14. Please note that we use destage length in Figure 9-12 to denote the length of evicted block before *merge-on-flush*, and write length to denote the length of block passed to FTL or flash memory. Figure 14 shows that although CBM is efficient in increasing destage sequentiality for write buffer, *merge-on-flush* is able to further increase write sequentiality of blocks written to FLT or flash memory. For read dominated workload Finance and write dominated workload CAMWEBDEV, the effect of *merge-on-flush* is not obvious (see Fig. 14(a), Fig.14(d)). But for mixed workloads MSNFS and Exchange, *merge-on-flush* can further improve write sequentiality by up to 56% (see Fig. 14(b), Fig.14(c)).

V. Real Implementation Evaluation

A. Implementation Setup

We use OpenSSD as platform to implement and evaluate our CBM (See Fig. 15). The OpenSSD is the most up-to-date open platform available today for prototyping and evaluating new SSD designs [36]. It is the Indilinx's reference implementation of SSD based on the Barefoot controller, which is an ARM-based SATA controller used in numerous commercial SSDs. The controller runs firmware that can conduct read/write/erase operations to the flash banks over a 16-bit I/O channel.

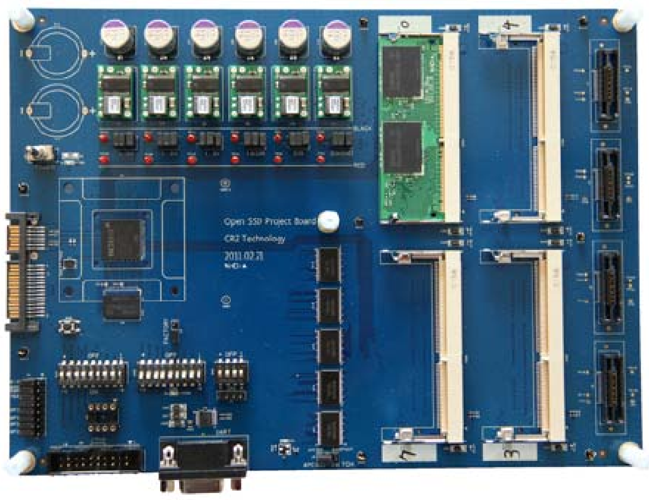


Fig. 15. OpenSSD Platform

TABLE III. OpenSSD configuration

Barefoot SSD Controller	ARM7TDMI-S, 175MHZ	Package	4
Internal SRAM	96KB	Host Interface	SATA 2.0
Mobile DRAM	64MB	Dies/package	2
NAND Flash	64GB-256GB	Banks/package	4
NAND Flash Type	35nm, 64Mbit MLC	Ways	2
Physical Page	8KB	Physical Block	1MB
Virtual Page	32KB	Virtual Block	4MB

The OpenSSD evaluation board is composed of 96KB internal SRAM, 64MB SDRAM for storing the FTL mapping table and for SATA buffers, and 8 slots holding up to 256GBof MLC NAND flash. It also supports standard SATA interfaces. The chips use two planes and have 8KB physical pages. The device uses large 32KB virtual pages, which improve performance by striping data across physical pages on 2 planes on 2 chips within a flash bank. Erase blocks are 4MB and composed of 128 contiguous virtual pages (See Table III).

The OpenSSD platform comes with open-source firmware libraries for accessing the hardware and five sample FTLs: Tutorial, Dummy, Greedy, DAC and FASTer (successor of FAST) [35]. Hybrid FTL FASTer is used in our prototype experiment. We use a system with 2.4GHZ CPU, 2GB DRAM and a 1TB SATA hard disk plus the OpenSSD board configured with 2 flash modules (64GB total). For FASTer FTL, 64GB flash memory is divided as 15240 data blocks, 1120 log blocks and 272 isolation blocks. We use 24MB DRAM segment as read cache and 32MB DRAM segment as write buffer. The remainder 8MB is used for FTL mapping table. As a real storage device, it allows us to test buffer algorithms with existing I/O benchmarks and real application workloads.

