HBM: A HYBRID BUFFER MANAGEMENT SCHEME FOR SOLID STATE DISKS

GONG BOZHAO

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

June 2010

Acknowledgement

Fist of all, I want to thank my parents for their love and encouragement when I felt depressed during this period.

I would like to express my deep-felt gratitude to my supervisor, Prof. Tay Yong Chiang, for his guidance and patience. He always gave me valuable suggestions when I did not know how I should do research. He also cared about my life and offered his help for job opportunities. I also wish to thank Dr. Wei Qingsong for his help on this thesis. It was his research work that inspired me. The comments from Assoc. Prof. Weng Fai WONG and Assoc. Prof. Tulika Mitra for my Graduate Research Paper are greatly appreciated.

I like to thank a lot of my friends around me, Wang Tao, Suraj Pathak, Shen Zhong, Sun Yang, Lin Yong, Wang Pidong, Lun Wei, Gao Yue, Chen Chaohai, Wang Guoping, Wang Zhengkui, Zhao Feng, Shi Lei, Lu Xuesong, Hu Junfeng, Zhou Jingbo, Li Lu, Kang Wei, Zhang Xiaolong, Zheng Le, Lin Yuting, Zhang Wei, Deng Fanbo, Ding Huping, Hao Jia, Chen Qi, Ma He, Zhang Meihui, Lu Meiyu, Liu Linlin, Cui Xiang, Tan Rui, Chen Kejie, for sharing wonderful time with me.

Special thanks to friends currently in China, Europe and US. They were never ceasing in caring about me.

Gong Bozhao

Contents

Summary

Random writes significantly limit the application of flash memory in enterprise environment due to its poor latency and high garbage collection overhead. Several buffer management schemes for flash memory have been proposed to overcome this issue, which operate either at page or block granularity. Traditional page-based buffer management schemes leverage temporal locality to pursue buffer hit ratio improvement without considering sequentiality of flushed data. Current block-based buffer management schemes exploit spatial locality to improve sequentiality of write accesses passed to flash memory at a cost of low buffer utilization. None of them achieves both high buffer hit ratio and good sequentiality at the same time, which are two critical factors determining the efficiency of buffer management for flash memory. In this thesis, we propose a novel hybrid buffer management scheme referred to as HBM, which divides buffer space into page region and block region to make full use of both temporal and spatial localities among accesses in hybrid form. HBM dynamically balances our two objectives of high buffer hit ratio and good sequentiality for different workloads. HBM can make more sequential accesses passed to flash memory and efficiently improve the performance.

We have extensively evaluated HBM under various enterprise workloads. Our benchmark results conclusively demonstrate that HBM can achieve up to 84% performance improvement and 85% garbage collection overhead reduction compared to existing buffer management schemes. Meanwhile, the energy consumption of flash chips for HBM is limited.

List of Tables

List of Figures

Chapter 1

Introduction

Flash memory has shown its obvious merits especially in the storage space compared to the traditional hard disk drive (HDD), such as small size, quick access, saving energy [14]. It is originally used as primary storage in the portable devices, for example, MP3, digital camera. As its capacity is increasing and its price is dropping, replacing HDD over the personal computer storage and even server storage with flash memory in the form of Solid State Drive (SSD) has been paid more attention. Actually, Samsung[1] and Toshiba[2] have launched the laptops with only SSDs. Google[3] considers replacing parts of its storage with Intel[4] SSD storage in order to save energy [10], and MySpace[5] has made use of the Fusion-IO[6] ioDrives Duo as its primary storage servers instead of hard disk drives, and this switch brought it large energy consumption [29].

[1]www.samsung.com
[2]www.toshiba.com
[3]www.google.com
[4]www.intel.com
[5]www.myspace.com
[6]www.fusionio.com

1.1 Motivation

Although SSD shows its attractive worthiness especially on improving the random read performance due to no mechanical characteristic, however, it could suffer from random write[7] issue especially when it is applied in the enterprise environment [33].

Just like HDD, SSD can make use of RAM inside as the buffer to improve performance [22]. The buffer can delay the 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. Different from HDD, buffer inside SSD can be managed not only at page granularity but also at block granularity[8]. In other words, the basic unit in the buffer could be a logical block equal to the physical block size in flash memories. Block is larger than page in flash memory, which usually consists of 64 or 128 pages. The internal structure of flash memory will be introduced in section 2.1. Existing buffer management algorithms try to exploit the temporal locality or spatial locality in the access patterns in order to get high buffer hit ratio or good sequentiality of flushed data, which are two critical factors determining the efficiency of buffer management inside SSD.

However, these two targets could not be achieved simultaneously under the existing buffer management algorithms. Therefore, we are motivated to design a novel hybrid buffer management algorithm which manages data both at page granularity and block granularity, in order to fully utilize both temporal and sequential localities to achieve high buffer hit ratio and good sequentiality for SSD.

[7]In this thesis, random request means small-to-moderate sized random request if not specified.

[8]Another expression of page granularity or block granularity is page-level or block-level, page-based or block-based

To illustrate the limitation of current buffer management schemes and our motivation to design a hybrid buffer management, a reference pattern including sequential and random accesses is shown in the Table 1.1.

Table 1.1: Comparison of page-level LRU, block-level LRU and hybrid LRU. Buffer size is 8 pages and an erase block contains 4 pages. Hybrid LRU maintains buffer at page and block granularity, and only full blocks will be managed at block granularity and will be selected as victim. In this example, we use [] to denote block boundary.

Access	Page-Level LRU			Block-Level LRU			Hybrid LRU		
	Buffer(8)	Flush	Hit?	Buffer(8)	Flush	Hit?	Buffer(8)	Flush	Hit?
0,1,2,3	3,2,1,0		Miss	[0,1,2,3]		Miss	[0,1,2,3]		Miss
5,9,11,14	14,11,9,5,3,2,1,0		Miss	[14],[9,11],[5],[0,1,2,3]		Miss	14,11,9,5,[0,1,2,3]		Miss
7	7,14,11,9,5,3,2,1	0	Miss	[5,7],[14],[9,11]	[0,1,2,3]	Miss	7,14,11,9,5	[0,1,2,3]	Miss
3	3,7,14,11,9,5,2,1		Hit	[3],[5,7],[14],[9,11]		Miss	3,7,14,11,9,5		Miss
11	11,3,7,14,9,5,2,1		Hit	[9,11],[3],[5,7],[14]		Hit	11,3,7,14,9,5		Hit
2	2,11,3,7,14,9,5,1		Hit	[2,3],[9,11],[5,7],[14]		Miss	2,11,3,7,14,9,5		Miss
14	14,2,11,3,7,9,5,1		Hit	[14],[2,3],[9,11],[5,7]		Hit	14,2,11,3,7,9,5		Hit
1	1,14,2,11,3,7,9,5		Hit	[1,2,3],[14],[9,11],[5,7]		Miss	1,14,2,11,3,7,9,5		Miss
10	10,1,14,2,11,3,7,9	5	Miss	[9,10,11],[1,2,3],[14]	[5,7]	Miss	10,1,14,2,11,3,7,9	5	Miss
7	7,10,1,14,2,11,3,9		Hit	[7],[9,10,11],[1,2,3],[14]		Miss	7,10,1,14,2,11,3,9		Hit
	Sequential flush	0			1			1	
	Buffer hit		6			2			3

In this example, page-level LRU achieves 6 hits higher than block-level LRU, and block-level LRU has 1 sequential flush better than page-level LRU. Hybrid LRU achieves 3 buffer hits and 1 sequential flush, which combines the advantages of both page-level LRU and block-level LRU.

1.2 Contribution

In order to research on the device-level buffer management[9] for SSD using FlashSim [25] SSD simulator designed by the Pennsylvania State University, some implementation work has been done first. Firstly, we add BAST [24] FTL scheme into FlashSim, because some existing buffer management algorithms are based on this basic log-block FTL [24] scheme. Then we integrate a buffer module above FTL level and implement four buffer management algorithms for SSD, which are BPLRU [22], FAB [18], LB-CLOCK [12], and HBM.

We propose a hybrid buffer management scheme referred to as HBM, which

[9]It means the buffer is inside SSD.

gives consideration to buffer hit ratio and sequentiality by exploiting both temporal and spatial localities among access patterns. Based on this hybrid scheme, the whole buffer space is divided into two regions: page region and block region. These two regions are managed in different ways. Specifically, in the page region, data is managed and adjusted in logical page granularity to improve buffer space utilization, while logical block is the basic unit in the block region. Page region prefers the random small sized access pages, while sequential access pages in the block region are replaced first when new incoming data cannot be hold any more. Data can not only be moved inside page region or block region, but also dynamically migrated from page region to block region when the number of pages in the same logical block reaches a threshold that is adaptive to different workloads. According to hybrid management and dynamic migration, HBM improves the performance of SSD by significantly reducing the internal fragmentation and garbage collection overhead associated with random write, meanwhile, the energy consumption of flash chips for HBM is limited.

1.3 Organization

The remainder of this thesis is organized as follows: Chapter 2 gives an overview of background knowledge of flash memory and SSD, and surveys some existing well known buffer management algorithms inside SSD. Chapter 3 presents details of hybrid buffer management scheme. Evaluation and experiment results are presented in Chapter 4. In Chapter 5, we conclude this thesis and possible future work is summarized.

Chapter 2

Background and Related Work

In this chapter, basic background knowledge of flash memory and SSD is introduced first. The issue of random writes for SSD is subsequently explained. Then we mainly present three existing buffer management algorithms for SSD. After each buffer management, the work will be summarized in brief. Specially, in the end of this chapter, we introduce a similar research framework with ours: BPAC, however, from which our research work have different internal techniques.

2.1 Flash Memory Technology

Two types of flash memories[1], NOR and NAND [36], are existing. In this thesis, flash memory refers to NAND specifically, which is much like block devices accessed in units of sectors, because it is the common data storage material regarding to flash memory based SSDs on the market.

Figure 2.1 shows the internal structure of a flash memory chip, which consists dies sharing a serial I/O bus. Different operations can be executed in different dies. Each die contains one or more planes, which contains blocks (typically 2048 blocks) and page-sized registers for buffering I/O. Each block includes

[1]We also use term ”flash chips” or ”flash memory chips” as alternative expression of flash memory.

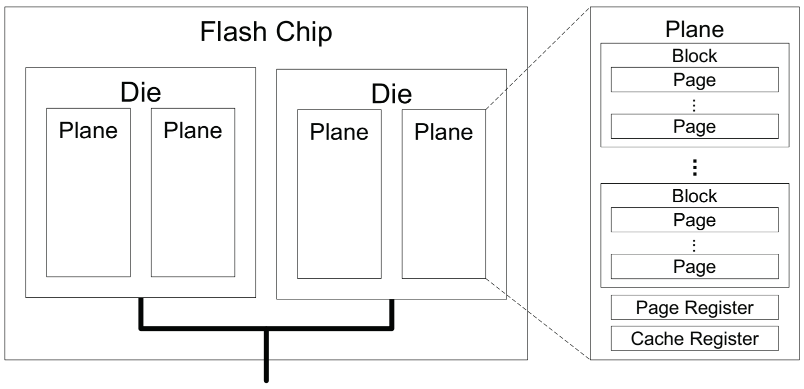


Figure 2.1: Flash memory chip organization. Figure adapted from [35]

pages, which has data and mete data area. The typical size of data area is 2KB or 4KB, and meta data area (typically 128 bytes) is used to store identification or correction information and page state: *valid*, *invalid* or *free*. Initially, all the pages are in free state. When a write operation happens on a page, the state of this page is changed to valid. For updating this page, mark this page invalid first, then write data into a new free page. This is called *out-of-place update* [16]. In order to change the invalid state of a page into free again, the whole block that contains the page should be erased first.

Three operations are allowed for NAND: *Read*, *Write* and *Erase*. As for reading a page, the related page is transferred into the page register then I/O bus. The cache register is especially useful for reading sequential pages within a block, specifically, pipelining the reading stream by page register and cache register can improve the read performance. Read operation costs least in the flash memory. To write a page, the data is transferred from I/O bus into page register first, similar to the read operation, for sequentially writing, the cache register can be used. A write operation can only change bit values from 1 to 0 in the flash chips. Erasing is the only way to change bit values back to 1. Unlike read and write both of which can be performed at the page level, the block accessed unit is for erasing. After erasing a block, all bit values for all pages within a block are set to 1. So erase operation cost most in the flash memory. In addition, for each block, erase count that can be endured before it is worn out is finite, typically around 100000.

2.2 Solid State Drive

SSD is constructed from flash memories. It provides the same physical host interface as HDD to allow operating systems to access SSD in the same way as conventional HDD. In order to do that, an important firmware called Flash Translation Layer (FTL) [4] is implemented in the SSD controller. Three important functions provided by FTL are *address mapping*, *garbage collection* and *wear leveling*.

Address Mapping **-** FTL maintains the mapping information between logical page and physical page [4]. When it processes the write operation, it writes the new page to a suitable empty page if the requested place has already been accessed before. Meanwhile, it marks the valid data in the requested place invalid. Depending on the granularity of address mapping, FTL can be classified into three groups: page-level, block-level and hybrid-level FTL [9]. In the page-level FTL, each logical page number (LPN) is mapped to each physical page number (PPN) in flash memory. However this efficient FTL requires much RAM inside SSD in order to store the mapping table. Block-level FTL associates logical blocks with physical blocks, in which the mapping table is less. However, the mechanism that requires the same page offsets between the logical block and the corresponding physical block makes it not efficient because updating one page could lead to update the whole block. Hybrid-level FTL [2] combines page mapping with block mapping. It reserves a small amount of blocks called log blocks in which page-level mapping is used to buffer the small size write requests. Other than log blocks, the rest blocks called data blocks in which block-level mapping is used to hold ordinary data. The data block holds old data after write requests, the new data will be written in the corresponding

[2]It is also called Log-scheme FTL

log block. Hybrid-level FTL shows less garbage collection overhead and the required size of mapping table is less than page-level FTL. However, it incurs expensive full merges for random write dominant workloads.

***Garbage Collection* -** when free blocks are used up or a pre-defined threshold, garbage collection module is triggered to produce more free blocks by recycling invalidated pages. Regarding page-level mapping, it should first copy the valid pages out of the victim block and then write them into some new block. For block-level and hybrid-level mappings, it should merge the valid pages together with the updated pages whose logical page number is the same as them. During merge operation, due to copying valid pages of the data block and log block (under hybrid-level mapping), extra read and write operations must be invoked besides the necessary erase operations. Therefore, merge operations cost most during garbage collection [21].

There are three kinds of merge operations: *switch merge*, *partial merge* and *full merge* [16]. Considering the hybrid-level mapping, switch merge usually happens when the page sequence of log block is the same as that of data block. Log block will become the new data block because of all the new pages within it, while data block which contains all the old pages will be just erased without extra read or write operations. So switch merge cost less among merge operations. Partial merge happens when log blocks still can become new data block. In other words, all the valid pages in the data block can be copied to the log block first, then the data block is erased. Compared to partial merge, full merge happens in the condition that some valid page in the data block can not be copied to the log block and only a new allocated data block can hold it. During full merge, not only valid pages in the data block should be copied to the new allocated data block, but also the ones in the log block, after that, the old data block and log block are erased. So full merge cost most among merge operations.

On the basis of the cost of merge operations, an efficient garbage collection should make good use of switch merge operations and avoid full merge operations. Sequential writes which update sequentially can create opportunities of switch merge operations, and small sized random writes often go with expensive full merges. This is the reason why SSD suffers from random writes.

Wear leveling - some blocks are often written because of the locality in most workloads. So there exists wear out problem for some blocks due to frequently erasure compared to other blocks. FTL takes the responsibility for ensuring that even use is made of all the blocks by some wear leveling algorithm [7].

There are many kinds of FTLs proposed in academia, such as BAST, FAST [27], LAST [26], Superblock-based FTL [20], DFTL [16] and NFTL [28] and so on. Of these schemes, BAST and FAST are two representative ones. The biggest difference between BAST and FAST is that BAST has one to one correspondence between log block and data block, while FAST has many to many correspondence. However, in this thesis, BAST is used as the default FTL because almost every existing buffer management algorithm in SSD is based on BAST FTL [21].

