WAFTL: A Workload Adaptive Flash Translation Layer with Data Partition

Qingsong Wei[†], Bozhao Gong[€], Suraj Pathak[€], Bharadwaj Veeravalli[€], LingFang Zeng[€], Kanzo Okada[†]

[†]Data Storage Institute, A*STAR, Singapore
{WEI_Qingsong, Kanzo_OKADA}@dsi.a-star.edu.sg

[€]National University of Singapore
{gbozhao, suraj, elebv, lfzeng}@nus.edu.sg

Abstract—**Current FTL schemes have inevitable limitations in terms of memory requirement, performance, garbage collection overhead, and scalability. To overcome these limitations, we propose a workload adaptive flash translation layer referred to as WAFTL. WAFTL explores either page-level or block-level address mapping for normal data block based on access patterns. Page Mapping Block (PMB) is used to store random data and handle large number of partial updates. Block Mapping Block (BMB) is utilized to store sequential data and lower overall mapping table. PMB or BMB is allocated on demand and the number of PMB or BMB eventually depends on workload. An efficient address mapping is designed to reduce overall mapping table and quickly conduct address translation. WAFTL explores a small part of flash space as Buffer Zone to log writes sequentially and migrate data into BMB or PMB based on threshold. Static and dynamic threshold setting are proposed to balance performance and mapping table size.**

WAFTL has been extensively evaluated under various enterprise workloads. Benchmark results conclusively demonstrate that proposed WAFTL is workload adaptive and achieves up to 80% performance improvement, 83% garbage collection overhead reduction and 50% mapping table reduction compared to existing FTL schemes.

Keywords-Flash Memory; Solid State Drive; Flash Translation Layer; Data Partition

I. Introduction

Flash memory is rapidly becoming a promising technology for next-generation storage and receives strong interest in both academia and industry [6,7,12,14]. Flash memory has been traditionally used in portable devices. More recently, as price drops and capacity increases, this technology has made huge strides into enterprise storage space in the form of Solid State Drive (SSD).

However, SSD suffers from random writes when applied in enterprise environment. Firstly, random writes is much slower than sequential writes. Secondly, NAND flash memory can incur only a finite number of erases for a given physical block. Therefore, increased erase operations due to random writes shorten the lifetime of SSD. Finally, random writes result in higher overhead of internal garbage collection than sequential writes. If the incoming writes are randomly distributed over the logical address space, sooner or later all physical blocks will be fragmented which results in large number of page copies during garbage collection. Therefore, random write is a critical problem to both performance and lifetime, which restricts SSDs' widespread acceptance in data centers [7].

In current practice, there are several major efforts concerning the issues of random writes at various levels [1,2,3,12,14]. One effort is to reduce negative impacts of random writes through efficient Flash Translation Layer (FTL). FTL emulates the functionality of a block device and enables operating system to use flash memory without any modification. However, internally FTL needs to deal with erase-before-write, which makes it critical to overall performance and lifetime of SSD.

FTL schemes can be classified into three groups depending on the granularity of address mapping: page-level FTL, block-level FTL, and log-buffer based hybrid FTL [8,9,15]. Page-level FTL is a fine grain mapping scheme and efficient in handling random writes. This mapping scheme achieves high performance, high space usage and low garbage collection overhead, but it requires very large memory space to store the entire page mapping table. Block-level FTL significantly lowers memory requirement by maintaining coarse grain block level mapping. However, an update to a page in a block level mapping will trigger the erasure of the block containing the corresponding page. Thus, performance of block-level FTL is much worse than that of page-level FTL under random workload.

In order to take advantage of the both page-level and block-level mapping, various log-buffer based hybrid FTL scheme have been proposed [11,13,15]. Most of these schemes are fundamentally based on a block-level mapping with an additional page-level mapping restricted only to a small number of log blocks. Both sequential and random data in the log blocks are replaced into data blocks without differentiation. If the data in data blocks is updated randomly in future, large number of expensive merge operations will be triggered. Hybrid schemes suffer from performance degradation under random workload due to excessive full merges that are caused by the difference in mapping granularity of data and log blocks. Recently, a page level mapping scheme called DFTL (Demand-based Flash Translation Layer) was proposed to reduce memory requirement and lookup overhead by loading partial of mapping table into SRAM [10]. However, its efficiency is heavily depended on temporal locality of workload. DFTL suffers from frequent replacements and updates in the pages storing the page mapping table in case of write intensive workloads.

Most FTLs use single granularity of mapping for normal data blocks without differentiating data. This paper presents a workload adaptive flash translation layer referred to as WAFTL. WAFTL manages sequential and random data in different granularity. Page Mapping Block (PMB) is used to store random data and handle large number of partial updates. Block Mapping Block (BMB) is explored to store sequential data and lower overall mapping table. PMB or BMB is allocated on demand and thus, the number of PMB or BMB eventually depends on workload. WAFTL uses an efficient address mapping to reduce overall mapping table and quickly translate logical address to physical address of a BMB or PMB. WAFTL explores a small part of flash space as Buffer Zone to log all the writes in sequential manner. Data in the Buffer Zone is migrated into BMB or PMB based on threshold. Static and dynamic threshold setting are proposed to balance performance and mapping table size. WAFTL has been extensively evaluated under various enterprise workloads. Benchmark results conclusively demonstrate that proposed WAFTL is workload adaptive, efficient with reduced mapping table and garbage collection.

The rest of this paper is organized as follows. Section II provides an overview of background and motivation. We present design details of WAFTL in Section III. Section IV gives evaluation and measurement results. Related work is presented in Section V. We summarize this paper with the conclusions and possible future work in Section VI.

II. Background and Motivation

In this section, we present basic background and motivation that are essential to our work.

A. Flash Memory

In this paper, flash memory refers to NAND flash memory specifically. NAND flash memory can be classified into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. A SLC flash memory cell stores only one bit, while a MLC flash memory cell can store two bits or even more. For both SLC and MLC NAND, a flash memory package is composed of one or more dies (chips). Each die within a package contains multiple planes. A typical plane consists of thousands (e.g. 2048) of blocks and one or two registers of the page size as an I/O buffer. Each block in turn consists of 64 to 128 pages. Each page has a 2KB or 4KB data area and a metadata area (e.g. 128 bytes) for storing identification, page state and Error Correcting Code (ECC) information [6].

Flash memory supports three major operations, read, write, and erase. Read and write are performed in units of pages. Flash blocks must be erased before they can be reused and erase operation must be conducted in block granularity. In addition, each block can be erased only a finite number of times. A typical MLC flash memory has around 10,000-30,000 erase cycles, while a SLC flash memory has around 100,000-300,000 erase cycles.

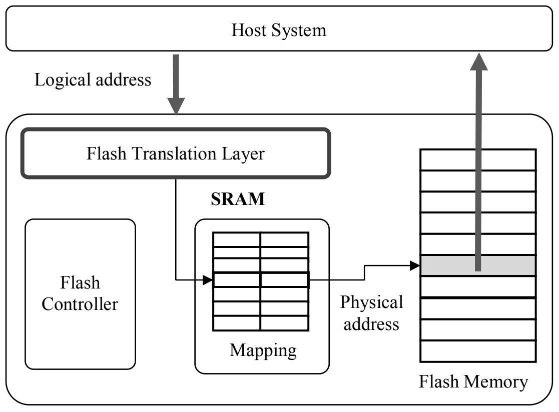


Figure 1. NAND flash SSD system architecture

B. Flash Translation Layer

FTL is a critical firmware implemented in SSD and plays a key role in providing address mapping, wear leveling and garbage collection (See Figure 1).