B. *Iometer Testing*

We use widely used benchmark tool Iometer to generate random mixed I/O with 50% reads and 50% writes. Request size is varied from 4KB to 128KB. Single worker is selected to generate workloads. Figure 16 shows the performance of different buffer algorithms on OpenSSD platform. From the results we can see that proposed CBM outperforms BPLRU, FAB and hybrid BPAC across different request size. CBM is up to 2X, 1.6X and 30% faster than FAB, BPLRU and BPAC. Let's take 4K requests as example. CBM achieves 2890 IOPS, while FAB, BPLRU and BPAC only perform 1450 IOPS, 1840 IOPS and 2220 IOPS respectively (See Fig. 16). Real implementation and evaluation on the OpenSSD further validate that proposed CBM is efficient and superior to existing buffer schemes.

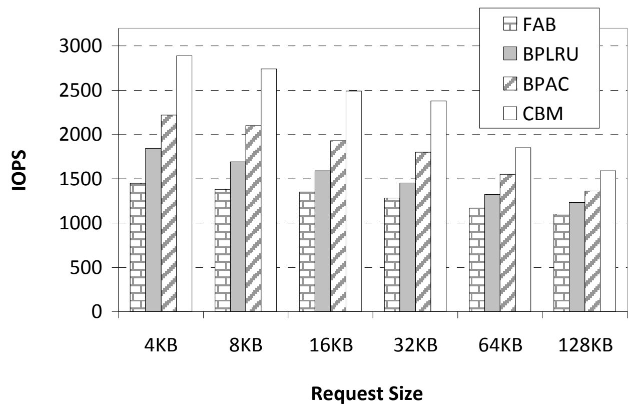


Fig. 16. Performance comparison of different buffer algorithms on OpenSSD.

VI. CONCLUSION

In this paper, we propose a cooperative buffer management scheme referred to as CBM, which coordinates write buffer and read cache to fully exploit temporal and spatial localities among I/O intensive workload. To improve both buffer hit rate and destage sequentiality, CBM divides write buffer into *Page Region* and *Block Region*. Randomly written data is put in the *Page Region* at page granularity, while sequentially written data is stored in the *Block Region* at block granularity. Leveraging threshold-based migration, CBM dynamically classifies random write from sequential writes. When a block is evicted from write buffer, CBM merges the dirty pages in write buffer and the clean pages in read cache belonging to the evicted block to maximize the possibility of forming full block write. We have implemented CBM on real OpenSSD. Testing results demonstrate that proposed CBM is efficient in improving performance, reducing block erasure with high buffer utilization.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their feedback and suggestions for improvement. This work was supported by Agency for Science, Technology and Research (A*STAR), Singapore under Grant No. 112-172-0010.

REFERENCES

[1] MINA, C., KIMB, K., CHOC, H., LEED, S., AND EOM, Y. I. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceeding of 10th USENIX Conference on File and Storage Technologies (FAST'12)*, CA, USA.

[2] LIU, R., YANG, C., AND WU, W. 2012. Optimizing NAND Flash-Based SSDs via Retention Relaxation. In *Proceeding of 10th USENIX Conference on File and Storage Technologies (FAST'12)*, USENIX, San Jose, CA.

[3] REN, J. AND YANG, Q. 2011. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceeding of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, San Antonio, Texas, 278-289.

[4] BADAM, A. AND PAI, V. S. 2011. SSDAlloc: hybrid SSD/RAM memory management made easy. In *Proceedings of the 10th USENIX Symposium on Network Systems Design and Implementation(NSDI'11),* USENIX, Lombard, IL.

[5] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. 2012. The Bleak Future of NAND Flash Memory. In *Proceeding of 10th USENIX Conference on File and Storage Technologies (FAST'12)*, USENIX, San Jose, CA.

[6] PAN, Y., DONG, G., AND ZHANG, T. 2011. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *Proceeding of 9th USENIX Conference on File and Storage Technologies (FAST'11)*, USENIX, San Jose, CA, 245-258.

[7] Jianhui Yue, Yifeng Zhu, Accelerating Write by Exploiting PCM Asymmetries, In Proceeding of HPCA'13, 2013, China.

[8] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. IEEE Design & Test of Computers, 28(1):52–63, 2011.

[9] Hyojun Kim and Umakishore Ramachandran, FlashLite: a user-level library to enhance durability of SSD for P2P File Sharing, *In Proc. ICDCS'09*.