2.3 Issues of Random Write for SSD

Firstly, according to out-of-place update for flash memory (see section 2.1), *internal fragmentation* [8] could be seen sooner or later if small size and random writes are distributed in much range of logical address space. It could result in some invalid page existing in almost all physical blocks. In that case, the prefetching mechanism inside SSD could not be effective because pages which are logically contiguous are probably physically distributed. This causes the bandwidth of sequential read to drop closely to the bandwidth of random read.

Secondly, the performance of sequential writes could be optimized over striping or interleaving mechanism [5][31]inside SSD, which is not effective for random writes. If a write is sequential, the data can be striped and written across different parallel units. Moreover, multi-page read or write can be efficiently interleaved over pipeline mechanism [13], while multiple single-page reads or writes can not be conducted in this way.

Thirdly, more random writes can incur higher overhead of garbage collection, which is usually triggered to produce more free blocks when the number of free blocks gets lower than a pre-defined threshold. During garbage collection, sequential writes can incur lower-cost switch merge operations, and random writes can incur much higher-cost full merge operations which are usually accompanied by extra reads or writes. In addition, these internal operations running in the background may compete for resources with incoming foreground requests [8] and therefore increase latency.

Finally, increased erase operations due to random writes could incur more erase operations and shorten the lifetime of the SSD. Experiments in [23] show that random write intensive workload could make flash memory wear out over hundred times faster than sequential write intensive workload.

2.4 Buffer Management Algorithms for SSD

Many existing disk based buffer management algorithms are based on page level, such as LRU, CLOCK [11], 2Q [19] and ARC [30]. These algorithms try to increase buffer hit ratio as much as possible. Specifically, they only focus on utilizing temporal locality to predict the next pages to be accessed and minimize page fault rate [17]. However, directly applying these algorithms is not enough for SSD because spatial locality is not catered for, and the sequential requests may be broken up into small segments so that the overhead of flash memories may increase when replacement happens.

In order to exploit spatial locality and provide more sequential writes for flash memories in SSD, buffer algorithms based on block level are proposed, such as FAB, BPLRU and LB-CLOCK. According to these algorithms, accessing a logical page results in adjusting all the pages in the same logical block based on the assumption that all pages in this block have the same recency. In the end of this section, a similar algorithm with our work called BPAC [37] will be introduced in brief. However, we have several different internal designs and implementations. Because BPAC is introduced by a short research paper which shows not much information about its details, moreover, BPAC and our work has been done independently at the same time, so here we just briefly describe some similarities and differences.

2.4.1 Flash Aware Buffer Policy

The flash aware buffer (FAB) [18] is a block-level buffer management algorithm for flash storage. Similar to LRU, it also maintains a LRU list in its data structure. However, the node in the list is not a page unit, but a block unit, meaning that pages belonging to the same logical block of flash memory are in the same node. When a page is accessed, the whole logical block which belongs to is moved to the head of the list which is the most recent accessed end. If a new page is added to the buffer, it is also inserted into the most recent used end of the list. Moreover, due to block-level algorithm, FAB flushes the whole victim block, not a single victim page. The logical view of FAB is shown in figure 2.2.

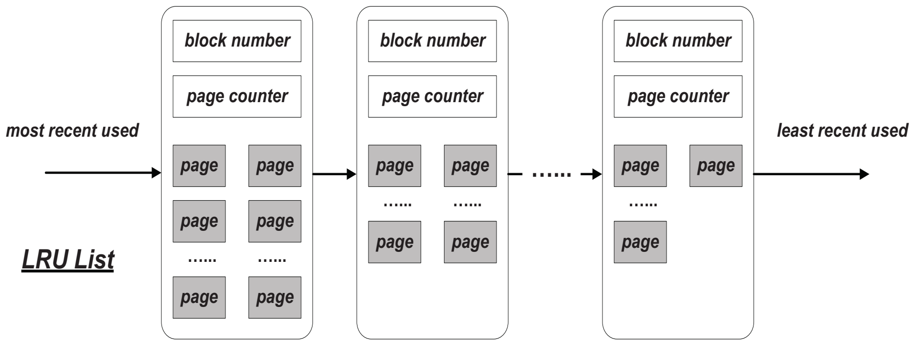


Figure 2.2: The main data structure of FAB

In the block node, the page counter means the number of pages which belong to the block. In FAB, a block whose has the largest page counter is always to be selected to be flushed. If there is not only one candidate victim block, it will choose the least recently used one.

In some cases, FAB decreases the number of extra operations in the flash memory, because it flushes valid pages in the buffer as often as possible, and it may decrease copy valid page operations when erasing a block in the flash memory. Especially, when the victim block is full, the switch merge can be executed. Therefore, FAB shows better performance than LRU when most of I/O requests are sequential due to the small latency of erase operation when it is triggered. However, when the I/O requests are random, it may lower its performance. For example, if the page counter of every block node is one and the buffer is full. FAB becomes the normal LRU in this extreme case. FAB has Another problem that the recently used pages will be evicted if they belong to the block that has the largest page counter. This problem results from the fact that selecting a victim page is mostly based on the value of page counter, not the page recency. In addition, based on the rule of FAB, only dirty pages are actually written into the flash memory, and all the clean pages are discarded. This policy may results in internal fragmentation, which significantly impacts the efficiency of garbage collection and performance.

2.4.2 Block Padding Least Recently Used

Similar to FAB, Block Padding Least Recently Used (BPLRU) [22] also a block-level buffer algorithm, moreover, it manages the blocks by LRU. Besides block-level LRU, BPLRU adopts a kind of Page Padding technique which improves the performance of random writes. With this technique, when a block needs to be evicted and it is not full, first reads those vacant pages not in the evicted block now but in the flash memory, then writes all pages in victim block sequentially.

So this technique can bring BPLRU sequentiality of flushed block at the cost of more extra read operations, because read operation is the least costly in flash memory. Figure 2.3 shows working of page padding.

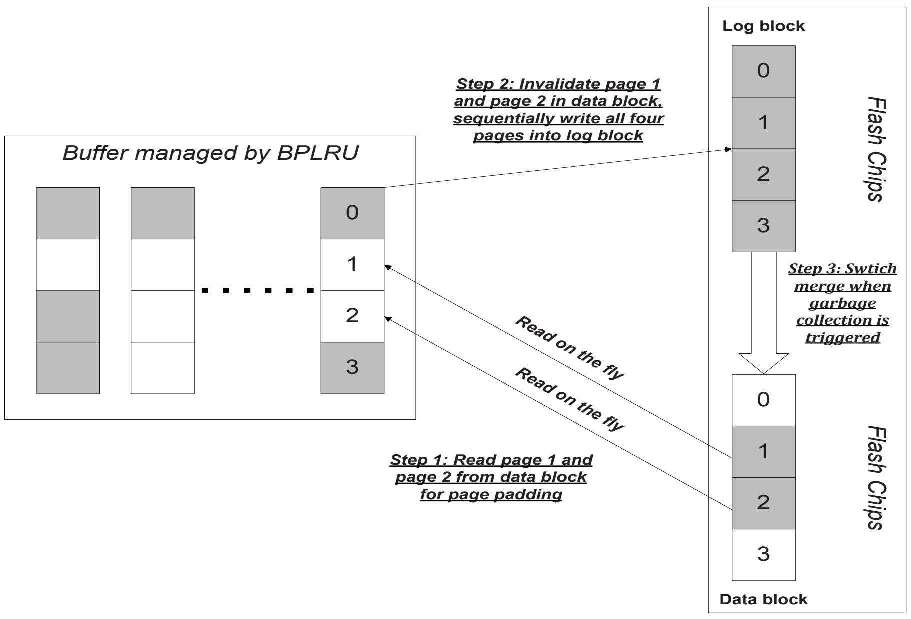

Log block
0
1
2
3
Flash Chips
Step 2: Invalidate page 1 and page 2 in data block, sequentially write all four pages into log block
Buffer managed by BPLRU
0
1
2
3
Step 3: Swtich merge when garbage collection is triggered
Read on the fly
Read on the fly
0
1
2
3
Flash Chips
Step 1: Read page 1 and page 2 from data block for page padding
Data block

Figure 2.3: Page padding technique in BPLRU algorithm

In this example, the current victim block has page 0 and page 3, and page 1 and page 2 are in the data block of flash memory, so BPLRU first reads page 1 and page 2 from the flash memory in order to make the victim block full, then writes the full victim block into the log block sequentially, and only a switch merge may happens.

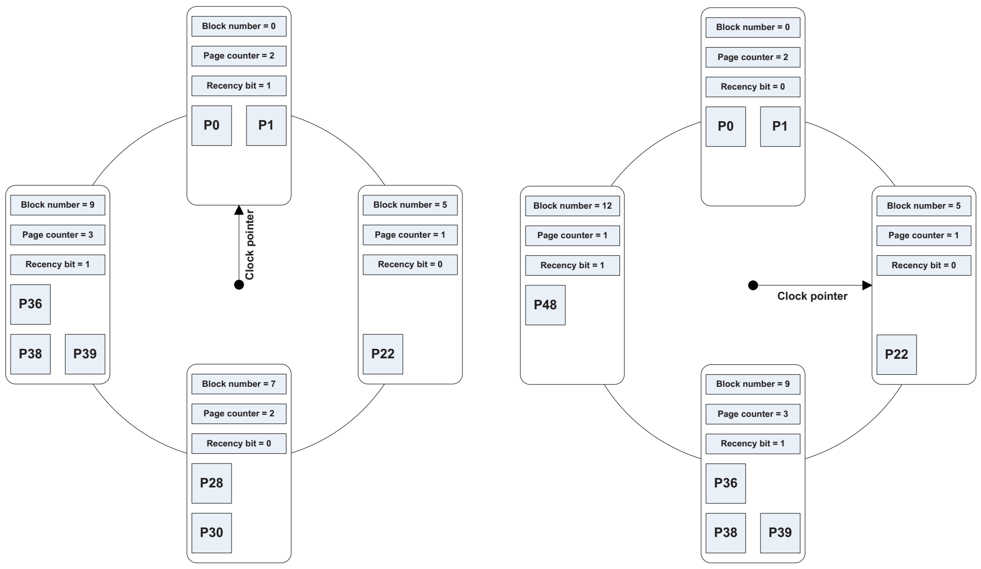
In addition to page padding, BPLRU uses another simple technique called LRU Compensation. It assumes that a block that is written sequentially shows the least possibility that some page is written in this block again in the near future. So if the most recently accessed block is written sequentially, it is moved to the end of LRU list that is least recent used.

It is also worthy to note that BPLRU is just a writing buffer management algorithm. For read operation, BPLRU first checks buffer, if buffer hit happens, it will read data from buffer, but it does not re-arrange the LRU list by read operations. If buffer miss happens, it will directly read data from the physical flash memory storage, and does not allocate buffer space for read data. Normal buffer including FAB allocates buffer for data which is read, but BPLRU does not.

On the one hand, although page padding may increase the read overhead, an efficient switch merge operation is introduced as many as possible instead of the expensive full merge operation, so BPLRU improves the performance of random writes in flash memory. On the other hand, when most of blocks only include few pages, the increased read overhead could be so large that it in turn lowers the performance. In addition, if the vacant pages are not in the flash memory either, the efficiency of page padding could be impacted. Despite the fact that BPLRU concerns the page recency by selecting the victim block in the end of the LRU list, it just considers some page of high recency. In other words, if one of pages in a block has a high recency, other not recently used pages belonging to the same block also stay in the buffer. These pages will waste the space of buffer and increase the buffer miss ratio. Additionally, when page replacement has to happen, all the pages in the whole victim block are flushed simultaneously, including the pages that may be accessed later. Therefore, while spatial locality is aware in block-level scheme, temporal locality is ignored to some extent. So it will result in low buffer space utilization or low buffer hit ratio, and further decrease the performance of SSD. This is also the common issue of block-level buffer management algorithm.

2.4.3 Large Block CLOCK

Large Block CLOCK (LB-CLOCK) [12] also manages buffer with logical blocks. Other than the algorithms above, it is not designed based on LRU, but the CLOCK [11]. A reference bit is tagged in every block in the buffer. When any page of one block is accessed, the reference bit is set to 1. Logical blocks in the buffer are managed in the form of a circular list, and a pointer traverses clockwise. When it has to select a victim block, LB-CLOCK first finds the



(a) the state when buffer is full (b) the state after page 48 is inserted

Figure 2.4: Working of the LB-CLOCK algorithm

block that the clock pointer is pointing to, then checks its reference bit. It sets the reference bit to 0 if the value 1 is shown, and moves the clock pointer to the next block. The clock pointer stops moving until the value 1 of reference bit is encountered. Different from CLOCK algorithm, LB-CLOCK further chooses the victim block from the candidate victim blocks set which includes the blocks whose reference bits are 0 prior to current victim selection until the block which has the largest number of pages is selected. Figure 2.4 shows a running example of LB-CLOCK.

In this example, suppose a block can include 4 pages at most, when page 48 is going to be inserted, LB-CLOCK has to replace a victim block with new block 12 (48/4) due to full buffer now. Now the clock pointer is pointing to block 0 when starting to choose a victim block. Because the reference bit of block 0 is 1, the clock pointer moves next after the reference bit is set to 0. Now it is pointing to block 5 whose reference bit is 0, so the victim selection process is over. As shown in figure 2.4(a), the candidate victim blocks are block 5 and block 7, because their reference bits are 0. Block 0 is not considered because its reference bit is just changed into 0 in this current selection round. Finally, block 7 has the highest number of pages and it is chosen as the final victim block. After replacement, block 12 with page 48 is inserted into the position which is just before block 0 as the clock pointer initially points to block 0, and its reference bit is set to 1, as shown in figure 2.4(b).

In addition, LB-CLOCK makes use of the following heuristic: it assumes that there is low probability that a block will be accessed again in the near future if the last page (i.e. page which has the biggest page number) of the block is written. So if the last page is written and the current block is full, this block is one victim candidate. If the current block is not full after the last page written but it has more pages than the previously evicted block, it is also one victim candidate. Besides, just like BPLRU, the block written sequentially shows low possibility that it will be accessed later such that it can be a victim candidate block.

Similar to BPLRU, LB-CLOCK is also a writing buffer management algorithm, meaning that it will not allocate buffer space for read data. So it reduces the opportunity that a full block is formed in the buffer. When a victim block has to be chosen, LB-CLOCK is different from FAB which takes preference for block space utilization (page counter described in section 2.2), and then recency. On the contrary, it takes preference for recency and then block space utilization. Although it tries to make a balance between the priority given to recency and block space utilization, the assumptions in the heuristic are not strongly supported.

2.4.4 Block-Page Adaptive Cache

Block-Page Adaptive Cache (BPAC) is a write buffer algorithm which aims to fully make use of temporal locality and spatial locality to improve the performance of flash memory. It is a similar research work with our HBM, but has different strategies and details. Here we just briefly shows some similarities and differences before our HBM is introduced.

Just like HBM, BPAC [37] has the framework in which page list and block list are separately maintained to better explore temporal locality and spatial locality. In addition, there exist dynamically page migrations between page list and block list.

In the similar framework, BPAC and HBM has many obvious and significant differences. BPAC is just a write buffer compared to HBM that not only focuses on write operations but also read operations. In addition, BPAC makes use of thresholds based on experiments to control page migrations between page list and block list. Not like BPAC, only dynamically page migration from page list to block list is designed in HBM, because the migration from block list to page list may result in a great number of page insert operations, especially when the number of pages in a block get bigger as capacity of flash memory increases, massively inserting pages into page list lowers the performance of algorithm. Besides two differences above, a new algorithm called LAR is designed in HBM to manage the block list. Moreover, a B+ tree is implemented in HBM to quickly index the nodes. The details of HBM will be shown in the next section.