Address Mapping – Since data in flash memory cannot be updated in place, the FTL simply writes the data to another clean page and marks the previous page as invalid. Therefore, address mapping table is required to translate host logical address to physical flash address.

FTL schemes can be classified into page-level, block-level, and hybrid FTLs. Page-level FTL scheme translates a logical page number (LPN) to a physical page number (PPN) in flash memory. This mapping approach is efficient and shows great garbage collection efficiency, but it requires a large amount of RAM space to store the mapping table. Block-level FTL significantly lowers memory requirement by translating logical block number (LBN) to physical block number (PBN). However, it requires an expensive read-modify-write operation when writing only part of a block.

Hybrid FTL is a compromise between page-level and block-level mapping. In this scheme, a small portion of physical blocks is reserved as log buffer. While the log blocks in the log buffer use page-level mapping scheme, the normal data blocks are handled by the block-level mapping. Hybrid mapping requires a small-sized mapping table since only the log blocks are handled by the page-level mapping [15]. Log block is associated with one or multiple data blocks. The pages in a log block might originate from several different data blocks. When there is no empty space in the log buffer, one of the log block is selected as victim to make space for on-going write requests and all of the valid pages in the log block are copied into the corresponding data blocks by merging operations. With randomness of accesses increase, data distribution among pages of a log block will be more random, which results in large number of full merges. From this point of view, the problems of expensive real-time merges are inherent to log-buffer based hybrid FTL.

Garbage Collection – When running out of clean blocks, a garbage collection module scans flash memory blocks and recycles invalidated pages. If a page-level mapping is used, the valid pages in the scanned block are copied out and condensed into a new block. For block-level and hybrid-level mappings, the valid pages need to be merged together with the updated pages in the same block. Efficiency of FTL highly depends on overhead of garbage collection.

Wear Leveling – Due to the locality in most workloads, writes are often performed over a subset of blocks. Thus some flash blocks may be frequently overwritten and tend to wear out earlier than other blocks. FTLs usually employ wear leveling algorithm to ensure that equal use is made of all the available write cycles for each block [6].

C. *Motivation*

While the capacity of flash memory keeps increasing significantly, efficient management of flash memory space is desirable to better utilize the potential of flash in enterprise-scale environments. The internal idiosyncrasies of flash technology make its performance highly dependent on workload characteristics.

TABLE I. COMPARISON OF FTL SCHEMES

Metrics	Pure Page-level FTL	DFTL	Block-level FTL	Hybrid FTL
Performance	High	High	Low	Middle
Garbage collection overhead	Low	Low	High	High
Space Usage	High	High	Low	Middle
Mapping table	Large	Large	Small	Middle
Mapping scheme	Static	Static	Static	Static
Workload Adaptive	No	No	No	No

Different granularities in space management impose different management costs and mapping efficiency. Table I illustrates the features of state-of-art FTL schemes. Coarse grain address mapping lowers memory requirement, which is crucial for cost and power consumption, while fine grain address mapping is efficient in handling random writes. Workload in an enterprise system is a mixture of random and sequential accesses. Partitioning enterprise workload into sequential and random data and managing them by block-level and page-level mapping respectively could significantly improve overall performance, mapping table size and garbage collection efficiency. We are motivated to design a novel FTL by leveraging the two different granularities in address translation for different data.

III. WAFTL: WORKLOAD ADAPTIVE FTL

WAFTL judiciously takes advantage of both page-level and block-level address mapping based on workload pattern so that it can overcome the innate limitations of existing FTL schemes.

A. *Logical Layout*

Flash space is divided into data blocks, mapping table blocks and buffer zone. Data block is used to permanently store user data, and mapping table block is to store mapping information in flash, while buffer zone is used to temporally receive incoming writes (See Figure 2).

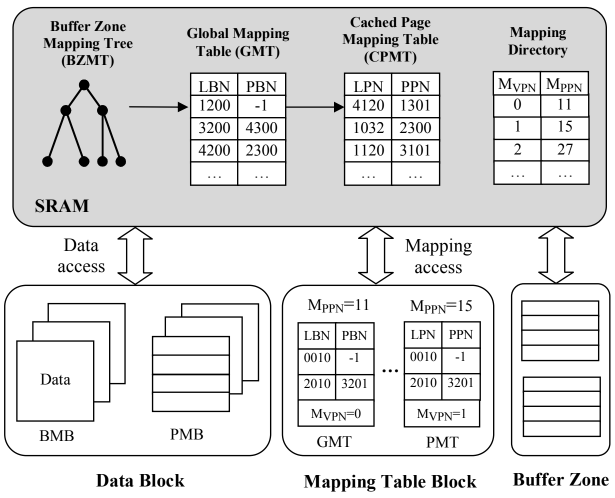


Figure 2. Logical Layout of WAFTL. *LBN*: Logical Block Number, *PBN*: Physical Block Number, *LPN*: Logical Page Number, *PPN*: Physical Page Number, M_{VPN}: Virtual Mapping Page Number, M_{PPN}: Physical Mapping Page Number, *BMB*: Block Mapping Block, *PMB*: Page Mapping Block, *BZMT*: Buffer Zone Mapping Tree, *GMT*: Global Mapping Table, *PMT*: Page Mapping Table, *CPMT*: Cached Page Mapping Table.

Data blocks are further classified into Block-level Mapping Block (BMB) and Page-level Mapping Block (PMB). PMB is used to store random data and handle large number of partial updates, while BMB is to store sequential data and lower overall mapping table. BMB and PMB are dynamically allocated from free block pool. Current FTLs pre-allocate mapping table based on whole flash space and mapping scheme is static. In WAFTL, mapping scheme and mapping entry in mapping table for each data block is determined and initialized only when it is allocated. Therefore, the number of BMB and PMB is determined by workload. Sequential intensive workload has more BMBs, while random intensive workload has more PMBs. This feature enables WAFTL workload adaptive.

Mapping table blocks are managed by page-level mapping. WAFTL explores a Global Mapping Table (GMT) and a Page Mapping Table (PMT) for address translation. The GMT and PMT are stored in the data area of a page instead of OOB area. At runtime, GMT is fully loaded into SRAM, while PMT is partially cached to save memory requirement. LRU is used for replacement of the Cached Page Mapping Table (CPMT). Mapping Directory is used to record the physical page location of GMT and PMT on mapping table block. Address translation process will be discussed in following subsection B.

WAFTL explore a small portion of flash memory as Buffer Zone to log incoming write in sequential way. Buffer Zone Mapping Tree (BZMT) is designed to organize the pages in the Buffer Zone. BZMT is fully loaded into SRAM and functions as access portal, which will be discussed in subsection C.