[10] S. Lee, D. Shin, Y. Kim, and J. Kim, LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems, In *Proc. SPEED'08*, Feburary 2008.

[11] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis,M. Manasse, and R. Panigrahy, Design tradeoffs for SSD performance, *In Proc. of USENIX'08*, June 2008.

[12] F. Chen, D. A. Koufaty and X.D.Zhang, Understanding intrinsic characteristics and system implications of flash memory based solid state drives, *In Proc. of SIGMETRICS'09*, 2009.

[13] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis, Block Management in Solid-State Devices, *In Proc. of* USENIX'09, 2009.

[14] H. Kim and S. Ahn, BPLRU: A buffer management scheme for improving random writes in flash storage, *In Proc. of FAST'08*, 2008.

[15] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang, DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality, *In Proc. of FAST'05*, 2005.

[16] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, The DiskSim Simulation Environment Version 4.0

Reference Manual, *Technical Report, CMU-PDL-08-101*, Carnegie Mellon University, May,2008.

[17] OLTP Trace from UMass Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage

[18] N. Megiddo and D. Modha, ARC: A Self-tuning, Low Overhead Replacement cache, *In Proc. of FAST'03*, 2003.

[19] T. Johnson and D. Shasha, 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *In Proc. of VLDB '94*, 1994.

[20] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, FAB: Flash-aware Buffer Management Policy for Portable Media Players, In *IEEE Transactions on Consumer Electronics,* vol. 22, no. 2, 2006.

[21] Biplob Debnath, Sunil Subramanya, David Du and David J. Lilja, Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks, *In Proc. of MASCOTS'09*, 2009.

[22] Block traces from SNIA, http://iotta.snia.org/traces/list/BlockIO

[23] F. Corbato, A Paging Experiment with the Multics System, In *MIT Project MAC Report MAC-M-384*, 1968.

[24] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, CFLRU: A Replacement Algorithm for Flash Memory, *In Proc. of CASES'06*, 2006.

[25] Raghu Ramakrishnan and Johannes Gehrke, Database Management Systems 3rd Edition.

[26] Guanying Wu; Eckart, B.; Xubin He, BPAC: An Adaptive Write Buffer Management Scheme for Flash-based Solid State Drives, *In proceeding of IEEE 26th Symposium on Mass Storage Systems and Technologies* (MSST 2010), May, 2010.

[27] Jian Hu, Hong Jiang, Lei Tian, Lei Xu, PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD, *In Proceeding of 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems* (MASCOTS 2010), Aug. 2010.

[28] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber, Extending SSD Lifetimes with Disk-Based Write Caches, *In Proc. FAST'10.*

[29] B. Gill and D. Modha, WOW: Wise Ordering for Writes – Combining Spatial and Temporal Locality in Non-Volatile Caches, *In Proc. of FAST'05*, 2005.

[30] L.P. Chang, On efficient wear leveling for large-scale flash memory storage systems, *In Proceedings of the 2007 ACM Symposium on Applied Computing*. USA, 2007.

[31] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, A space-efficient flash translation layer for compact flash systems, *In IEEE Transactions on Consumer Electronics*, volume 48(2):366-375, 2002.

[32] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, A Log Buffer based Flash Translation Layer Using Fully Associative Sector Translation, *IEEE Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[33] Dean Hildebrand, Lee Ward and Peter Honeyman, Large files, small writes, and pNFS, *In Proc. of ICS'06*, 2006.

[34] R. Cheveresan, M. Ramsay, C. Feucht, etc., Characteristics of workloads used in high performance and technical computing, *In Proc. of ICS'07*, 2007.

[35] A. Gupta, Y. Kim, and B. Urgaonkar, DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings, *In Proc. Of ASPLOS'09*, 2009.

[36] LIM, S.P., LEE, S. W., MOON, L. B. 2010. FASTer FTL for Enterprise-Class Flash Memory SSDs. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os(SNAPI 2010)*, USA.

[37] THE OPENSSD PROJECT. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[38] RRAM Technology Overview. http://www.crossbar-inc.com/technology/resistive-ram-overview.html.