Chapter 3

Hybrid Buffer Management

We design HBM as a universal buffer scheme, meaning that it is not only for write operations but also read operations. We have assumed that the buffer memory is RAM. A RAM usually exists in current SSDs in order to store mapping information of FTL [22]. When SSD is powered on, mapping information is read from flash chips into RAM. Once SSD is powered off, mapping information is written back to flash chips. We choose to use all of available RAM as buffer for HBM.

Figure 3.1 shows the system overview considered in this thesis. Host system may include a buffer where LRU could be applied. However in this thesis, we do not assume any special buffer algorithm in host side. SSD includes RAM for buffering read and write accesses, FTL and flash chips.

In this chapter, we will describe the design of HBM in detail. Hybrid management and universal feature servicing both read and write accesses are proposed first. Then a locality-aware replacement policy called LAR[1] is designed to manage the block region of HBM. In order to implement page migration from page region to block, we advance threshold-based migration method and meanwhile adopt B+ tree to manage HBM efficiently. Space overhead due to B+ tree is

[1]We designed LAR in the paper ”FlashCoop: A Locality-Aware Cooperative Buffer Management for SSD-based Storage Cluster”, which is published in ICPP 2010.

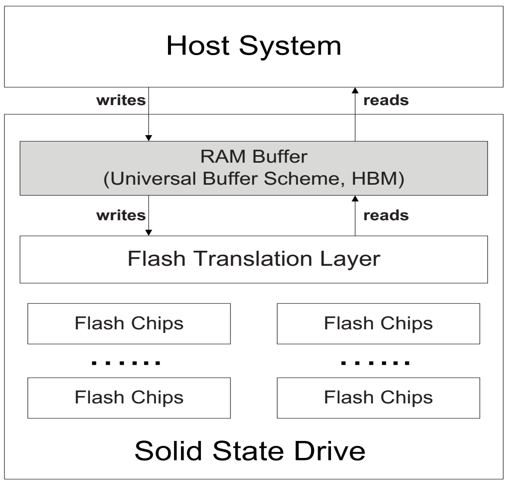


Figure 3.1: System overview. The proposed buffer management algorithm HBM is applied to RAM buffer inside SSD.

also analyzed in theory. How to dynamically adjust threshold will be discussed in the final section of this chapter.

3.1 Hybrid Management

Some previous researches [34][15] claimed that the more popular the file is, the smaller size it has, and large files are not accessed frequently. So file size and its popularity have inverse relation. As [26] reports, 80% of file requests are to files whose size is less than 10KB and the locality type of each request is deeply related to its size.

Figure 3.2 shows the distribution of request sizes over ten traces which we randomly downloaded from Storage Network Information Association (SNIA) [2]. CDF curves are used to show percentage of requests whose sizes are less than a certain value. As shown in figure 3.2, most of request sizes are between 4K and 64K, and few request sizes are bigger than 128K. Although only ten traces are analyzed, we can see that small size request is much more popular than big size request.

Random accesses are small and popular, which have high temporal locality. As shown in Table 1.1, page-level buffer management exhibits better buffer space

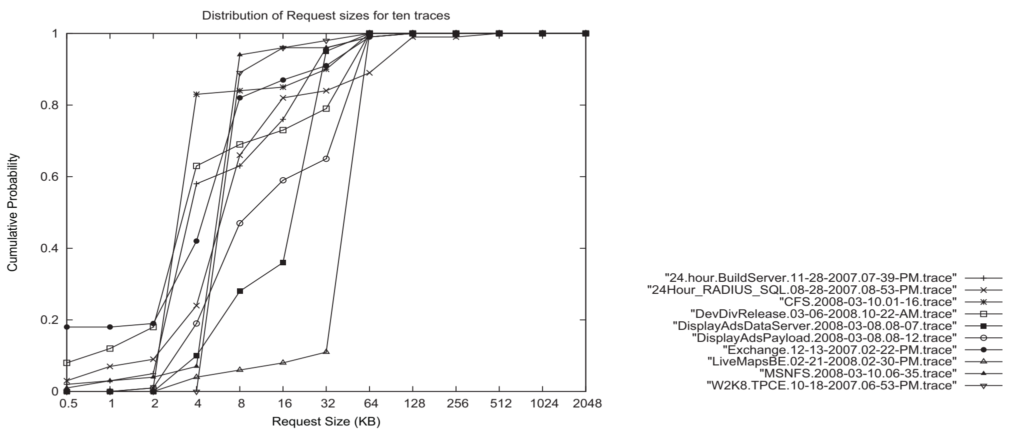


Figure 3.2: Distribution of request sizes for ten traces from SNIA [2]

utilization and it is good at exploiting temporal locality to achieve high buffer hit ratio. Sequential accesses are large and unpopular, which have high spatial locality. The block-level buffer management scheme can effectively make use of spatial locality to form a logical erasable block in the buffer, and meanwhile good block sequentiality can be maintained in this way.

Enterprise workloads are a mixture of random and sequential accesses. Only page-level or only block-level buffer management is not enough to fully utilize both temporal and spatial localities among enterprise workloads. So it is reasonable for us to make use of hybrid management, which divides the buffer into page region and block region, as shown in the figure 3.3. These two regions are managed separately. Specifically, in the page region, buffer data is managed at single page granularity to improve buffer space utilization. Block region operates at the logical block granularity that has the same size as the erasable block size in the NAND flash memory. One unit in the block region usually includes two pages at least. However, this minimum value can be adjusted statically or dynamically, which will be explained in the section 3.6.

Page data is either in page region or in block region. Both regions serve incoming requests. It is worthy to note that many existing buffer management algorithms can be used to manage pages in page region such as LRU, LFU. LRU is the most common buffer management algorithm in operating systems.

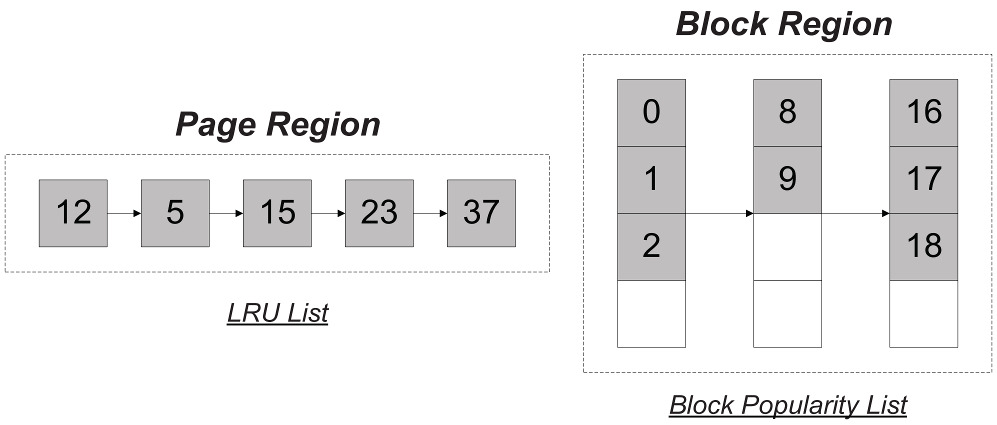


Figure 3.3: Hybrid Buffer Management. Buffer space is divided into two regions, page region and block region. In the page region, buffer data is managed and sorted in page granularity, while block region manages data in block granularity. Page can be placed in either of two regions. Block in block region is selected as victim for replacement.

Due to its efficiency and simplicity, pages in page region are organized as page-level LRU list. When a page buffered in the page region is accessed (read or write), only this page is placed at the most recent used end of the page LRU list. As for block region, we design a specific buffer management algorithm called LAR which will be described in the section 3.3.

Therefore, the temporal locality among the random accesses and spatial locality among sequential accesses can be fully exploited by page-level buffer management and block-level buffer management respectively.

3.2 A Buffer for Both Read and Write Operations

As for flash memory, the temporal locality and spatial locality can be understood as *block-level temporal locality*: the pages in the same logical block are likely to be accessed (read/write) again in the near future. In the real application, read and write accesses are mixed and exhibit the block-level temporal locality. In this case, separately servicing the read and write accesses in different buffer space may destroy the original locality present among access sequences. Some existing buffer managements for flash storage such as BPLRU and LB-CLOCK only allocate memory for write requests. Although it creates more space for write requests than the buffer which serves both read and write operations, however, it may suffer from more extra overhead due to the read miss. As [12] claims, servicing foreground read operations is helpful for the shared channel which sometime has overload caused by both read and write operations. Moreover, the saved channel's bandwidth can be used to conduct background garbage collection task, which helps to reduce the influences of each other. In addition, read operations are very common in some read intensive applications such as digital picture reader, so it is reasonable for buffer to serve not only write requests but also read operations.

Taking BPLRU as an example, as described in section 2.4.2, it is designed only for writing buffer. In other words, BPLRU exploits the block-level temporal locality only among write accesses, and especially full blocks are constructed only through writes accesses. So in this case, there is not much possibility for BPLRU to form full blocks when read misses happen. BPLRU uses page padding technique to improve block sequentiality of flushed data at a cost of additional reads, which in turn impacts the overall performance. For random dominant workload, BPLRU needs to read a large number of additional pages, which can be seen in our experiment later. Unlike BPLRU, we leverage the block-level temporal locality not only among write accesses but also read accesses to naturally form sequential block and avoid large numbers of extra read operations. HBM treats read and write as a whole to make full use of locality of accesses, meanwhile, HBM groups both dirty and clean pages belonging to the same erasable block into a logical block in the block region. How to read or write data will be presented in detail in section 3.3.

3.3 Locality-Aware Replacement Policy

This thesis views negative impacts of random writes on performance as penalty. The cost of sequential write is much lower than that of random write. Popular data will be frequently updated. When replacement happens, unpopular data should be replaced instead of popular data. Keeping popular data in buffer as long as possible can minimize the penalty. For this purpose, we give preference to random access pages for staying in the page region, while sequential access pages in block region are replaced first. What's more, the sequentiality of flushed block is beneficial to garbage collection of flash memory.

Block popularity - small sized file is accessed frequently and big sized file is not accessed frequently. In order to make good use of the access frequency in block region, block popularity is introduced, which is defined as block access frequency including reading and writing of any pages of the block. Specifically, when a logical page of a block is accessed (including read miss), we increase the block popularity by one. Sequentially accessing multiple pages of a block is treated as one block access instead of multiple accesses. Thus, block with sequential accesses will have low popularity value. One advantage of using block popularity is that full blocks formed due to accessing big size file usually have low popularity. Full blocks will be probably flushed into flash memory when replacement is necessary, which is beneficial to reduce garbage collection overhead of flash memory.

A locality aware replacement policy called LAR is designed for block region. The functions of LAR in form of pseudo code are shown in Algorithm 3.1, 3.2 and 3.3, which consider the case that the request size is only one page data. For requests which include more than two pages, several small sized requests, each of which only includes the pages belonging to a single block, will be processed after breaking up the original big request. For one request, sequentially accessing multiple pages of a block is treated as one block access, thus, the block popularity will be only increased by one.

How to read and write - when requested data is in the page region, re-arrange the LRU list of page region. Because LAR is designed for block region, here all the operations below happen in the block region.

Algorithm 3.1: Read Operation For LAR

```
   Data: LBN(logical block number), LPN(logical page number)
 1 if found then
 2 |   Read page data in the buffer;
 3 end
 4 else
 5 |   Read page data from flash memory;
 6 |   if not enough free space then
 7 |   |   Replace() ;                          /* refer to Algorithm 3.3 */
 8 |   end
 9 |   if LBN is not found then
10 |   |   Allocate a new block;
11 |   |   Write page data in the buffer;
12 |   |   Block popularity = 1;
13 |   |   Page state for LPN = clean;
14 |   |   Number of pages = 1;
15 |   end
16 |   if LBN is found but LPN is not found then
17 |   |   Write page data in the buffer;
18 |   |   Block popularity ++;
19 |   |   Page state for LPN = clean;
20 |   |   Number of pages ++;
21 |   end
22 |   Re-arrange the LAR list;
23 end
```

For read requests, if the read request is hit, read data directly (Alog 3.1, lines 1-3), and re-arrange the block region based on LAR (Alog 3.1, line 22). Here, we simply suppose that block region is managed in LAR list, the specific data structure managing block region will be presented in section 3.6. Otherwise, HBM would then fetch data from flash memory and a copy of the data will be placed in the buffer as reference for future requests (Alog 3.1, line 5). At this time, if buffer is full, replacement operation is triggered to produce more space (Alog 3.1, lines 6-8). When there is enough space to hold new data, put it in the buffer. Then two cases should be considered. If the logical block which new page belongs to has been already in the buffer, we update the corresponding information of this logical block (Alog 3.1, lines 16-21); otherwise, we should allocate a new logical block first (Alog 3.1, lines 9-15). Finally, re-arrange the LAR list (Alog 3.1, line 22).

Algorithm 3.2: Write Operation For LAR

```
Data: LBN(logical block number), LPN(logical page number), PageData
if found then
    Update the corresponding page in the buffer;
    Block popularity++;
    Page state for LPN = dirty;
end
else
    if not enough free space then
        Replace() ;                              /* refer to Algorithm 3.3 */
    end
    if LBN is not found then
        Allocate a new block;
        Write page data in the buffer;
        Block popularity = 1;
        Page state for LPN = dirty;
        Number of pages = 1;
    end
    if LBN is found but LPN is not found then
        Write page data in the buffer;
        Block popularity ++;
        Page state for LPN = dirty;
        Number of pages ++;
    end
    Re-arrange the LAR list;
end
```

For write requests, if the write request is hit, modify the old data, update the corresponding information of the logical block which requested page belongs to and re-arrange the LAR list; otherwise, the operations are similar to the ones for read requests, except that page state should be set dirty (Alog 3.2, line 4, 14 and 20).

Victim block selection **-** every page in the buffer keeps a state value for itself: *clean* and *dirty*. Modified page will be dirty, and page read from flash memory due to read miss will be clean. When there is not enough space in the buffer, the least popular block indicated by block popularity in the block region is selected as victim (Alog 3.3, line 1). If more than one block has the same least popularity,

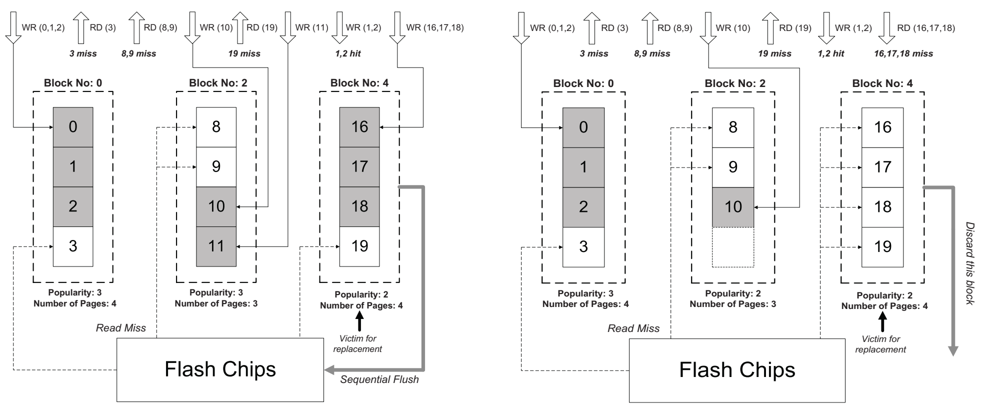
Algorithm 3.3: Replacement For LAR

```
1  Find the victim block which has the smallest block popularity;
2  if not only one victim block then
3      Block of them, which has the largest number of pages, will be chosen;
4      if Still not only one victim block then
5          randomly pick one from them;
6      end
7  end
8  if there are dirty pages in victim block then
9      Both dirty pages and clean pages in victim block are sequentially
       flushed;
10 end
11 else
12     All the pages in victim block will be discarded;
13 end
14 Re-arrange the LAR list;
```

a block having the largest number of buffered pages is further selected as a victim (Alog 3.3, line 3). After this selection, if there is still more than one block, the final victim block will be further chosen randomly from them (Alog 3.3, lines 4-6).