Note that WAFTL is different from existing hybrid FTLs. Existing hybrid FTL schemes only use block-level mapping for normal data blocks. Both sequential and random data in the log blocks are replaced into data blocks without differentiation. If the data in data blocks are updated randomly in future, large number of expensive merges will be triggered because of the difference in granularity of address mapping between data blocks and log blocks. WAFTL eliminates this difference by dividing data blocks into BMB and PMB. By contrast, WAFTL differentiates data by storing sequential data into BMB and random data into PMB. Partial updates and random writes can be efficiently absorbed by PMB, which significantly reduces overhead of garbage collection.

B. Address Translation Process

A Global Mapping Table (GMT) and a Page Mapping Table (PMT) are explored to translate host logical address to physical flash address. A modified block-level mapping is used for GMT, which maintains mapping from logical block number to physical block number (See Figure 3). WAFTL uses value of physical block number to indicate whether the data is in a BMB or PMB. Positive value of physical block number means that data is in BMB, while -1 means that data is in PMB. Therefore, BMB can be addressed by GMT directly. This design can significantly reduce the size of overall mapping table because only PMB requires additional mapping table.

With logical page number (LPN) of a request, WAFTL firstly calculates corresponding logical block number (LBN) and block offset. For write request, Buffer Zone simply logs it and then updates the Buffer Zone Mapping Tree (BZMT), which will be discussed in following subsection C. For read request, LBN is used to search the BZMT. If corresponding physical page number is found, the requested data will be read from the Buffer Zone.

Otherwise, address translation is forwarded to the GMT. Physical block number (PBN) is obtained by searching the GMT with LBN. If corresponding PBN is positive, request can be directly accessed with PBN and block offset. If corresponding PBN is -1, WAFTL further obtains physical page number (PPN) by searching the Cached Page Mapping

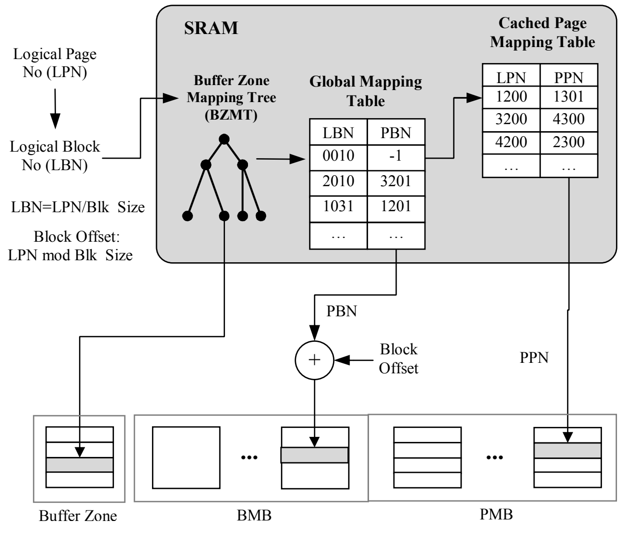


Figure 3. WAFTL Address Translation Process

Table (CPMT) with LPN. If address translation is hit by the CPMT, request can be quickly served with corresponding PPN. Otherwise, a least recently used entry of the CPMT is selected as victim for replacement. Mapping entry with requested LPN is loaded from mapping table blocks by searching Mapping Directory. Request will be eventually served with obtained PPN.

Comparing to page-level DFTL, WAFTL has following advantages. Firstly, overall mapping table reduced because of data partition. Secondly, Buffer Zone is used to improve performance. Any write is logged into the Buffer Zone which only needs to update BZMT. Lookup overhead is reduced. Thirdly, overall lookup is faster than DFTL. Because BZMT and GMT are loaded in SRAM, read requests to the data in the Buffer Zone and BMB can be quickly located and served. Finally, WAFTL reduces the overhead of lookup and replacement. Because only part of blocks is managed by the PMT, the size of the PMT is much less than mapping table of DFTL. Thus, lookup and replacement overhead between cached PMT (CPMT) and PMT is much less than DFTL.

C. Buffer Zone and Data Migration

1) Using Flash as Buffer Zone

WAFTL explores a small part of flash space as Buffer Zone to improve write performance. Buffer Zone could consist of several hundreds of blocks and uses page-level address mapping. Any write or update is sequentially logged into the Buffer Zone. Valid data in the Buffer Zone will be migrated into data blocks only when the Buffer Zone is full. The Buffer Zone will be marked as Dirty Buffer Zone after migration. At same time, a new Buffer Zone is allocated from free block pool. Block erasure for this Dirty Buffer Zone is intensively delayed to reduce impact on foreground accesses, which will be discussed in following subsection D.

Current hybrid FTLs conduct block-level replacement. When there is no empty space in the log buffer, one of the log block is selected as victim to make space for on-going write requests and all of the valid pages in the log block are merged with the corresponding data blocks. Data pages of a logical block may distribute among the victim block and other log blocks. Only flushing victim block may destroy spatial locality. By contrast, WAFTL flushes data at the level of Buffer Zone, which provides more opportunities to form sequential writes.

WAFTL uses a B+tree, Buffer Zone Mapping Tree (BZMT) to maintain logical block association of pages in Buffer Zone (See Figure 4). The BZMT uses logical block number as key. A data structure called *block node* is introduced to describe a logical block in terms of block popularity, page count including clean and dirty pages, and physical page location for each pages of the block. *Block popularity* is defined as block access frequency including reading and writing of any pages of the block. 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 lower

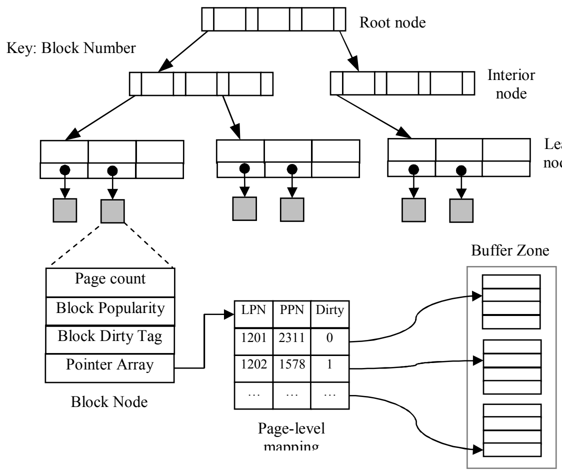


Figure 4. Buffer Zone Mapping Tree (BZMT)

popularity value, while block with random accesses has higher popularity value.

