ADAPT: Efficient Workload-sensitive Flash Management Based on Adaptation, Prediction and Aggregation

Chundong Wang and Weng-Fai Wong
School of Computing, National University of Singapore
Email: {wangc, wongwf}@comp.nus.edu.sg

***Abstract*—Solid-state drives (SSDs) made of flash memory are widely utilized in enterprise servers nowadays. Internally, the management of flash memory resources is done by an embedded software known as the flash translation layer (FTL). One important function of the FTL is to map logical addresses issued by the operating system into physical flash addresses. The efficiency of this address mapping in the FTL directly impacts the performance of SSDs. In this paper, we propose a hybrid mapping FTL scheme, called *Aggregated Data movement Augmenting Predictive Transfers* (ADAPT). ADAPT observes access behaviors online to handle both sequential and random write requests efficiently. It also takes advantage of locality revealed in the history of recent accesses to avoid unnecessary data movements in the required merge process. More importantly, by these mechanisms, ADAPT can adapt to various workloads to achieve good performance. Experimental results show that ADAPT is as much as 35.4%, 44.2% and 23.5% faster than a state-of-the-art hybrid mapping scheme, a prevalent page-based mapping scheme, and a latest workload-adaptive mapping scheme, respectively, with a small increase in space requirement.**

I. Introduction

The access to secondary storage of enterprise servers varies significantly. As flash-based solid-state drives (SSDs) start replacing traditional hard disks, an efficient and adaptive algorithm on flash management that takes advantage of runtime access behaviors is likely to achieve better performance.

There are two types of flash memory, NOR flash and NAND flash. The latter has a higher density and is cheaper, making it more prevalent today. Unlike NOR flash, NAND flash is not byte addressable. Read and write operations on a NAND flash chip must be performed in units of *pages*. Each page has a data area for storage and a spare area for essential information [16]. However, data in a page cannot be written ("*programmed*") unless the *block* it is in is first erased [22]. A block is the unit for an erasure and contains multiple pages [6]. Such *out-of-place* data updating is a primary concern on flash management, especially the logical to physical address mapping.

The management of flash memory is performed by an embedded software called the *flash translation layer* (FTL). The FTL services requests from the upper-level file system and performs actions at the lower-level flash memory. Its basic functionalities include address mapping, wear leveling and bad block management. Among these, the mapping from logical address to physical address has the most impact on the performance due to its frequent use, making it an ideal candidate for optimization.

Existing FTLs, like BAST [8] and FAST [13], target the address mapping of embedded systems. With the widespread use of SSDs in enterprise servers, workload characteristics of general-purpose computing systems have to be considered. For example, FAST's successor FASTer [15] focuses on online transaction processing (OLTP) systems. I/O requests of OLTP systems are generally random and small with a handful of data highly accessed.

Besides OLTP, however, there are also other important classes of I/O workloads. For instance, mail and media servers serve contents that may be fairly large. These types of workloads differ from OLTP in that accesses are less skewed and generally more data need to be written or read in a request; sequential and random write requests may mix in different ratios and form dynamic access patterns, which requires FTLs to adapt to them efficiently for high access performance.

Several FTLs have been proposed to exploit the access behaviors of workloads. The above mentioned FASTer targets OLTP systems. LAST [14] separates sequential and random writes by the number of pages to be accessed in a request, and deals with them differently. Both of them are *hybrid mapping* scheme [14] [23]. Hybrid mapping is a combination of basic *page mapping* and *block mapping* by dividing all physical blocks into the *data space*, *log space* and *free block pool*. Each logical block is block-mapped to a block in data space. Block mapping is not flexible when updating data to a page because of the coarse granularity of mapping unit and out-of-place updating. Therefore, the log space is maintained to temporarily hold updates using page mapping. Newly-arrived updates will be put into log pages. When no clean page is left in the log space, a victim log block will be selected and *merged* with corresponding data blocks. After merging, the victim is erased and returned to the free block pool. Another clean block will be allocated to replenish the log space. During a merge in FASTer, a page containing valid data is given a *second chance* by being retained inside the log space.