Selection compensation - only if block region is empty, we select the least recently used page as victim from page region. The pages belonging to the same block as this victim page will be also flushed sequentially. This policy tries to avoid flushing single page, which has high negative impact on garbage collection and internal fragmentation.

How to flush the victim block - once a block is selected as victim, there are two cases to deal with: (1) If there are dirty pages in this block, both dirty pages and clean pages of this block are sequentially flushed into flash memory (Alog 3.3, lines 8-10). This policy guarantees that logically continuous pages can be physically placed onto continuous pages, so as to avoid internal fragmentation and keep the sequentiality of flushed pages. By contrast, FAB flushes only dirty pages in the victim block and discards all the clean pages without considering the sequentiality of flushed data. (2) If there are no dirty pages in the block, all



(a) The victim block which has the smallest block popularity is sequentially flushed

(b) The victim block is further chosen by number of pages, and discarded due to no dirty pages

Figure 3.4: Working of LAR algorithm

the clean pages of this block will be discarded (Alog 3.3, lines 11-13).

Figure 3.4 illustrates working of our LAR. In figure 3.4(a), upon write request WR(0,1,2) is coming, because they belong to block 0 and block 0 is not in the buffer, a new block 0 should be allocated first, and pages of 0, 1 and 2 are written in the buffer. Therefore, the popularity of block 0 is 1 and number of pages is 3. As read request RD(3) is coming, one missed page is read from flash chips and stored in the block 0, whose popularity is then increased by 1 and number of pages is updated as 4. Similarly, pages of 8 and 9 form block 2 with popularity 1. As write request WR(10) is coming, both popularity and number of pages in block 2 are increased by 1. Read request RD(19) initially forms block 4, whose popularity is 1 and number of pages is 1.Write request WR(11) increases the popularity and number of pages of block 2 by 1, respectively. Two page hits happen when write request WR(1,2) is coming, which updates the popularity of block 0 as 3. Finally, write request WR(16, 17, 18) updates the popularity and number of pages of block 4 as 2 and 4, respectively. Of three blocks in the buffer, block 4 is regarded as victim block due to its least popularity, and it will be sequentially flushed into flash chips.

Due to the different request sequence from figure 3.4(a), the final state of buffer in figure 3.4(b) is different. Specifically, the popularity, number of pages of block and page states are different. When replacement happens, block 4 is still victim block although its popularity is equal to the one of block 2, because its number of pages is bigger than block 2. Then block 4 will be discarded since all the pages in block 4 are clean.

After LAR is used, more sequential requests are passed to the flash chips, while most random requests are filtered. Requests which show spatial stronger locality can be processed efficiently.

3.4 Threshold-based Migration

A threshold which is the minimum number of pages included in each block in block region can be set statically or dynamically. Whichever policy is applied, buffer data in page region will be migrated to block region if the number of pages in a block reaches the threshold, as shown in figure 3.5. How to determine the threshold value will be discussed in section 3.6. For instance, in figure 3.5, suppose that the threshold is 3, page 0, page 1 and page 2 which belong to block 0 are all in the page region at the same time. According to threshold-based migration, these three pages should be constructed to block 0 and migrated into the block region. Block region is updated then.

The blocks in the block regions are formed in the two ways: one the one hand, when a large sized request involving many continuous pages is issued, the block may be constructed directly. On the other hand, it could be constructed due to many small sized requests involving pages belonging to the same block as block 0 in figure 3.5. Therefore, with filter effect of the threshold, random pages due to small size requests will stay in the page region, while the selected blocks as block 0 in figure 3.5 reside in the block region. Temporal locality among random pages and spatial locality among sequential blocks can be fully utilized in the hybrid buffer management.

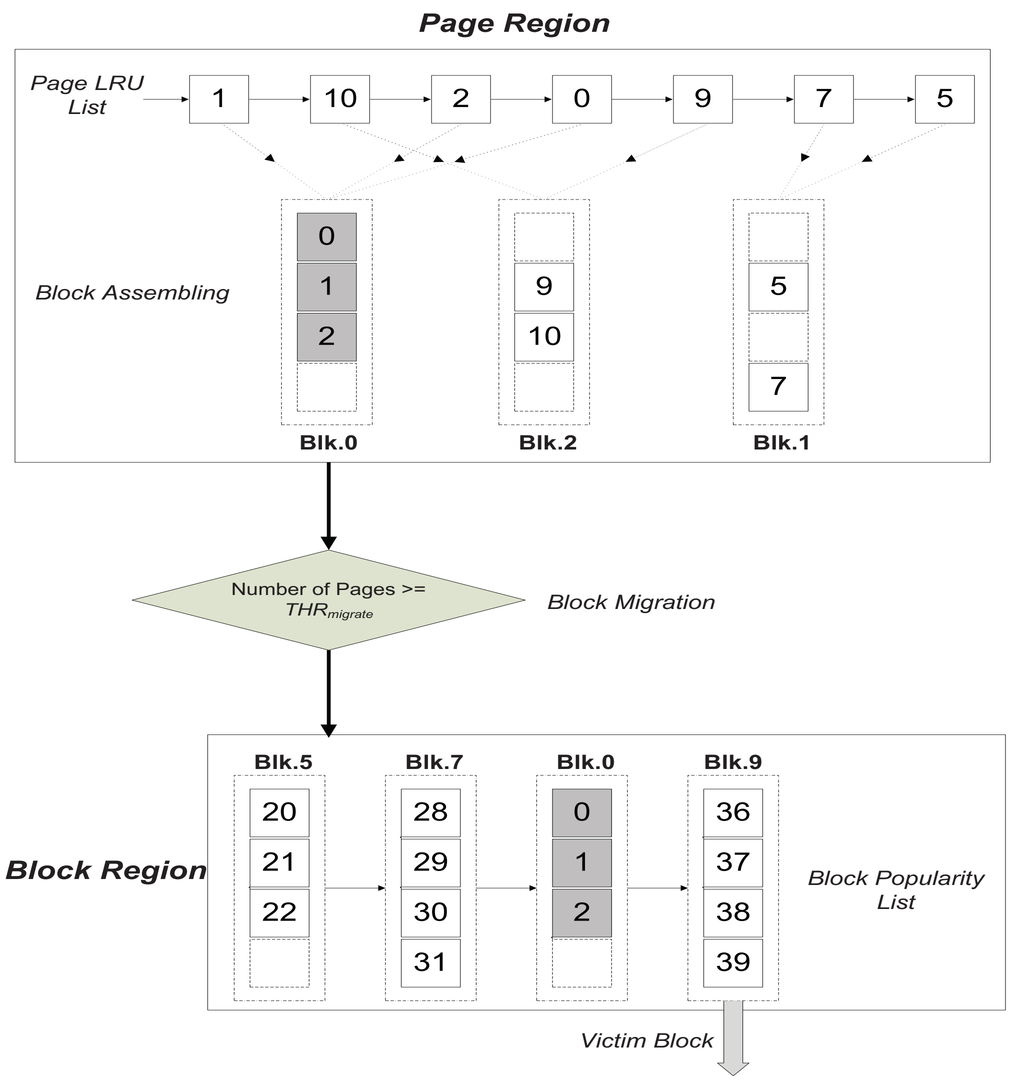


Figure 3.5: Threshold-based Migration. $THR_{migration}$ is a threshold which denotes the minimum number of pages for a block in block region. Buffer data in page region will be migrated to block region only if the number of pages in a block reaches $THR_{migration}$. Grey boxe (Blk.0) denotes that a block is found and migrated to block region. An erase block consists of 4 pages.

3.5 Implementation Details

Suppose page region and block region are managed by LRU and LAR list respectively, finding an associated page in buffer is not efficient, and we must traverse two lists every time searching pages are necessary. So CPU power should be cared when we design HBM to search one particular page quickly. In addition, how to implement threshold-based data migration should be also efficiently designed. Meanwhile, limited memory space should be achieved due to the precious memory size inside SSD.

3.5.1 Using B+ Tree Data Structure

B+ tree [1] is primarily used in data storage and it is used for quickly data search requirement. For example, in file systems, some of which use B+ tree for metadata indexing. Unlike binary search tree, a high fan out value helps B+ tree to shorten the path length to search an element in the tree. In some relational database management systems such as IBM DB2 [1], B+ tree is supported for table indices.

We adopt B+ tree indexing to manage data for two reasons: one is its efficient retrieval for a particular page; the other is that the memory size for B+ tree is limited, which will be analyzed in section 3.5.3.

Figure 3.6 shows B+ tree indexing to manage data for HBM. Two basic data structures should be present first: *block node* and *page node*. Block node describes a block in terms of block popularity, number of pages in the block including clean and dirty pages, and pointer array which points to the page nodes; Page node describes a page in terms of page number, two link pointers for LRU, and physical address of page data. B+ tree index is built over the block nodes. It uses block number as key to assemble the pages that belong to the same block. The leaf node of B+ tree has pointers to corresponding block nodes.

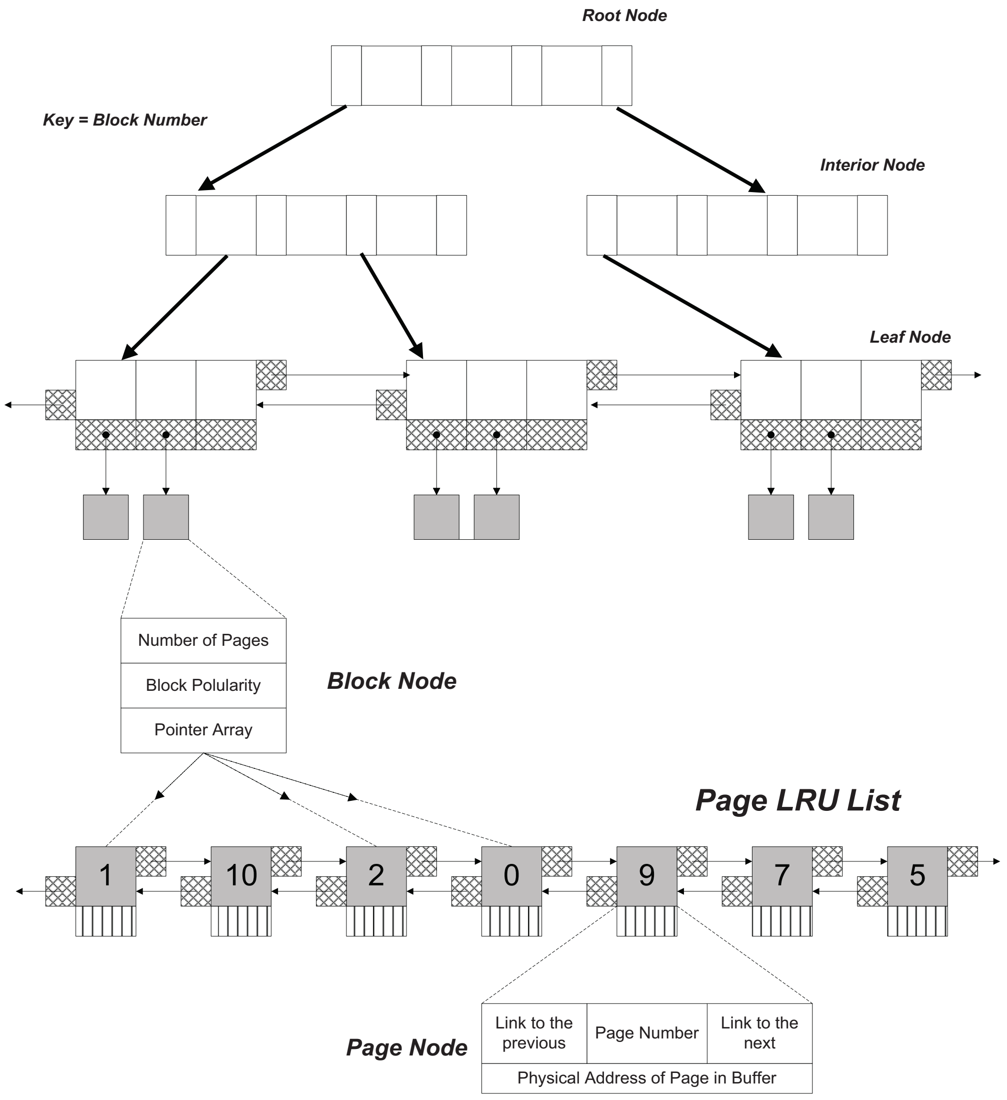


Figure 3.6: B+ tree to manage data for HBM

3.5.2 Implementation for Page Region and Block Region

Figure 3.7 shows the implementation for page region and block region of HBM.

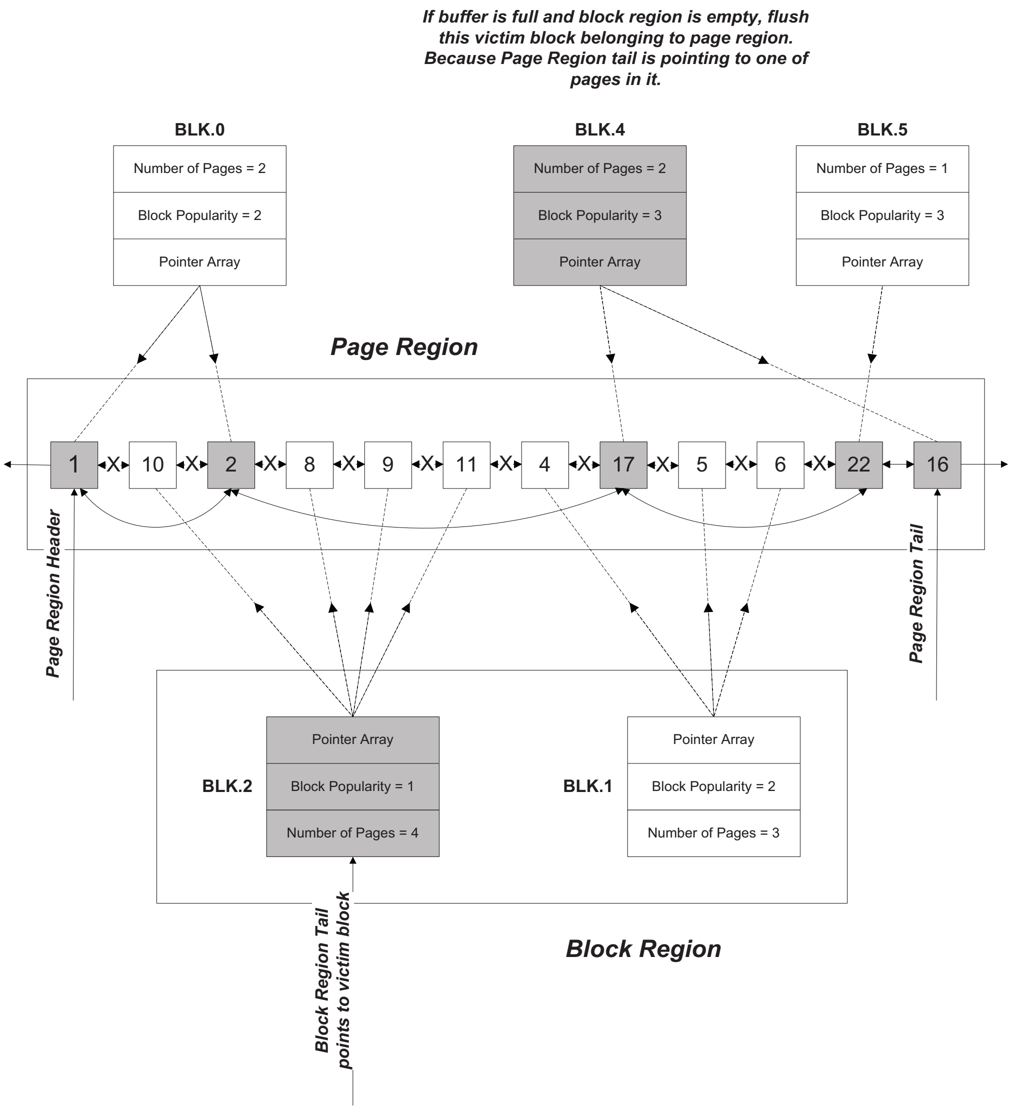


Figure 3.7: Data management in page region and block region

Forming block region - initially, all the page nodes in the buffer are included in page region, meaning that they are all linked by pointers for LRU. A page region header indicates the most recent used end of LRU list, pointing to the first page node in page region. Meanwhile, a page region tail indicates the least recent used end of LRU list, pointing the last page node in page region. Upon arrival of an access in the page region, we deal with it in following way. After dividing the page number by the number of pages per block, we first get the block number.