In the real application, read and write accesses are mixed. Usage patterns exhibit block-level temporal locality: the pages in the same logical block are likely to be accessed (read/write) again in the near future. To increase opportunities to form sequential writes, reading missed data will be written into Buffer Zone.

```
Function Buffe_Zone_Migration()
1.  for each block in Buffer Zone
2.      /*existing BMB update*/
3.      if GMT[$block_i$->lbn].pbn != -1 then
4.          if BZMT[$block_i$->lbn].dirty==1 then
5.              write valid pages of $block_i$ to active BMB
6.              copy rest pages of previous BMB to active 8.BMB;
7.              invalidate previous BMB;
8.          endif
9.      else
10.         /*existing PMB update*/
11.         if GMT[$block_i$->lbn].exist != -1 then
12.                 for each page in $block_i$
13.                     if BZMT[$block_i$->lbn].$page_j$.dirty==1 then
14.                         write $page_j$ of $block_i$ to active PMB
15.                         invalidate old page in previous PMB;
16.                     endif
17.                 endfor
18.         else
19.             /*new writes*/
20.             if $block_i$ is most popular then
21.                 write valid pages of $block_i$ to active PMB;
22.             else
23.                 if $block_i$->pagecount >=threshold then
24.                     write valid pages of $block_i$ to active BMB;
25.                 else
26.                     write valid pages of $block_i$ to active PMB;
27.                 endif
28.             endif
29.         endif
30.     endif
31.     update GMT;
32.     update PMT;
33. endfor
```

Algorithm 1: Buffer Zone Migration

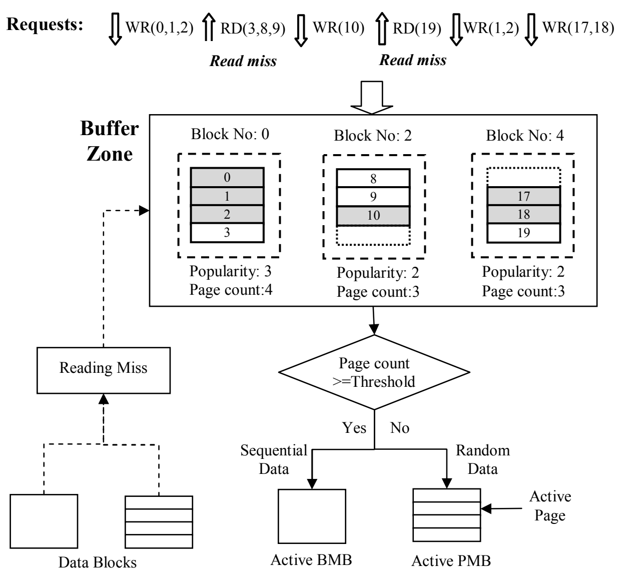


Figure 5. Buffer Zone Architecture. RD: read, WR: write. Grey box and white box in the Buffer Zone indicate dirty and clean page respectively.

An active BMB and an active PMB are maintained at any time. WAFTL differentiates the data in the Buffer Zone during migration. Block popularity and page count are used to determine whether data in the Buffer Zone should be migrated into BMB or PMB. Algorithm 1 describes the process of data migration.

For update of an existing BMB, all valid pages of this BMB including dirty and clean pages are written from the Buffer Zone to the active BMB, and the rest valid pages of this BMB are copied to the active BMB. The previous BMB are marked as invalid block for later erasure. For update of an existing PMB, only dirty pages are written into the active PMB starting from active page pointer. The previous pages are invalidated.

For new arriving data, WAFTL always migrate the pages of the most popular block to the active PMB regardless its page count. Storing the most popular data in PMB can efficiently reduces garbage collection overhead caused by frequent updates. For other blocks, if page count of a logical block reaches the threshold, all valid pages of this block are sequentially written into the active BMB. Otherwise, all valid pages are written into the active PBM (See Figure 5).

2) Determining the value of Threshold

Our objective is to improve performance, at the meantime reduce mapping table. Different threshold may result in different performance and mapping table because it determines the distribution of BMB and PMB. Determining the value of threshold is a trade-off between performance and mapping table size. There are two choices for threshold setting.

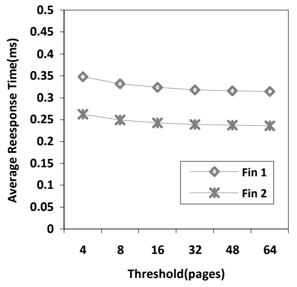
a) Performance oriented threshold

In this scheme, value of threshold is statically set as full block size and only full block data is stored into BMB. This method achieves maximum performance because it minimizes garbage collection overhead by storing most data into PMBs.

At meantime, its mapping table is maximized, which requires big SRAM memory. This can be seen in our evaluation result in Section V. For random dominant workload, most data will be stored into PMBs because it is difficult to form full block sized writes. This method can be used in application without memory constraint.

b) Dynamic threshold considering memory constraint

In order to investigate the proper threshold value, we tested the effects of different threshold value through repetitive experiments over different workloads. We found in our experiments that the value of threshold has different effects on performance and mapping table size (See Figure 6). Mapping table size is more sensitive than performance. The difference of performance when threshold is set as 48 pages and 64 pages (full block) is very small. At mean time, mapping table increases linearly with threshold.



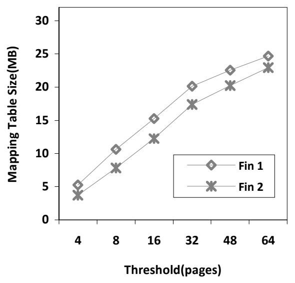


Figure 6. Effect of threshold on performance and mapping table size (32GB SSD, 4KB page,256KB block)

We realized that the value of threshold can be dynamically adjusted considering memory constraint, without scarifying much more performance. Dynamic threshold is achieved by using the following heuristic method. WAFTL uses *THR* to denote the threshold. Clearly, the value of *THR* is from average request size to full block size. P_{THR} and P_{Blk_size} represent performance when threshold is set as *THR* and full block, respectively. M_{THR} and M_{Blk_size} represent mapping table size when threshold is set as *THR* and full block, respectively.

Initially, the value of *THR* is set to Blk_Size (i.e. 64 pages). THR decreases to a smaller value if performance degradation, the ratio between ΔP=(P_{THR} - P_{Blk_size}) and P_{Blk_size} is smaller than D, and mapping table reduction, the ratio between ΔM=(M_{Blk_size}-M_{THR}) and M_{Blk_size} is larger than D, i.e.,

$$\frac{\Delta P}{P_{Blk\_size}} = \frac{P_{THR} - P_{Blk\_size}}{P_{Blk\_size}} \leq D \qquad (1)$$

$$\frac{\Delta M}{M_{Blk\_size}} = \frac{M_{Blk\_size} - M_{THR}}{M_{Blk\_size}} \geq D \qquad (2)$$

Where D is a performance control parameter defined by users. WAFTL explores above constraint to control whether to enlarge or reduce the threshold. With our experiments, the dynamic method can properly adjust the threshold value under different enterprise workloads.

D. Garbage Collection

There are Dirty Buffer Zones, invalid BMBs waiting for block erasure. To reduce impact of garbage collection on user accesses, WAFTL explores offline garbage collection to erase these invalid blocks during idle time (See Figure 7).

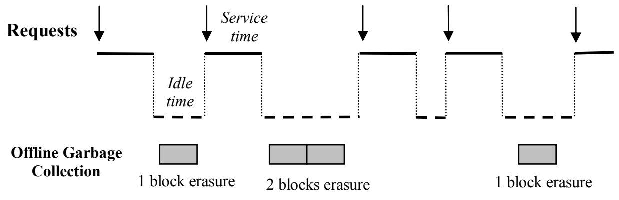


Figure 7. Offline Garbage collection

Idle period ($t_{predict}^{i}$) is predicted according to Equation (3). WAFTL monitors real idle time to adjust the prediction. Instead of erasing all invalid blocks at one time, WAFTL use predicted idle length to calculate and determine how many invalid blocks (N_i) to be erased according to Equation (4).

$$t^{i}_{predict} = \alpha t^{i-1}_{real} + (1-\alpha) t^{i-1}_{predict} \qquad (3)$$

Where t_{real}^{i-1} is last monitored idle length, $t_{predict}^{i-1}$ of last predicted idle length, and α is adjust factor.

$$N_i = \begin{cases} \dfrac{t^{i}_{predict}}{T_{erasure}} & \text{if} \quad t^{i}_{predict} \geq T_{erasure} \\ 0 & \text{if} \quad t^{i}_{predict} \prec T_{erasure} \end{cases} \qquad (4)$$

Where $T_{erasure}$ is time required to erase one block. If predicted idle period is less than $T_{erasure}$, block erasure is not initiated. If the duration of the idle period is longer than $T_{erasure}$, off-line garbage collection is triggered and N_i blocks will be erased. When to initiate an off-line garbage collection is an interesting issue for future investigation. How to determine timeout value has been extensively studied in dynamic power management (DPM) of hard disk drives [19], which puts a disk into a low-power state after a certain idle time in order to save energy.