Then we search the B+ tree using the block number to find the corresponding block node. If the block node exists, we update the block node including block popularity, number of pages and pointer (if page does not exist, add a new page node into LRU list, and a pointer corresponding to new page node is added into block node). If the block node does not exist, we add a new page node and a corresponding block node, and then update the LRU list. If the number of pages in the block is below the threshold, update the LRU list; otherwise, all the pages in the block will be migrated to block region. Specifically, we extract these pages from LRU list in the page region by modifying the related links for LRU of page nodes, and then the two links for LRU of extracted page nodes are set NULL, for example, marker ”X” in figure denotes that there are not any links between two page nodes. In other words, how to determine whether a page node belongs to page region or block region depends on the links for LRU of it. If the link is NULL, the page node is in block region; otherwise, it is in page region. The reason why we manage page region and block region in this way is that we do not have to really manage a LAR list (the replacement policy in block region is LAR) in block region. All we need is quickly finding the victim block when replacement has to happen.

***Selecting the victim block* -** the victim block should has the smallest block popularity, such as BLK.2 in figure 3.7. If there is more than one block like this, the one that has the least number of pages will be chosen. So there is a pointer called block region tail which points to the current victim block.

When one block node is just migrated into block region, we compare this block node with the victim to see whether this block can replace the current victim block, and update the block region tail pointer if necessary. When the victim block is updated, we need to traverse all the block nodes by leaf nodes of B+ tree to determine whether we need to change the victim block. Because updating the victim block seldom happens, the cost of traversing all the block nodes is limited.

***What to do when block region is empty* -** that the pointer of block region tail is NULL means that the block region is empty. In this condition, if we have to replace pages from page region for free space, the page which the page region tail points to will be chosen, and besides this page, other pages that belongs to the same block as this page will also be chosen. In other words, we first find the victim page by page region tail, then search the block node that belongs to, e.g., BLK.4 in figure 3.7, and sequentially flush all the current pages indicated in the block node from low page number to high page number.

3.5.3 Space Overhead Analysis

By using B+ tree indexing, the pages belonging to the same block can be quickly searched and located. Meanwhile, the space overhead of B+ tree and the block node is limited. As shown in figure 3.6, B+ tree generally includes two parts: leaf nodes and interior nodes (including root node). In order to analyze the space overhead, we first make following assumptions:

1. Integer or pointer type consumes 4 bytes;

2. B+ tree uses a ”fill factor” to control the growth and shrinkage. A 50% fill factor [32] would be the minimum for any B+ tree. In other words, at least half of child pointers are valid. The typical fill factor is 67% in practice [32], however we set it to be 50% for convenience of analysis. In addition, as fill factor increases, the number of interior nodes will decrease. In order to analyze the worse case, the minimum fill factor 50% should be set. In this case, the leaf node will also remain at least half-full;

3. Suppose the ratio of number of interior nodes to number of leaf nodes is r, $0<r<1$. In practice, the number of interior nodes is much smaller than leaf nodes. B+ tree is height balanced and the fan out of B+ tree is 133 on average [32]. In this condition, because the fill factor is at least 50%

according to assumption 2, r is usually much smaller than 1. Here, we set r to be 0.5, meaning that the number of interior nodes is half of the number of leaf nodes. Suppose that the size of one interior node is the same as one leaf node (although interior node is strictly a little bigger than leaf node, it will not impact on analysis here due to few interior nodes);

4. Every block node including block popularity integer value, number of pages integer value and one pointer consumes 12 bytes. Here, we assume that pointer array only includes one pointer. In this case, the number of the block nodes is max for worse case. So the number of block node is equal to the number of page node;

5. Every page node has 4*4=16 bytes, including page number, two link pointers for LRU, and one pointer indicating physical address of page data;

When the length of page list is L, the size of buffer pages is L*2*1024 bytes (one page is 2KB) in sum. Accordingly, the number of block nodes is L (*assumption 4*), the total size of block nodes is L*12 bytes (*assumption 4*), the total size of leaf nodes is L*2*8 bytes (*assumption 2,* one slot corresponding to one block node in leaf node consumes 8 bytes, one integer for a key which is block number, one pointer for block node), the total size of interior nodes is (L*2*8)/2=L*8 bytes (*assumption 3*), and the total size of page nodes is L*16 bytes. Therefore, the space overhead of B+ tree, block nodes and page nods in sum is L*12 + L*2*8 + L*8 + L*16 = L*52, which is less than 3% of L*2*1024 buffer pages.

Although the above analysis of space is not strict and precise, it can generally reflect the real space overhead. For example, leaf nodes of B+ tree are linked together to form a double linked list according to definition of the original B+ tree. Because it is difficult for us to calculate the number of leaf nodes and furthermore get this additional space overhead due to double links, it is ignored. However, in the worse case that the number of leaf nodes is equal to the number of block nodes L (it is even impossible due to 50% fill factor), this additional space overhead is only L*4*2 (two pointers, one is pointing to the previous leaf node, one is pointing to the next leaf node). To sum up, the analysis basically indicates that total space overhead of HBM is small and limited.

3.6 Dynamic Threshold

Our objective is to improve sequentiality of accesses passed to flash memory, at the meantime maintain high buffer space utilization. The threshold is utilized to balance the two objectives, whose value is critical to the efficiency of our proposed HBM buffer management scheme.

In order to investigate the proper threshold value, we tested the effects of different threshold values 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 for different workloads. Even for the same workload, varied threshold value during processing workloads can be better for improving performance than the fixed threshold value. Therefore, statically setting threshold value can not adapt to enterprise workloads with complex features and interleaved I/O requests.

We realize that the value of threshold is highly dependent on workload features. For small sized random dominant workload, a small value is suitable because it is difficult to form big sized sequential blocks, and few blocks exist in the block region. Once we need to replace pages for free space, pages in page region have to be flushed, which exhibits little sequentiality. For sequential dominant workload, a large value is desirable because a lot of partially filled blocks instead of full blocks will be migrated from page region to block region if a small threshold is set, which lowers the space utilization and buffer hit ratio.

HBM uses the following heuristic to achieve the dynamic threshold-based migration: we use $THR_{migrate}$ to denote the threshold. Clearly, the value of $THR_{migrate}$ must be at least 1. As long as the number of pages in some block is higher than or equal to $THR_{migrate}$, it will be placed in the block region. So when $THR_{migrate}$ is 1, all the pages are in the block region, which is managed by LAR algorithm; suppose a block can at most accommodate 64 pages, when $THR_{migrate}$ is larger than 64, all the pages are in the page region, which is managed by LRU algorithm. N_{block} represents the total size of block region in term of number of pages. N_{total} represents the total size of buffer space in term of number of pages. We use γ to denote the ratio between N_{block} and N_{total}. The value of γ is under following simple constraint (3.1), which is used to control whether to enlarge or reduce the threshold during processing workloads.

$$\alpha \leq \gamma = \frac{N_{block}}{N_{total}} \leq \beta \tag{3.1}$$

Where α, β are parameters which should be configured as the minimum and maximum reasonable ratio between N_{block} and N_{total}.

On the one hand, since the block region is used to store block candidates whose pages are sequential enough for replacement, and the total size of these block candidates should be much smaller than the total size of others, therefore, the size of the block region should be much smaller than the size of page region. In addition, once one block is migrated to block region, it will not be moved back to page region again, because LRU list in page region can not adjust to this block, in other words, it is difficult to make sure where the pages in this block should be inserted in the LRU list. Therefore, the buffer hit ratio may decrease as the size of page region shrinks. So we should keep the block region in small size. For example, we set the value of β as 10% for small buffer size such as 4MB and 20% for large buffer size such as 64MB.

On the other hand, if block region is too small, it may be consumed quickly if big sized requests come when buffer is full. In this case, we have to flush pages in page region. Therefore, the value of α is also related to distribution of request sizes to some extent. As described in section 3.1, 80% of file accesses are to files of less than 10KB and most of request sizes are between 4K and 64K as shown in figure 3.2. Suppose that page size in flash memory is 2K, the size of block region is at least 32 pages (64K/2K) for ten traces in section 3.1. Because a full block usually consists of 64 pages, it means that block region can be 64 pages (one full block size). However, we can not know the distribution of request sizes in advance, and the request size may be larger sometimes. So simply, we had better set the smallest value of α a little larger, such as the ratio of 128 pages (2 full blocks, 256K) to the total size of buffer.

It is worth noting that 2 full blocks can occupy large part of buffer size as for small size buffer. For example, as for 1MB buffer, if we still set α as the ratio of 128 pages to 512 pages (8 full blocks, 1MB), it means that α is equal to 25% (128/512) which is bigger than 10%, the assumed value of β. So it contradicts the constraint (3.1) where α is not larger than β. Therefore, for the small sized buffer where β is smaller than α based on general rules described above, we still set α as the ratio of 128 pages to the total size of buffer, however, β will be set as the ratio of 256 pages (4 full blocks) to the total size of buffer.

Table 3.1 shows the our proposed rule for setting the values of α and β.

Table 3.1: The rules of setting the values of α and β

General rule: set α as the ratio of 128 pages to the total size of buffer; set β as 10% for small buffer size and 20% for large buffer size; **Exception:** if the α is bigger than β after calculation based on the general rule, then set α as the ratio of 128 pages to the total size of buffer; set β will be set as the ratio of 256 pages to the total size of buffer

The size of block region should be adjusted dynamically based on constraint 3.1 above. Algorithm 3.4 shows how we adjust $THR_{migrate}$ dynamically.

Initially, the value of $THR_{migrate}$ is set to 1. The size of block region is monitored every time its total size is changed. It is worth noting that not every request

Algorithm 3.4: Dynamically_Adjust_Threshold

```
1 if $N_{block}$ is changed $\bigwedge$ old $THR_{migrate}$ stays the same at least for 100
  requests since the last time it was changed then
2     if $\gamma > \beta \bigwedge THR_{migrate} \leq 64$ then
3         $THR_{migrate}$ += 1;
4     end
5     if $\gamma < \alpha \bigwedge THR_{migrate} \geq 2$ then
6         $THR_{migrate}$ -= 1;
7     end
8 end
```

can make the size of block region changed, such as requests which are hit in the buffer. The maximum $THR_{migrate}$ is 65, which will occur when γ is larger than β and old $THR_{migrate}$ is equal to 64. The minimum $THR_{migrate}$ is 1, which will occur when γ is smaller than α and old $THR_{migrate}$ is equal to 2. We do not change $THR_{migrate}$ for at least 100 requests since last change happened. Because the value of γ does not vary too much especially for big sized buffer, if we change $THR_{migrate}$ for every request when N_{block} varies, $THR_{migrate}$ will quickly vary from 1 to 65 or from 65 to 1, and stay 65 or 1 for quite a long time, other possible $THR_{migrate}$ values between 1 and 65 will not be evaluated enough in that case.

After satisfying the first condition (Algo 3.4, line 1), if the value of γ becomes larger than β , 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 page migration from the page region to the block region. Then, the value of $THR_{migrate}$ will be increased by 1 until γ is less than β. On the other hand, the value of $THR_{migrate}$ will be decreased by 1 if γ becomes smaller than α.

Chapter 4

Experiment and Evaluation

In this chapter, we use simulation to evaluate HBM and compare it to three other buffer management algorithms: BPLRU, FAB and LB-CLOCK. We first introduce our workload traces and experiment environment. Then the experiment results are analyzed based on performance and energy consumption.

4.1 Workload Traces

Both real and synthetic traces are used in our experiments to study the performance of different buffer management algorithms. The features of our traces are presented in Table 4.1.

Table 4.1: 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
Synthetic Trace	20.32	70	80	52.22

A read-dominant I/O trace is used which is based on an OLTP application running at a financial institution [3] made available by the Storage Performance

Council (SPC), henceforth referred to as Financial trace. We also employ a write-dominant I/O trace called CAMWEBDEV, which was 1-week block I/O trace of enterprise servers at Microsoft [2] made available by the Storage Network Information Association (SNIA) [2]. Besides read and write dominant workloads, we want to assess the behavior of different buffer management schemes under mixed workloads. For this purpose, we use MSNFS which was collected for MSN Storage file server for duration of 6 hours and Exchange traces which were production traces collected at Microsoft using event tracing for Windows. All these two traces are also available by SNIA. Finally, we also use a synthetic trace which is generated by DiskSim 4.0 [6] to study the behavior of different buffer management schemes for a sequential dominant workload, which is referred to as Synthetic trace. The five traces used in our experiment cover workload characteristics from random to sequential and from read dominant to write dominant.

Originally, the address space of these traces is different. Some traces are within 32 GB, others are out of 32GB. Because the size of SSD in our simulator is fixed at 32GB, we extract the requests whose access address is within 32GB in order to adapt to our simulator.

4.2 Experiment Setup

4.2.1 Trace-Driven Simulator

FlashSim [25] is an event-driven simulator for NAND flash memory based Solid State Drive, designed by the Pennsylvania State University. It is developed based on the well-regarded DiskSim simulator [6], and adds a SSD module. Currently, FlashSim implemented three types of FTL, which are page level FTL, DFTL [16] that is another kind of page level FTL, and FAST. It also can analyze the energy consumption of these different FTL schemes.

In order to research on the device-level buffer management inside SSD using FlashSim, we add BAST FTL scheme into FlashSim, because some existing buffer management algorithms are based on this basic log-block FTL scheme. Then we add a buffer module and implement four buffer management algorithms inside SSD, which are BPLRU, FAB, LB-CLOCK, and HBM.

4.2.2 Environment

For our simulation, we assume a 32GB SLC NAND flash memory based SSD with 64 2KB pages in each block. The size of data register or cache register is also 2KB. The maximum erase times of every block is 100000. BAST FTL algorithm is configured to use 3% [24] of capability of SSD as log blocks. The trivial cost of updating mapping information is ignored. We simulate various evaluation metrics while varying the RAM buffer sizes from 1MB to 64MB. Parameters in Table 4.2 for performance calculation is used. These values are taken from [37]. The simulator is run on a Pentium dual-core 2.33GHz PC with Ubuntu 8.04 and Linux kernel 2.6.24.

Table 4.2: Timing parameters for simulation

Operations	**Time(μs)**
Page Read to Register	25
Page Program (Write) from Register	200
Block Erase	1500
Serial Access to Register (Data bus)	100

4.2.3 Evaluation Metrics

In this thesis, we utilize the following metrics:

- ***Response time***, which is seen at the I/O driver (the sum of the device service time and time spent waiting in the driver's queue [16])

- ***Buffer hit ratio***, which is calculated as the ratio between the number of pages hit in the buffer and total number of requests pages

- ***Number of erases***, which is the indicator of the garbage collection overhead

- ***Distribution of write length***, which indicates the sequentiality of write accesses passed to flash memory, and it characterizes the behavior of different buffer management schemes

In addition, the total number of read operations including read miss and extra reads is evaluated to analyze the additional overhead.

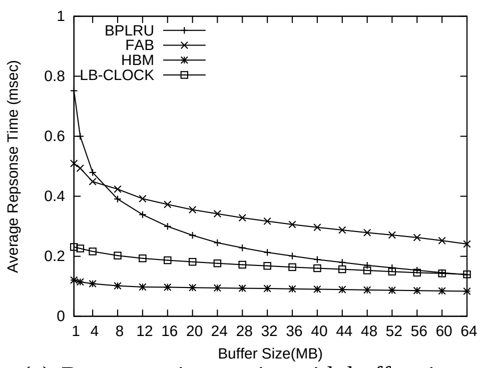
4.3 Analysis of Experiment Results

Figure 4.1, 4.2, 4.3, 4.4 and 4.6 show the average response time, buffer hit ratio, number of erases and distribution of write length of BPLRU, FAB, LB-CLOCK and HBM buffer management schemes under the five workloads when we vary buffer size. To indicate the write length distribution, we use CDF curves to show percentage (shown on Y-axis) of written pages whose sizes are less than a certain value (shown on X-axis). We present CDF curves for 1MB buffer in figure 4.1 through figure 4.4 for random workloads and figure 4.6 for synthetic workload. We also present CDF curves for 16MB buffer under different workloads in figure 4.5.

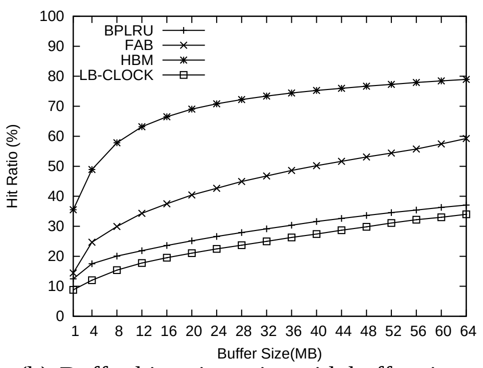
4.3.1 Analysis on Different Random Workloads

1) Financial trace

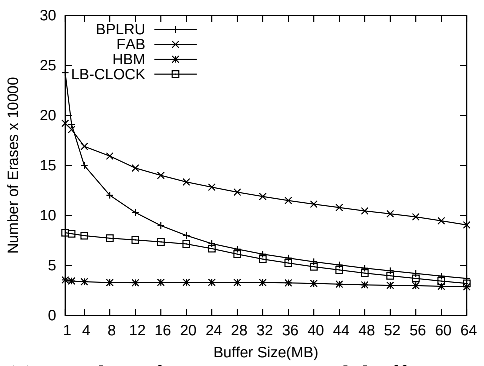
Figure 4.1 shows that HBM outperforms BPLRU, FAB and LB-CLOCK in terms of average response time, buffer hit ratio, number of erases and number of sequential writes under the completely read dominant trace.



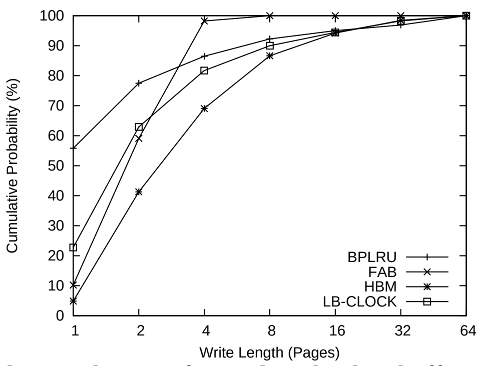
(a) Response time varies with buffer size



(b) Buffer hit ratio varies with buffer size



(c) Number of erases varies with buffer size



(d) Distribution of write length when buffer is 1MB

Figure 4.1: Result of Financial Trace

With the buffer size of 1MB, the average response time of HBM is 0.12 milliseconds (ms). By contrast, the average response time of BPLRU is 0.75 ms (see figure 4.1(a)). HBM makes 84% improvement in terms of average response time compared to BPLRU. Accordingly, there is a 185% hit ratio increase (see figure 4.1(b)) and an 85% erase reduction (see figure 4.1(c)) compared to BPLRU. HBM also exhibits 76% faster, 146% more buffer hits and 82% lower erases compared to FAB for a 1MB buffer. LB-CLOCK also exhibits better performance than BPLRU and FAB. However, HBM still has 50% improvement in term of average response time due to its higher buffer hit ratio and less number of erases than LB-CLOCK.

Figure 4.1(d) shows that the percentage of 1-page write of BPLRU, FAB and LB-CLOCK is 56%, 10% and 23% respectively. By contrast, HBM only has 5% small writes, better than BPLRU, FAB and LB-CLOCK. Furthermore, HBM provides much more large writes than others. For example, almost 32% of the writes are larger than 4 pages in size for HBM, while BPLRU, FAB and LB-

CLOCK only have 14%, 2% and 18% writes larger than 4 pages. So HBM is very efficient in increasing sequentiality of write accesses.

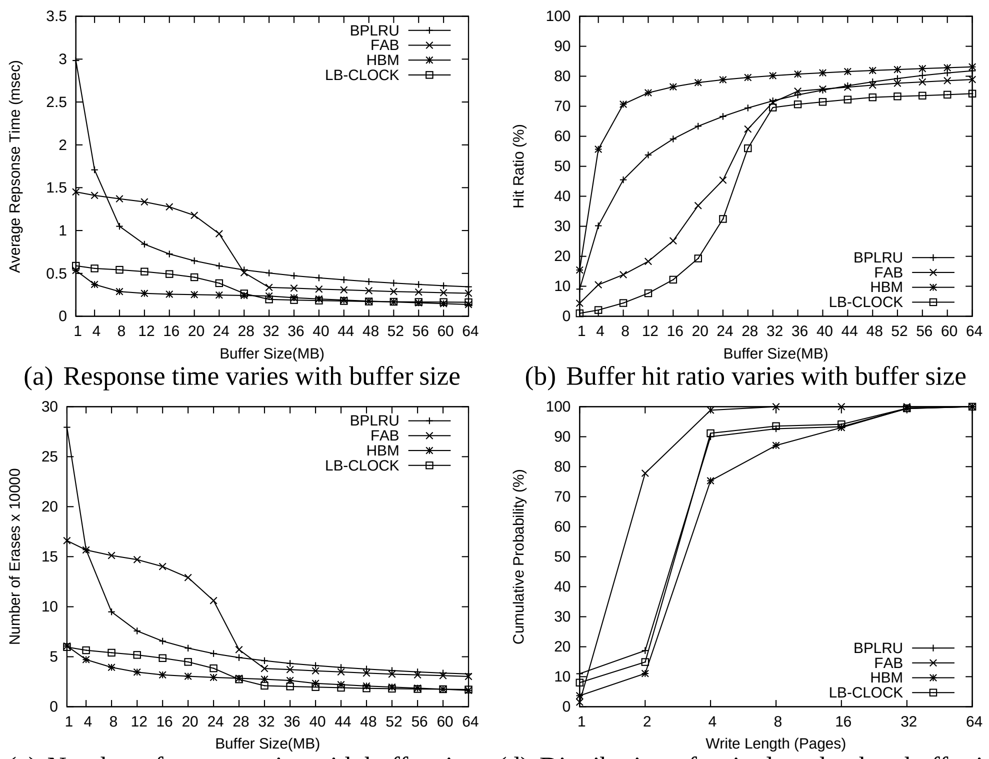
In addition, as shown in figure 4.1(a), not only the average response time in the case of only 1M buffer size is small, but also the performance of HBM is more stable than other three algorithms as buffer size varies. Because RAM inside SSD is an expensive part and consumes more energy if its size is large, a small and efficient buffer is necessary. The stability of performance exhibited by HBM just satisfies this requirement.

The results indicate that the performance gain of HBM comes from two aspects: high buffer hit ratio and reduced garbage collection or less number of erases. This is because HBM exploits page and block to mange buffer space in a hybrid way, taking both temporal and spatial localities into account. HBM 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. This workload exhibits a very high degree of both spatial and temporal locality.

Figure 4.2 shows that HBM exhibits up to 82% faster, 39% more buffer hits and 78% lower erases compared to BPLRU the buffer size up to 32MB. HBM also performs up to 63%, 63% and 308% better in terms of average response time, buffer hit ratio and number of erases compared to FAB. When buffer size is below 32MB, HBM has smaller average response time than LB-CLOCK. Especially when buffer size is 1MB, HBM has a little advantage in average response time due to its much higher buffer hit ratio than LB-CLOCK although the number of erases of LB-CLOCK is a little less (1%) than HBM. Beyond 32MB, the advantage of HBM over BPLRU and FAB narrows down because buffer is large enough to accommodate most accesses. It is thus obvious to see



(a) Response time varies with buffer size (b) Buffer hit ratio varies with buffer size

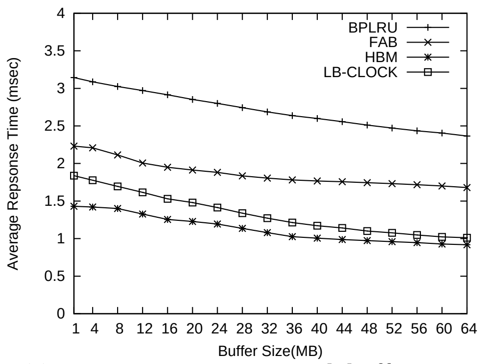
(c) Number of erases varies with buffer size (d) Distribution of write length when buffer is 1MB

Figure 4.2: Result of MSNFS Trace

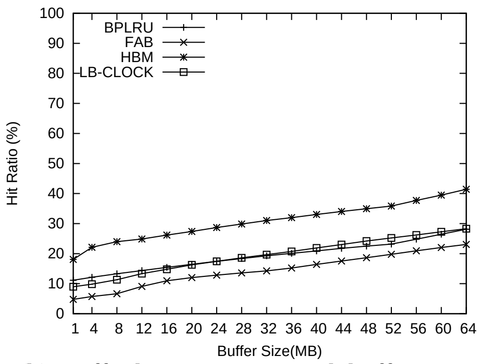
that HBM is better than BPLRU and FAB for workloads of this nature. This is because block-based BPLRU and FAB tradeoff spatial locality with temporal locality, while HBM can efficiently leverage both temporal and spatial localities. Thanks to the less number of erases of LB-CLOCK, its average response time is almost the same as HBM when buffer size is beyond 32MB. However, the buffer hit ratio of LB-CLOCK is lowest among four algorithms, which has a negative effect on its performance. Small sized writes in HBM are less than other algorithms. As shown in figure 4.2(d), 87% writes in HBM are smaller than 8 pages, while more than 95% is the case for others.

3) Exchange Trace

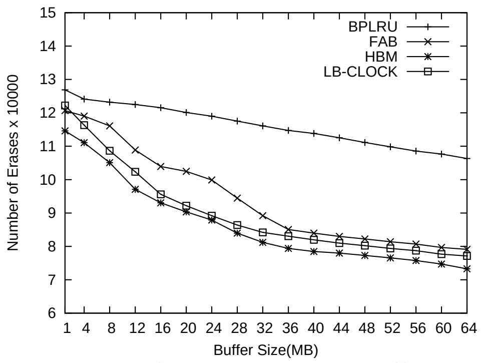
Exchange trace is a random workload in which writes are about 44% more than reads. For a buffer size of 1MB, the average response time of HBM is 1.73ms. By contrast, the average response time of BPLRU is 3.14ms (see figure 4.3(a)). HBM makes 45% improvement in terms of average response time compared to BPLRU. Accordingly, there is a 62% hit ratio increase (see figure 4.3(b)) and a



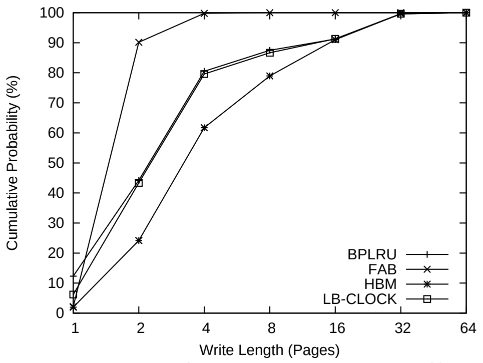
(a) Response time varies with buffer size



(b) Buffer hit ratio varies with buffer size



(c) Number of erases varies with buffer size

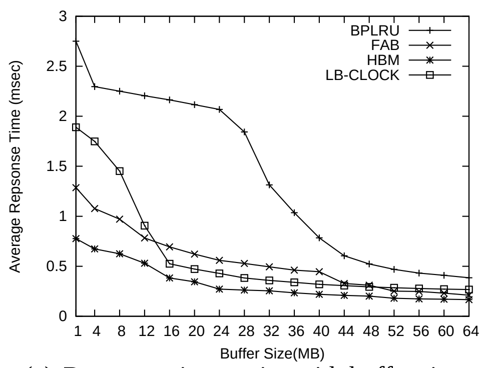


(d) Distribution of write length when buffer is 1MB

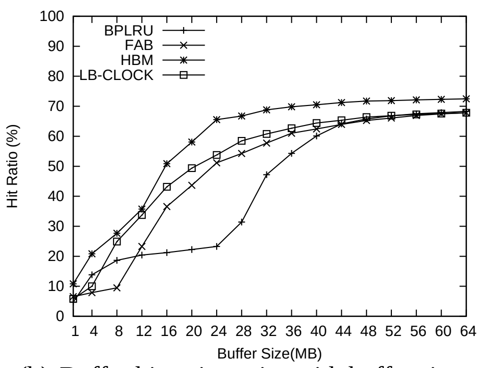
Figure 4.3: Result of Exchange Trace

10% erase reduction (see figure 4.3(c)). HBM also exhibits 23% faster, 280% more buffer hits and 5% lower erases than FAB for a 1 MB buffer. Compared to LB-CLOCK, the performance of HBM is still better. The percentage of small writes (less than 2 pages) of BPLRU, FAB and LB-CLOCK is 44%, 90% and 43%, respectively. By contrast, HBM has 24% small writes, better than BPLRU, FAB and LB-CLOCK (see figure 4.3(d)). Furthermore, HBM provides much more sequential writes than BPLRU and FAB. For example, almost 38% of the writes are larger than 4 pages in size for HBM, while BPLRU, FAB and LB-CLOCK only have 20%, 0.2% and 21% writes larger than 4 pages.

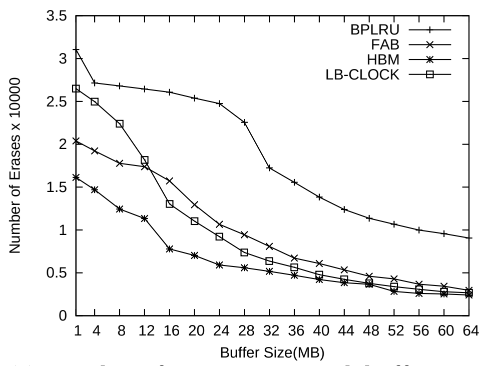
It is worth noting that the number of erases is the most important factor which finally influences the average response time for write dominant workloads such as Exchange trace. For example, as shown in figure 4.3(a)) and 4.3(c), if there is less number of erases for some algorithm, this algorithm shows faster average response time. On the other hand, buffer hit ratio has less influence, especially when its value is not high enough and large number of requests access flash



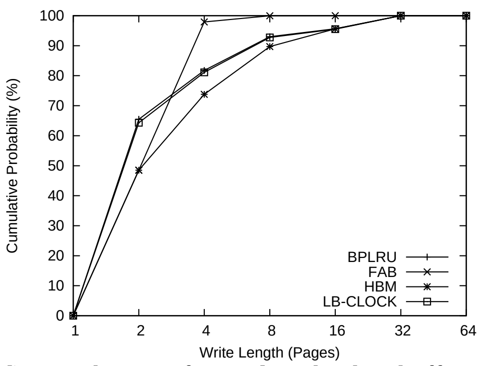
(a) Response time varies with buffer size



(b) Buffer hit ratio varies with buffer size



(c) Number of erases varies with buffer size



(d) Distribution of write length when buffer is 1MB

Figure 4.4: Result of CAMWEBDEV Trace

memory directly. As shown in figure 4.3(b), in spite of a little higher buffer hit ratio of BPLRU than FAB, the average response time of BPLRU is larger than FAB because of its much larger number of erases so that it suffers heavy garbage collection overload.