WAFTL set a garbage collection threshold for the whole SSD space: percentage of free blocks (20% is used in our experiment). If free space is under the threshold, online garbage collection will be triggered. Otherwise, offline garbage collection is exploited. Online garbage collection erases invalid blocks until free block number reaches the threshold.

If free block number is under threshold after erasing all invalid blocks of Dirty Buffer Zones and invalid BMB, garbage collection for PMB is required. There are invalid pages in PMBs caused by partial update.

PMB block with more invalid pages will be selected for garbage collection. Valid pages of selected PMB will be copies to the active PMB before erasing it. Garbage collection will be stopped once free block number researches threshold. We put garbage collection for PMB at lowest priority because of its extra copy overhead and high impacts on user accesses.

E. *Endurance-aware dynamic block allocation*

Free blocks are indexed by lifetime in free block pool. Data block (BMB/PMB) and Buffer Zone are dynamically allocated from free block pool. Since Buffer Zone is more write intensive than data block, we reserve top youngest blocks for future Buffer Zone allocation.

Most FTL schemes are static and their mapping methods are fixed. WAFTL explores data partition and dynamical block allocation to enable flexible mapping method and make it workload adaptive. Either page-level or block-level mapping for a data block is only determined at time of block allocation. Different workload environments have different number of BMB and PMB.

IV. EVALUATION

A. *Experiment Setup*

1) *Trace-driven Simulator*

We built a trace-driven simulator by modifying FlashSim [10]. Four FTL schemes are implemented for benchmark: 1) pure page-level FTL, 2) DFTL [10], 3) FAST[15] and 4) proposed WAFTL. SSD configuration values listed in the Table II are taken from [6].

TABLE II. SPECIFICATION OF SSD CONFIGURATION

SSD	Page Read to Register	25□s
	Page Program (Write) from Register	200□s
	Block Erase	1.5ms
	Serial Access to Register (Data bus)	100□s
	Die Size	2 GB
	Block Size	256 KB
	Page Size	4 KB
	Data Register	4 KB
	Erase Cycles	100 K
Flash Buffer Zone size : 256MB/512MB/1GB		
SRAM for DFTL and WAFTL: 256KB For pure page-level FTL and FAST, we assume SRAM is enough to hold entire mapping table.		

2) *Workload Traces*

We use a mixture of real-world and synthetic traces to study the efficiency of different FTL schemes on a wide spectrum of enterprise-scale workloads. Fin1 and Fin2 were collected at a large financial institution [17]. The Exchange, DevDiv and DevDiv91 traces were collected by Microsoft made available by SNIA[18]. Seq_read and Seq_write are sequential read dominant and sequential write dominant synthetic traces, which are generated by Disksim. Table III presents salient features of our workload traces [16].

TABLE III. SPECIFICATION OF WORKLOADS

Workload	Avg. Req. Size(KB)	Write (%)	Seq. (%)	Avg. Req. Inter-arrive Time(ms)
Fin1	8.29	35	0.6	8.189
Fin2	7.78	17	0.7	11.081
Exchange	54.60	74	0.5	1.315
DevDiv91	36.17	91	0.9	2.044
DevDiv	88.98	72	1.5	2.689
Seq_Read	67.68	19	60	0.0499
Seq_Write	67.55	80	70	0.0497

3) *Evaluation Metrics*

In this study, we utilize (i) average response time (ii) erase count and (iii) total number of page read and write (both ii and iii are indicators of the garbage collection overhead) to characterize the behavior of different FTL schemes.

B. *Experiment Results*

Figures 8 through Figure 14 show average response time, erase count and total number of page operations for different FTL schemes under the seven workloads when we vary SSD capacity. Because of space limitation, we only present total number of page operation for 32GB SSD. The following observations are made from the results.

1) *Performance*

WAFTL outperforms DFTL and FAST in terms of average response time under different workload traces. WAFTL almost researches performance of pure page-level mapping. Note that page-level mapping achieves best performance under different workloads. However, big mapping table make it impractical.

WAFTL achieves up to 34% and 72% performance improvement over DFTL and FAST under different workload traces. Let's take Figure 8(a) as an example. For 32GB SSD, the average response time of WAFTL with 1GB Buffer Zone is 0.29 msecond under Fin1 trace. By contrast, the average response time of DFTL and FAST is 0.39 msecond and 1.02 msecond, respectively. WAFTL makes 26% and 72% performance improvement compared to DFTL and FAST.

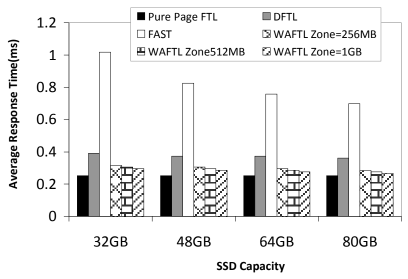
Performance gain of WAFTL comes from reduced random writes with use of Buffer Zone, faster address mapping and reduced overhead of garbage collection.

2) *Garbage Collection Overhead*

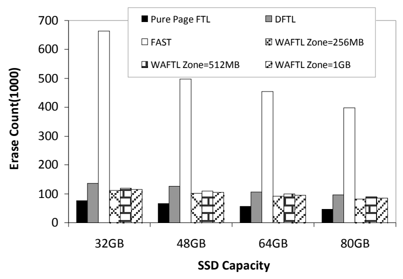
WAFTL exhibits up to 43% and 83% less block erasures compared to DFTL and FAST for different workloads. For 32GB SSD, WAFTL with 512MB Buffer Zone erased 20548 blocks under Fin2, while DFTL and FAST erased 31480 and 222126 blocks, respectively (See Figure 9(b)).

In addition, we can see from the results that FAST and DFTL result in up to 133% and 45% more page operations than WAFTL, respectively. Additional page operations are caused by valid data copy during garbage collection. FAST needs to read and write a large number of additional pages, which impacts the overall performance. The results clearly indicate that WAFTL efficiently reduce garbage collection overhead by efficiently differentiate random and sequential accesses.

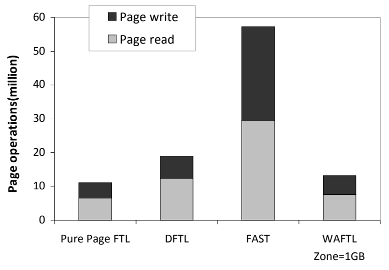
The results further show that performance is directly correlated to the garbage collection overhead. Under workload Exchange, WAFTL with 1GB Buffer Zone makes 43% and 80% less block erasure compared to DFTL and FAST (see Figure 10(b)). Accordingly, there is 26% and 62% performance improvement (see Figure 10(a)). The correlation clearly indicates that performance gain of WAFTL mainly comes from reduced overhead of garbage collection. FTL performance highly depends on efficiency of garbage collection. WAFTL exploits page-level and block-level mapping to manage different data in hybrid way and makes its contributions through reducing garbage collection overhead.



(a)Average response time

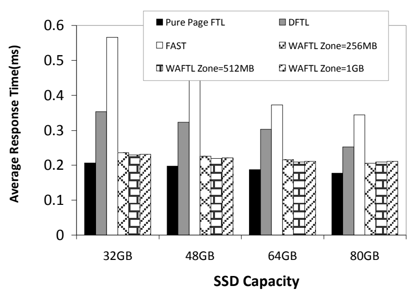


(b) Number of block erasures

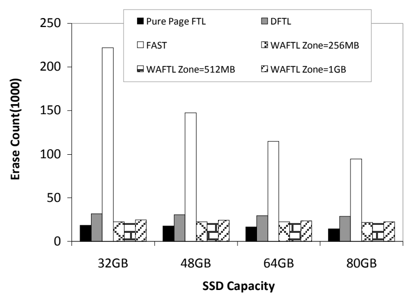


(c) Total page read and write (32GB SSD)

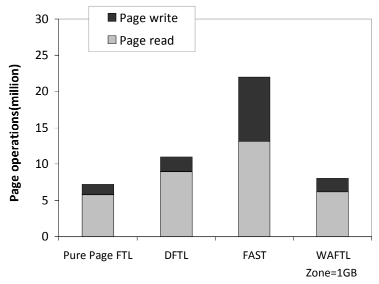
Figure 8. Fin 1 Trace (Migration threshold=64pages,256KB)



(a)Average response time

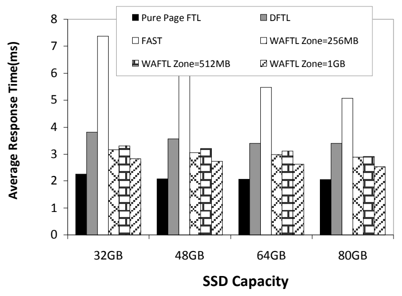


(b) Number of block erasures

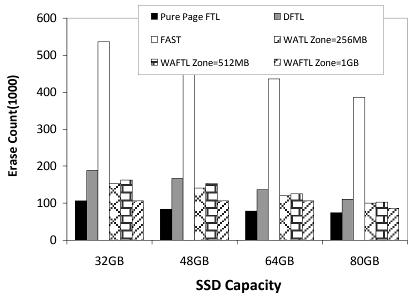


(c) Total page read and write (32GB SSD)

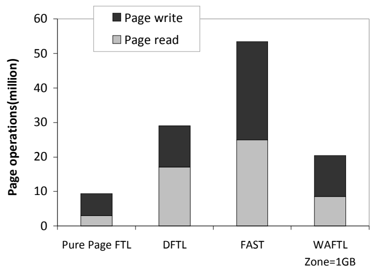
Figure 9. Fin 2 Trace (Migration threshold=64pages,256KB)



(a)Average response time

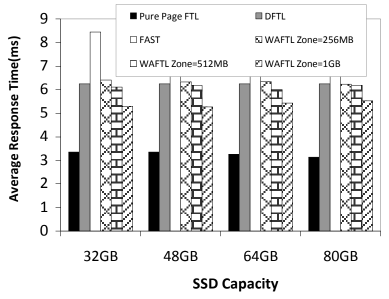


(b) Number of block erasures

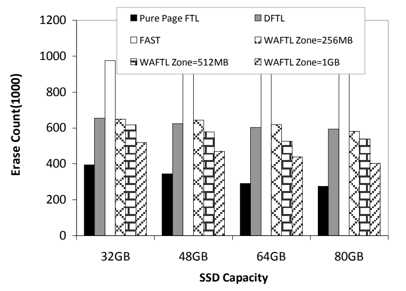


(c) Total page read and write (32GB SSD)

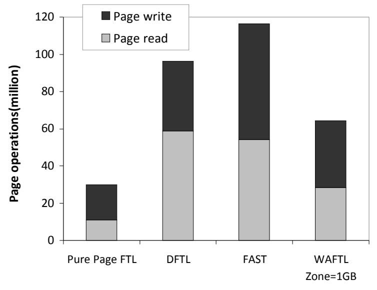
Figure 10. Exchange Trace (Migration threshold=64pages,256KB)



(a)Average response time

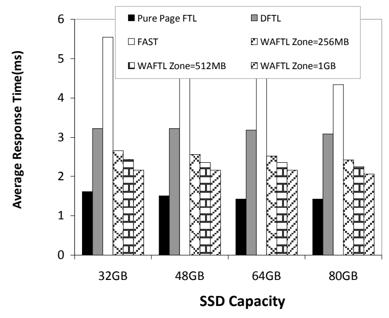


(b) Number of block erasures

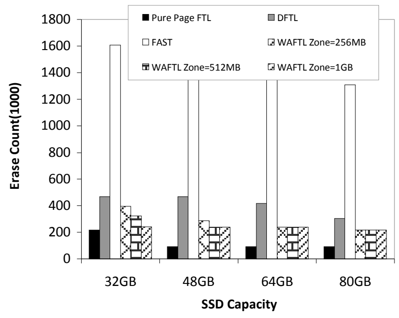


(c) Total page read and write (32GB SSD)

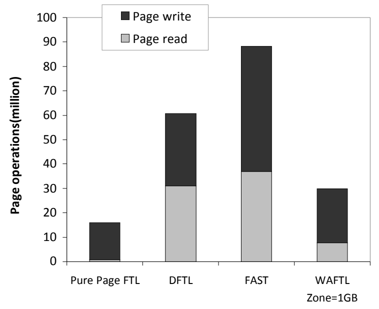
Figure 11. DevDiv Trace (Migration threshold=64pages,256KB)



(a)Average response time

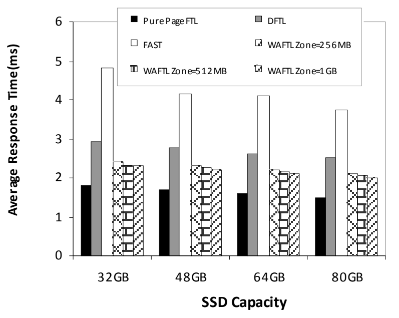


(b) Number of block erasures

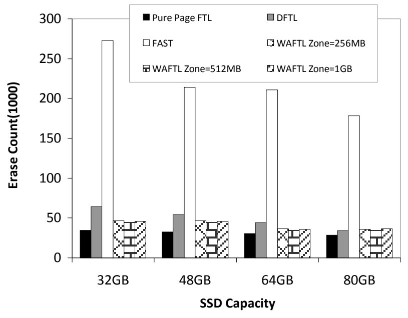


(c) Total page read and write (32GB SSD)