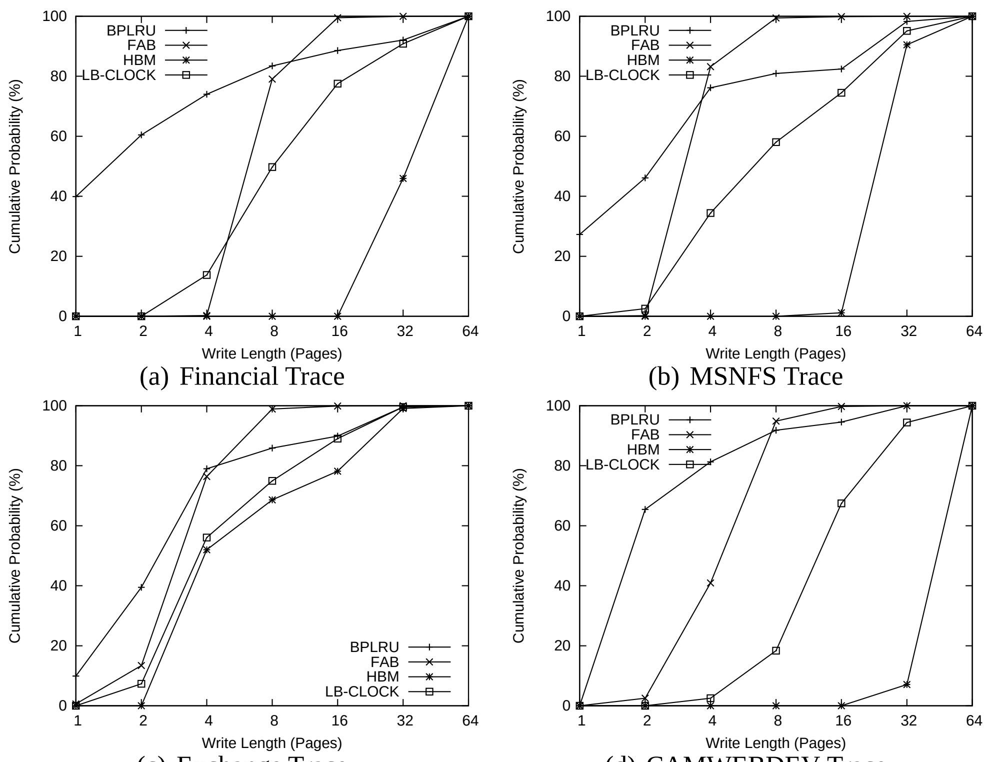
4) CAMWEBDEV Trace

CAMWEBDEV trace is a completely random write dominant workload. Figure 4.4 shows that HBM performs 72%, 40% and 61% faster than BPLRU, FAB and LB-CLOCK for the buffer size of 1 MB. Accordingly, there is a 128%, 65% and 18% buffer hit ratio increase compared to BPLRU, FAB and BPLRU. There is also a 48%, 21% and 39% reduction of erases.

We observe that HBM is also efficient in reducing the number of small writes and increasing the number of sequential writes for write intensive workload. Figure 4.4(d)) shows that BPLRU, FAB and LB-CLOCK produce 82%, 98% and 82% small writes (less than 4 pages). By contrast, HBM has 74% small write, which is better than others. HBM also provides 10% large writes, which are larger than 8 pages in size. However, BPLRU, FAB and LB-CLOCK only have 7%, 0.02% and 7% large writes.

Figure 4.5 also shows the distribution of write length for four algorithms with buffer size of 16MB under four traces. Take figure 4.5(d) as an example, with buffer size of 16MB, BPLRU, FAB and LB-CLOCK produce 81%, 40% and 2% small writes (less than 4 pages) under CAMWEBDEV trace. By contrast, HBM has not such small writes. HBM provides 93% large writes, which is larger than 32 pages in size. However, BPLRU, FAB have 8% and 5% large writes, which is larger than 8 pages in size. LB-CLOCK shows better performance in large writes, and it has 82% more than 8 pages writes. Compared figure 4.5(d) with figure 4.4(d), as buffer size increases from 1MB to 16MB, improvement of HBM is much larger than others. This indicates that HBM algorithm is more efficient than BPLRU, FAB and LB-CLOCK in increasing sequentiality of write accesses across different buffer sizes.

The results further show that the distribution of write length is directly correlated to the garbage collection overhead and performances. With buffer size of 1MB, HBM is able to produce 27% writes whose size are larger than 4 pages compared to 18%, 2% and 19% for BPLRU, FAB and LB-CLOCK (see figure 4.4(d)). Accordingly, there is a 48%, 21% and 39% garbage collection overhead reduction for HBM compared to BPLRU, FAB and LB-CLOCK (see figure 4.4(c)). Consequently, performance is improved by 72%, 40% and 59% compared to BPLRU, FAB and LB-CLOCK (see figure 4.4(a)). For other traces, we can also conclude this correlation by comparing other figures in figure 4.5 with the corresponding ones in figure 4.1, 4.2, and 4.3. The correlation clearly indicates that write length is a critical factor affecting SSD performance and garbage collection overhead.



(a) Financial Trace (b) MSNFS Trace

(c) Exchange Trace (d) CAMWEBDEV Trace

Figure 4.5: Distribution of write length of BPLRU, FAB, HBM and LB-CLOCK when buffer size is 16MB

4.3.2 Effect of Workloads

We observe that efficiency of HBM is different under different workload traces. With buffer size of 1MB, HBM achieves 84%, 82%, 45% and 72% performance improvement over BPLRU for Financial, MSNFS, Exchange and CAMWEB-DEV traces, respectively. There is a 76%, 63%, 22% and 40% performance improvement compared to FAB. Accordingly, HBM makes 50%, 10%, 22%, 61% performance improvement than LB-CLOCK. The results indicate that HBM outperforms BPLRU, FAB and LB-CLOCK for different types of random workloads in an enterprise system.

For sequential write dominant trace, we generate a trace called synthetic trace using the internal trace generator of DiskSim 4.0 [6]. Synthetic workload specification in DiskSim 4.0 is set as shown in Table 4.3, the corresponding interpretations of parameters can be found in [6].

We show the results in the figure 4.6. We can see that HBM still perform bet-

Table 4.3: Synthetic workload specification in Disksim_Synthgen

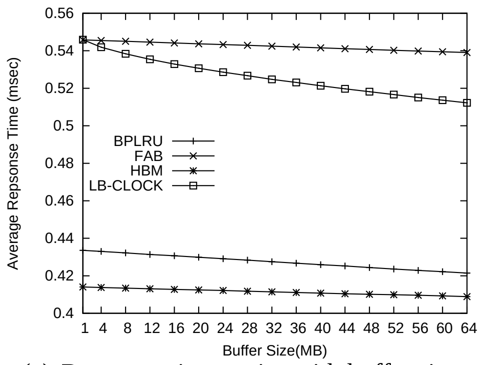
Storage capacity of device =32GB;
Number of I/O requests to generate = 5000000;
Block factor = 2KB;
Probability of sequential access = 0.8;
Probability of local access = 0.2;
Probability of read access = 0.3;
General inter-arrival times = [normal probability-distribution, 52, 52];
Local distances = [normal probability-distribution, -100000, 100000];
Sizes = [exponential probability-distribution, 0, 20];

ter than BPLRU, FAB and LB-CLOCK, but the advantage of HBM over others especially BPLRU is not so significant because this workload provides more spatial locality for them to exploit, compared to random workloads above. In addition, all the four algorithms exhibit stable performance as buffer size varies because most of write requests are sequential which are in favor with flash memory to reduce the garbage collection overhead. As shown in figure 4.6(a) and 4.6(c), the conclusion that number of erases are the important factor of performance in write dominant traces is verified again. The trend of number of erases as buffer size varies is almost the same as the trend of average response time, especially for LB-CLOCK. From figure 4.6(d), we can see that the write sizes for all the four algorithms are larger compared to random workloads. All the write sizes for HBM are nearly 64 pages or a full block.

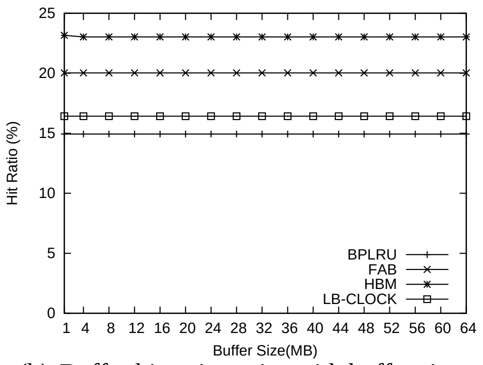
4.3.3 Additional Overhead

To study the overhead of different buffer management schemes under different workloads, we present total read pages during replaying traces in figure 4.7. We can see from the results that BPLRU conducts a large number of read operations under different types of random workloads.

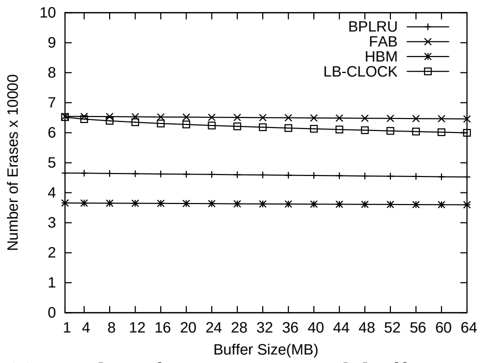
Let's take figure 4.7(a) as an example. With the buffer size of 1MB, BPLRU



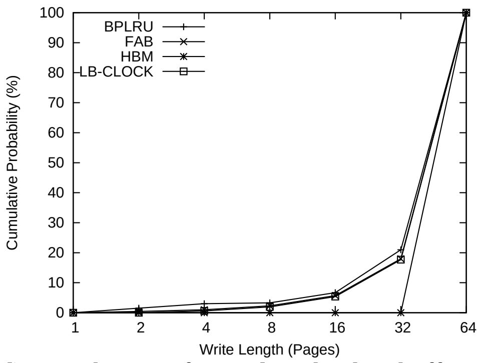
(a) Response time varies with buffer size



(b) Buffer hit ratio varies with buffer size



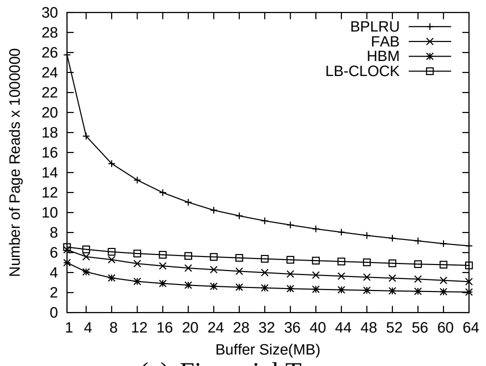
(c) Number of erases varies with buffer size



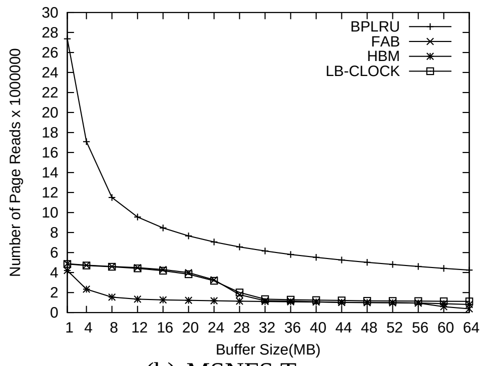
(d) Distribution of write length when buffer is 1MB

Figure 4.6: Result of Synthetic Trace

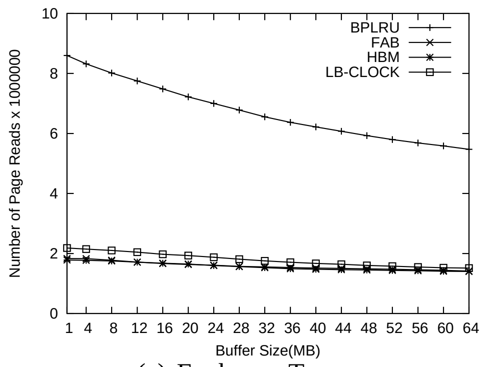
results in 368%, 275% and 294% more page reads than HBM, FAB and LB-CLOCK respectively. Accordingly, the average response time of BPLRU is 523%, 48% and 225% slower than HBM, FAB and LB-CLOCK (see figure 4.1(a)) respectively. This is because that BPLRU uses page padding to improve the number of sequential writes. For completely random workload in enterprise environment, BPLRU needs to read a large number of additional pages, which impacts the overall performance. Additionally, BPLRU is just a writing buffer algorithm, which does not consider read operations as HBM so that page reads increase due to read misses. As shown in figure 4.7(a), the total number of reads for LB-CLOCK is also larger than FAB and HBM but smaller than BPLRU, because LB-CLOCK does not apply the page padding technique, however, it is also a writing buffer algorithm as BPLRU. Although the cost of read operation inside SSD is cheaper than write and erase operation, quite large number of extra reads can obviously impact the overall performance especially for read dominant workload such as Financial trace. For other three traces than Finan-



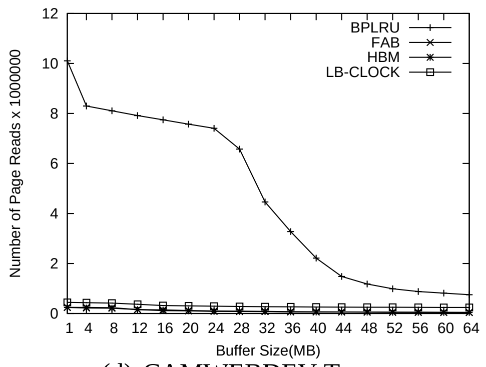
(a) Financial Trace



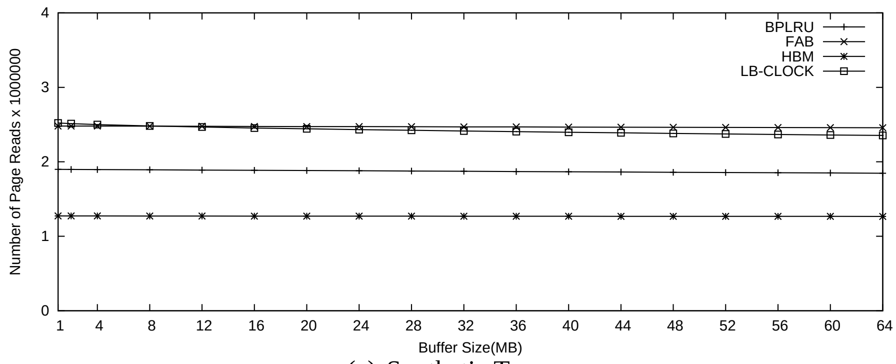
(b) MSNFS Trace



(c) Exchange Trace



(d) CAMWEBDEV Trace



(e) Synthetic Trace

Figure 4.7: Total page reads by BPLRU, FAB, HBM and LB-CLOCK under 5 traces

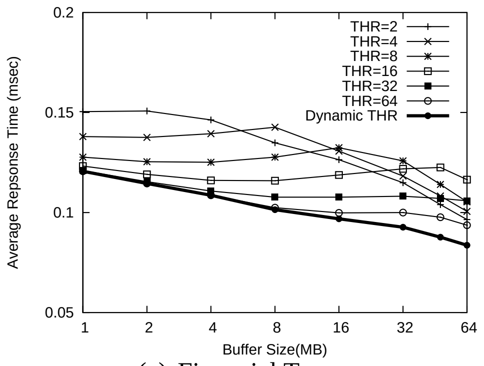
cial trace, the extra reads for LB-CLOCK are reduced as read requests in the workload decreases, as shown in figure 4.7(b), 4.7(c) and 4.7(d).

By contrast, our proposed HBM achieves better performance without additional reads. HBM treats read and write as a whole and leverages the block level temporal locality among read and write accesses to naturally form sequential block.