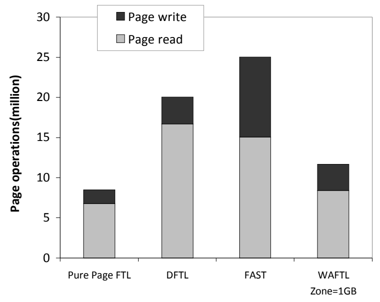
Figure 12. DevDiv91 Trace (Migration threshold=64pages,128KB)



(a)Average response time

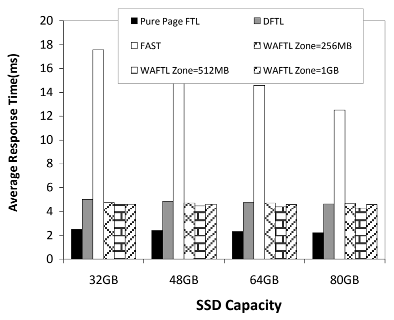


(b) Number of block erasures

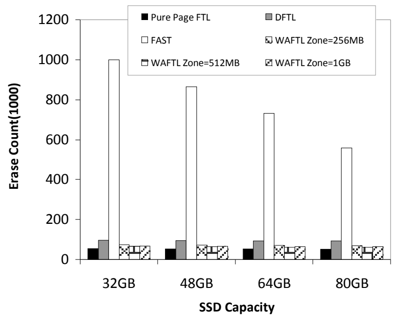


(c) Total page read and write (32GB SSD)

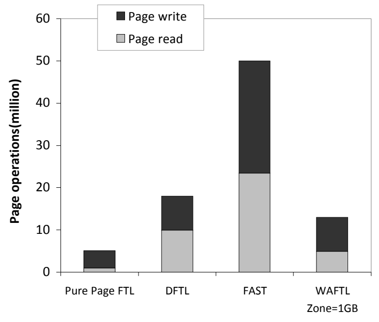
Figure 13. Seq_Read Trace (Migration threshold=64pages,256KB)



(a)Average response time



(b) Number of block erasures



(c) Total page read and write (32GB SSD)

Figure 14. Seq_Write Trace (Migration threshold=64pages, 256KB)

3) *Workload adaptive*

The Figures 15 shows the percentage of PMB across different workloads for different SSD capacity. Different workloads result in different distribution of PMB and BMB. Random intensive workload will have more PMB, while sequential intensive workload will have more BMB. For example, percentage of PMB is about 90% under random intensive workload DevDiv91. By contrast, PMB occupies only 45% under sequential read workload Seq_read.

The results indicate that WAFTL is workload adaptive. WAFTL achieves high performance by efficiently adapting to different workloads with on-demand block allocation and flexible address mapping.

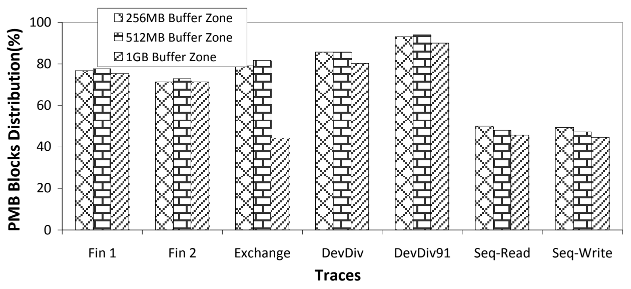


Figure 15. WAFTL is adaptive to different workloads (SSD model: 32GB, 4KB page, 256KB block)

4) *Mapping Table size*

Figure 16 shows the size of the mapping table of different FTL schemes for 32GB SSD. Note that we compare the size of the whole mapping table instead of cached mapping table. The size of mapping table is calculated on the basic of whole SSD space (i.e. 32GB) instead of workload volume. DFTL has same size of mapping table as pure page-level mapping (32MB mapping table for 32GB SSD). Mapping table of WAFTL is calculated according to distribution of PMB and BMB (See Figure 15). We can see that the mapping table of page-level mapping, DFTL, and FAST keeps unchanged when workload changes. This is because their mapping scheme is static. By contrast, mapping table of proposed WAFTL varies with workloads because distribution of PMB and BMB changes with workloads.

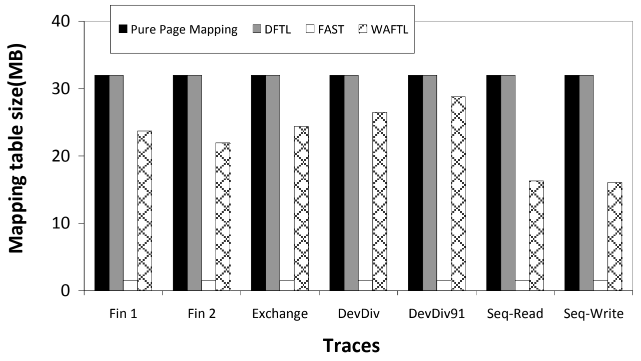


Figure 16. Comparison of mapping table size under different workloads (Buffer Zone=256MB, SSD model: 32GB, 4KB page, 256KB block)

We can see from result that WAFTL reduces mapping table up to 50% over pure page-level FTL and DFTL under different workload traces. At same time, performance of WAFTL is better than DFTL, approaching performance of pure page-level FTL. FAST maintains smallest mapping table because blocks are fundamentally managed by block-level. However, its performance is much worse than WAFTL.

5) *Effect of Migration Threshold*

Different threshold results in different distribution of PMB and BMB (See Figure 17). We can see from the figure 17 that percentage of PMB almost increases linearly with threshold.

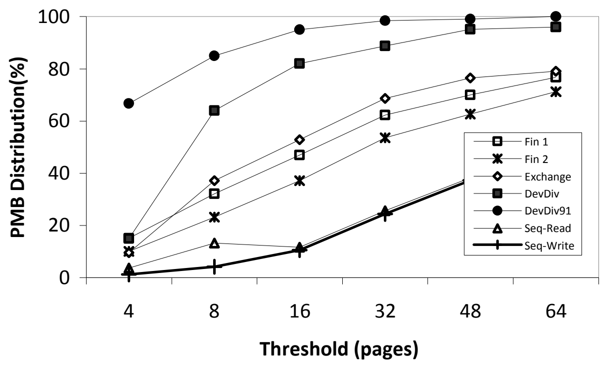


Figure 17. Effect of Threshold on PMB Distribution (Buffer Zone=256MB, SSD model: 32GB, 4KB page, 256KB block)

Figure 18 and Figure 19 present testing result of static threshold and dynamic threshold for different traces. We can see from the result that statically setting threshold as full block size (i.e. 64 pages) can achieve best performance at cost of biggest mapping table. In addition, effect of threshold on mapping table and performance are different. Mapping table is more sensitive than performance.

Dynamically adjusting the threshold under constraint *D* can efficiently reduce mapping table without much more performance degradation. Comparing to full block static threshold, dynamic threshold reduces mapping table up to 49% and 74% when D is set as 0.03 and 0.06.