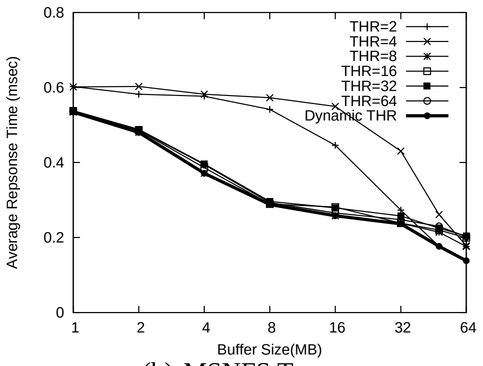
4.3.4 Effect of Threshold

To investigate how threshold value affects the efficiency of proposed HBM, we test HBM with static thresholds and dynamic threshold for different traces, as shown in figure 4.8. For dynamic threshold, we set the value of α and β in constraint 3.1 based on the rules in section 3.6 as 10% for small buffer size such as 1MB, 2MB, 4MB and 8MB and 20% for large buffer size such as 16MB, 32MB, 48MB and 64MB. Let's take figure 4.8(c) as an example. With the buffer size of 16MB, the average response time of HBM is 1.79ms, 1.65ms and 1.73ms when threshold value is 2, 4 and 8 respectively. By contrast, the average response time of HBM is 1.55ms for dynamic threshold, which is much better than that of static thresholds. As shown in figure 4.8(e), when the workload is sequential, HBM for dynamic threshold also shows better performance than static thresholds under various buffer sizes, because the dynamic scheme can make the best use of page region in order to increase the buffer hit ratio.

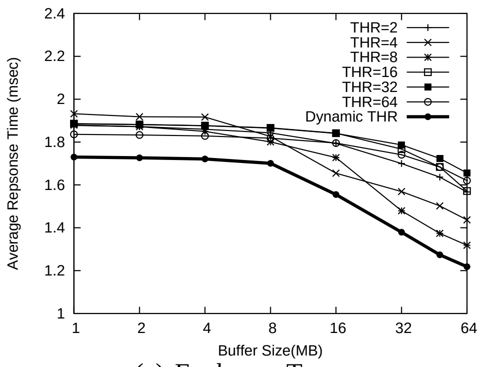
We further observe that the same threshold is also unable to achieve optimal performance for different workloads. Figure 4.8(a) shows that with buffer size of 16MB, HBM performs better when threshold is set as 64 for Financial trace, compared to other static thresholds. However, with buffer size of 16MB and threshold of 64, the average response time of HBM is 0.67ms for CAMWEB-DEV trace, which is worse compared to threshold of 2, 4, and 8 respectively (see figure 4.8(d)). By contrast, the results in figure 4.8 show that dynamic threshold



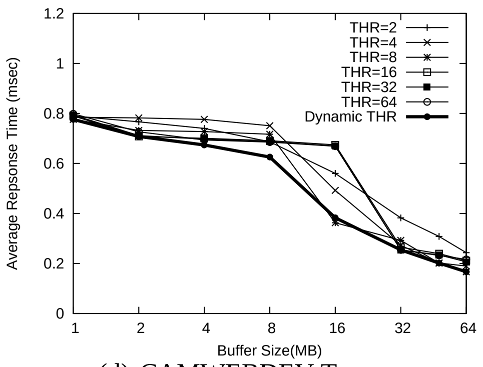
(a) Financial Trace



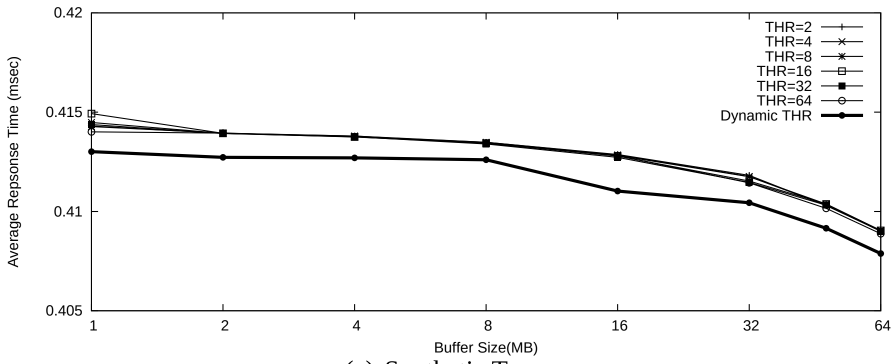
(b) MSNFS Trace



(c) Exchange Trace



(d) CAMWEBDEV Trace



(e) Synthetic Trace

Figure 4.8: Effect of thresholds on HBM

achieves best performance for Financial, MSNFS, Exchange, CAMWEBDEV and Synthetic traces respectively.

The variation in performance curves shown in the figure 4.8 clearly indicates that threshold value has significant impact on efficiency of our proposed HBM. Statically setting threshold is unable to achieve optimal performance. Dynamically adjusting the threshold for enterprise workloads makes proposed HBM workload adaptive.

4.3.5 Energy Consumption of Flash Chips

Although flash chips consume less energy than hard disks due to electronic characteristic, the energy could not be ignored when they are applied in enterprise datacenter where extensive write or read accesses happen. Especially, garbage collection inside SSD which takes more time also consumes more energy than normal read or write operations as shown in Table 4.4 that indicates energy consumption of operations inside SSD. In addition, more garbage collection overhead could result in longer total runtime of finishing a workload, in that case, buffer could experience longer idle time which is another part energy consumption, and in the paper [16], it is pointed out that the processor consumption can be highly correlated with garbage collection. In this thesis, we mostly focus on the performance improvement issue of SSD. As for energy saving issue, we only discuss energy consumption of flash chips, and analysis on energy consumption of other parts will be done in future work.

Read, write and erase operations contribute most energy consumption of flash chips. Reads and writes here also include the extra read/writes during garbage collection.

Suppose $N_{flashread}$, $N_{flashwrite}$, $N_{flasherase}$ are the number of corresponding operations needed for some workload, which can be measured in the simulator, so the total energy consumption of flash chips for processing this workload is

Table 4.4: Energy consumption of operations inside SSD. (The parameters are obtained from [38])

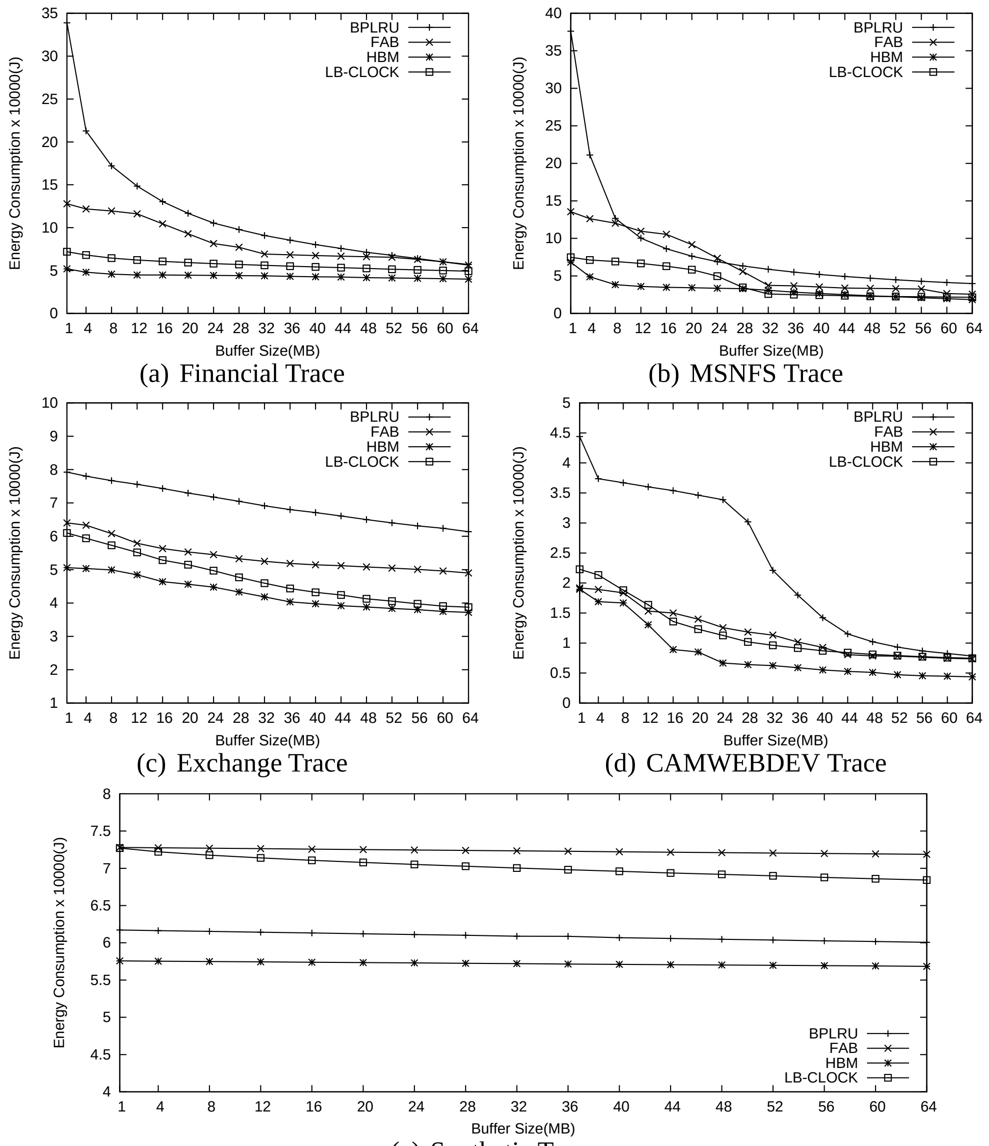
Type of Operation	**Energy (mJ)**	**Description**
$Energy_{flashread}$	0.0020625	Energy consumption for reading one page of flash chips
$Energy_{flashwrite}$	0.0165	Energy consumption for writing one page of flash chips
$Energy_{flasherase}$	0.12375	Energy consumption for erasing one block of flash chips

generally calculated based on the equation 4.1 below:

$$\begin{aligned} Energy_{total} \quad = \quad & N_{flashread} * Energy_{flashread} + N_{flashwrite} * Energy_{flashwrite} \\ & + N_{flasherase} * Energy_{flasherase} \end{aligned} \tag{4.1}$$

According to equation 4.1, figure 4.9 shows the energy consumption of flash chips under BPLRU, FAB, HBM, and LB-CLOCK during processing five traces. We can see that HBM consumes less energy than other four algorithms under all the five traces. Taking the corresponding number of erases of four algorithms (see figure 4.1(c), 4.2(c), 4.3(c), 4.4(c) and 4.6(c)) for reference, we are sure that the number of erases could be also an indicator of energy consumption of flash chips, because it consumes the most energy of three operations as shown in Table 4.4. HBM suffers from less overhead of garbage collection or number of erases under these traces thanks to its hybrid management which reduces the number of writes by increasing buffer hit ratio in page region, meanwhile, increases the sequentiality of data passed to flash memory in block region.

It is worth noting that BPLRU is not an energy saving approach shown in figure 4.9(a), 4.9(b), 4.9(c) and 4.9(d), because these four traces include a large number small sized requests, which incur so many extra reads during page padding of BPLRU. In spite of a little energy cost of reading flash chips (see table 4.4, only 0.0020625mJ), they contribute a lot when the number of reads is much more than other operations as shown in figure 4.7 in section 4.3.3. On the contrary, sequential requests are dominant in Synthetic trace, so extra reads are reduced



(a) Financial Trace (b) MSNFS Trace

(c) Exchange Trace (d) CAMWEBDEV Trace

(e) Synthetic Trace

Figure 4.9: Energy consumption of BPLRU, FAB, HBM and LB-CLOCK under five traces

during page padding, the number of erases take the leading role in consuming energy (see figure 4.6(c), 4.7(e) and 4.9(e)).

Chapter 5

Conclusion

In this thesis, we propose a hybrid buffer management scheme called HBM, which divides the buffer space into page region and block region to make full use of both temporal and spatial localities among accesses. HBM adopts buffer hit ratio and sequentiality of writes as design objectives. To achieve both objectives, Random access pages are staying in the page region in priority, while sequential access pages in the block region are replaced first. Leveraging hybrid management and dynamic migration, HBM can efficiently adjust the size of page region and block region based on different workloads. Our experiments conclusively demonstrate that HBM improves the performance of SSD, by significantly reducing the internal fragmentation and garbage collection overhead associated with random write, meanwhile, the energy consumption of flash chips for HBM is limited.

In our immediate future work, we want to investigate how to model the dynamic migration and calculate the value of threshold based on the model in future. The model is also expected to dynamically allocate more reasonable buffer size to block region based on workload characteristic. In addition, we currently design HBM as buffer for both read and write accesses and assume that volatile RAM based buffer is used. Therefore, we will have to address other issues of RAM, such as data loss when power is switched off abnormally. Then, we will implement and evaluate our proposed HBM on a real prototype based on some flash memory attached electronic board. All the research work that has been done by us targets at the buffer inside SSD, and we also want to implement our proposed HBM in Linux kernel to enable system investigation. In other words, we will do some research about applying HBM in the buffer of host system to study the performance of HBM at system level.

Bibliography

[1] B+ Tree. `http://en.wikipedia.org/wiki/B_tree`.

[2] Block traces from SNIA. `http://iotta.snia.org/traces`.

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

[4] Understanding the flash translation layer (ftl) specification. Intel Corporation, 1998.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[6] J.S. Bucy, J. Schindler, S.W. Schlosser, and G.R. Ganger. The DiskSim simulation environment version 4.0 reference manual. Technical report, Technical Report CMU-PDL-08-101, Carnegie Mellon University, 2008.

[7] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1126–1130, New York, NY, USA, 2007. ACM.

[8] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, pages 181–192, New York, NY, USA, 2009. ACM.

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

[10] Thomas Claburn. Google Plans To Use Intel SSD Storage In Servers. `http://www.informationweek.com/news/storage/systems`, 2008.

[11] F.J. Corbato. A paging experiment with the Multics system, 1968.

[12] Biplob Debnath, David Du, and David Lilja. Large Block CLOCK (LB-CLOCK): A write caching algorithm for solid state disks. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*. ACM, 2009.

[13] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, pages 279–289. ACM, 2009.

[14] F. Douglis, R. Caceres, M. Kaashoek, P. Krishnan, K. Li, B. Marsh, and J. Tauber. Storage alternatives for mobile computers. *Mobile Computing*, pages 473–505, 1996.

[15] G.R. Ganger and M.F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Technical Conference*, pages 1–18, 1997.

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

[17] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 8–8, 2005.

[18] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[19] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, 1994.

[20] J.U. Kang, H. Jo, J.S. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, page 170. ACM, 2006.

[21] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 58(6):744–758, 2009.

[22] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.

[23] Hyojun Kim and Umakishore Ramachandran. FlashLite: A User-Level Library to Enhance Durability of SSD for P2P File Sharing. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 534–541, 2009.

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

[25] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *SIMUL '09: Proceedings of the 2009 First International Conference on Advances in System Simulation*, 2009.

[26] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS*, 2008.

[27] S.W. Lee, D.J. Park, T.S. Chung, D.H. Lee, S. Park, and H.J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[28] Zhanzhan Liu, Lihua Yue, Peng Wei, Peiquan Jin, and Xiaoyan Xiang. An adaptive block-set based management for large-scale flash memory. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1621–1625, New York, NY, USA, 2009. ACM.

[29] Lucas Mearian. MySpace replaces all server hard disks with flash drives. `http://www.computerworld.com/s/article/9139280`, 2009.

[30] N. Megiddo and D.S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, 2003.

[31] A. Rajimwale, V. Prabhakaran, and J.D. Davis. Block Management in Solid-State Devices. *Proceedings of the USENIX Annual Technical Conference (USENIX'09)*, 2009.

[32] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, Inc. New York, NY, USA, 1999.

[33] D. Reinsel and J. Janukowicz. White Paper: Datacenter SSDs: Solid Footing for Growth. `http://www.samsung.com/global`, 2008.

[34] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

[35] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 338–349, New York, NY, USA, 2009. ACM.

[36] A. Tal. Two technologies compared: NOR vs. NAND White Paper. *M-Systems Inc*, 2003.

[37] G. Wu, B. Eckart, and X. He. BPAC: An adaptive write buffer management scheme for flash-based Solid State Drives. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[38] D. Feng L. Tian S. Zhang J. Liu W. Tong. Y. Hu, H. Jiang. Achieving Page-Mapping FTL Performance at Block-Mapping FTL Cost by Hiding Address Translation. In *26th IEEE (MSST 2010) Symposium on Massive Storage Systems and Technologies*, 2010.