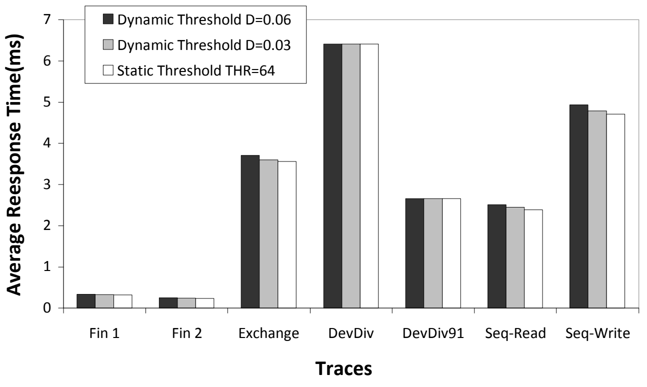


Figure 18. Effect of Threshold on Performance (Buffer Zone=256MB, SSD model: 32GB, 4KB page, 256KB block)

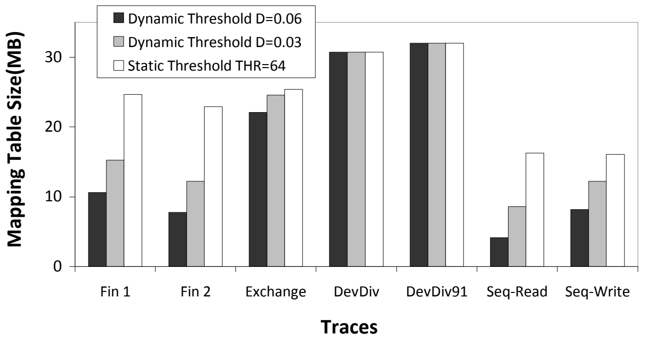


Figure 19. Effect of Threshold on Mapping Table (Buffer Zone=256MB, SSD model: 32GB, 4KB page, 256KB block)

V. RELATED WORK

Research works on FTL try to improve performance and address the problems of high garbage collection overhead. FTL schemes can be classified into page-level, block-level, and hybrid FTL. Page-level FTL scheme achieves high performance at cost of large mapping table. Block-level FTL significantly lowers mapping table, but its performance is poor.

In order to overcome these disadvantages, the hybrid FTL was proposed. Hybrid FTL uses a block-level mapping to manage most data blocks and uses a page-level mapping to manage a small set of log blocks, which works as a buffer to accept incoming write requests [9,20]. Research works on Hybrid FTL try to improve performance and address the problems of high garbage collection overhead. BAST [13] exclusively associates a log block with a data block. In presence of small random writes, this scheme suffers from increased garbage collection cost. FAST [15] keeps a single sequential log block dedicated for sequential updates while other log blocks are used for random writes. SuperBlock FTL scheme [11] utilizes block level spatial locality in workloads by combining consecutive logical blocks into a Superblock. It maintains page level mappings within the superblock to exploit temporal locality by separating hot and cold data within the superblock. The Locality-Aware Sector Translation (LAST) scheme [5] tries to alleviate the shortcomings of BAST and FAST by exploiting both temporal locality and sequential locality in workloads. It further separates random log blocks into hot and cold regions to reduce garbage collection cost. More recently, Janus-FTL [3] was proposed to provide a spectrum between the block and page mapping schemes. By adapting along the spectrum, Janus-FTL uses fusion and defusion to dynamically change the sizes of data block and log blocks for various workload patterns. Hybrid FTLs improve write performance and require a small-sized mapping table. However, they incur expensive garbage collection overhead for random write dominant workloads.

Unlike currently predominant hybrid FTLs, Demand-based Flash Translation Layer (DFTL) [10] is purely page-mapped, which exploits temporal locality in enterprise-scale workloads to store the most popular mappings in on-flash limited SRAM while the rest are maintained on the flash device itself. HAT[1] creates a separate access path to read/write the address mapping information to significantly hide the address translation latency by storing mapping table in PCM. This idea can be incorporated into our proposed WAFTL to further reduce translation latency. Convertible Flash Translation Layer (CFTL) [4] dynamically switches the mapping of a data block to either read-optimized or write-optimized mapping scheme in order to fully exploit the benefits of both schemes. However, mapping scheme switch incurs significant overhead for random intensive workloads.

This paper differs from the above mentioned studies in a number of ways. First, WAFTL partitions data blocks into PMB and BMB for random and sequential writes. PMB and BMB are dynamically allocated, which enables WAFTL workload adaptive. A second major feature of this study is that overall mapping table is reduced due to data partition. Therefore, address translation can be quickly conducted by in-memory BZMT, GMT and CPMT. Replacement overhead between CPMT and PMT is also reduced compared to DFTL because only partial blocks are managed by PMT. Third, Buffer Zone is used to log any write to flash, which not only improve performance and reduce random writes, but also reduce address translation overhead. PCM is exploited to further improve performance. Finally, threshold-based data migration is used to partition data. Static threshold and dynamic threshold are proposed and evaluated.

VI. Conclusion

In this paper, we present a workload adaptive flash translation layer called WAFTL. WAFTL explores page mapping block to store random data and handle large number of partial updates, and block mapping block to store sequential data and lower overall mapping table. Blocks are dynamically allocated and mapping scheme for data block eventually depends on workload. An efficient address mapping is used to reduce overall mapping table and quickly conduct address translation. WAFTL explores Buffer Zone to log data sequentially and partition data based on threshold. Static and dynamical threshold setting are proposed to balance performance and mapping table size. Our benchmark results conclusively demonstrate that proposed WAFTL is workload adaptive, high-performance and efficiently reduces mapping table size.

References

[1] Yang Hu, Hong Jiang, Dan Feng, etc., Achieving Page-Mapping FTL Performance at Block-Mapping FTL Cost by Hiding Address Translation, *In Proc. Of MSST 2010*, 2010.

[2] Guangyu Sun, Yongsoo Joo, Yibo Chen, etc., A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement, *In Proc. Of HPCA'10*, 2010.

[3] Hunki Kwon, Eunsam Kim, Jongmoo Choi, etc., Janus-FTL: Finding the Optimal Point on the Spectrum Between Page and Block Mapping Schemes, *In Proc. Of EMSOFT'10*, 2010.

[4] Dongchul Park, Biplob Debnath, and David Du, CFTL: A Convertible Flash Translation Layer with Consideration of Data Access Patterns, Technical Report, *TR 09-023*, University of Minnesota, Sept., 2009.

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

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

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

[8] H. J. Choi, S. Lim, and K. H. Park. JFTL: a flash translation layer based on a journal remapping for flash memory. *In ACM Transactions on Storage*, vol.4, Jan 2009.

[9] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System software for flash memory: a survey. *In Proc. of ICEUC'06*, 2006.

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

[11] J. Kang, H. Jo, J. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. *In Proc. of ICES'06*, 2006.

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

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

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

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

[16] John S. Bucy, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, The DiskSim Simulation Environment Version 4.0 Reference Manual, *Technical Report, CMU-PDL-08-101*, Carnegie Mellon University, May,2008.

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

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

[19] L. Benini, A. Bogliolo, and G. D. Micheli, A Survey of Design Techniques for System-level Dynamic PowerManagement, IEEE Transactions on VLSI Systems, vol. 8, no.3, 2000.

[20] Sang-Phil Lim, Sang-Won Lee and Bongki Moon, FASTer FTL for Enterprise-Class Flash Memory SSDs, 6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2010), Incline Village, NV, USA.