In this paper, we propose a novel hybrid mapping scheme called *Aggregated Data movement Augmenting Predictive Transfers* (ADAPT). It is a workload-dependent adaptation heuristic that considers access behaviors at runtime in its maintenance of the log space. The main contributions of this

978-1-4673-1747-4/12/$31.00 ©2013 IEEE

TABLE I
I/O REQUEST SIZE OF VARIOUS WORKLOADS

Trace	Small	Medium	Large
TPC-C_20	99.17%	0.83%	0.00%
SPC1	86.58%	10.63%	2.79%
MSR-hm_0	76.70%	13.72%	9.58%
MSR-mds_0	72.35%	19.79%	7.86%
MSR-prn_0	79.46%	8.88%	11.66%
MSR-prxy_0	87.91%	6.82%	5.27%
MSR-rsrch_0	68.22%	25.04%	6.74%
MSR-stg_0	72.33%	18.62%	9.05%
MSR-ts_0	67.81%	25.87%	6.32%
MSR-web_0	67.50%	23.85%	8.65%

paper, and also main components of ADAPT, are:

- An online algorithm *adaptively* partitions the log space so as to efficiently handle requests that are a mix of sequential and random writes. The log space is usually divided into a *sequential area* and a *random area* for sequential or random writes respectively. Unlike previous designs using fixed sizes, ADAPT will monitor the processing of write requests at runtime and dynamically adjust the capacities of the two areas.
- A merge-or-move decision procedure based on a *prediction* mechanism is employed in the random area of the log space. This mechanism considers the recent history of writes and will compute the likelihood of a page being updated in the near future. The basic idea is that during a merge, if a page is found to have been written recently, it will be given a second chance to stay in the log space. Otherwise, it will be directly merged.
- Sometimes, it may turn up that most pages in the log block to be merged are valid. During the merge, our *aggregated data movement* scheme will give the entire block a second chance instead of processing pages one at a time, and simply append the block to the end of random area of the log space.

These features complement each other, and enable ADAPT to outperform previous designs, as will be shown in our experiments.

The rest of this paper is organized as follows. Section 2 presents access patterns of various workloads and background of hybrid mapping. Section 3 describes the details of ADAPT. Section 4 shows experimental results with Section 5 presenting some related works. Section 6 will conclude this paper.

II. MOTIVATION AND BACKGROUND

A. Motivation

Workload characteristic is an important factor in the design of FTL. For instance, FAST paid more attention to random writes using only one log block for sequential writes [13]. FASTer was designed mainly with OLTP systems in mind [15]. Typical OLTP workloads are dominated by small and random I/O requests. A high-level access skewness exists on a handful of pages with other pages rarely touched.

However, besides OLTP, there are other types of server workloads. Table I shows that the variation in I/O request sizes is significant. Traces in Table I are from [20], [24] and [18], collected in various environments. Here we define a *small* request as one that is 4KB (2 pages with 2KB per page), or less. This same definition was used by previous works [14] [15]. A *medium* request is one whose size is smaller than 16KB (8 pages), and any request that is larger is classified as *large*. For preliminary analysis, we roughly deem large requests to be sequential, which agrees with LAST [14]. TPC-C_20 in Table I is a typical OLTP workload which hardly has sequential writes but is almost full of random requests in all 7.7 million write records. Comparatively, MSR-prxy_0, one that was taken in a proxy server and also has a large amount of small writes, contains a lot of large requests. For non-OLTP workloads in Table I, sequential writes compose about 3% to 12% of all requests. If these requests are handled, for example, with one log block as in FAST, there will be high capacity misses that can badly degrade the performance. For small random requests, since they are frequent and interpose with sequential writes, how to satisfy them is always attractive in the development of FTLs. One key insight of our design is that the FTL should use an intelligent strategy to deal with workloads that are mixed with sequential and random writes.

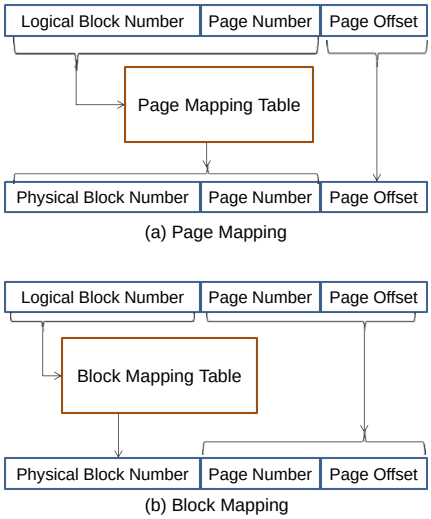

Fig. 1. Page mapping and block mapping

B. Background

Hybrid mapping is the preferred mechanism in FTL design. While the logical-to-physical block mapping is used as the basis, page mapping is employed to manage temporary updates. Fig. 1 sketches the two basic mapping schemes.

Typically, the log space is about 3% of all space [12] [15]. It is usually partitioned into a sequential area for sequential writes and a random area for random writes. FAST uses one log block for its sequential area while LAST has a fixed number of blocks. They also have methods to decide whether a request is sequential or random. When a write request arrives, the FTL first checks whether the page in the mapped data block is clean. If not, a log page will be allocated to accept the data. The old copy will be invalidated. The relationship between the logical page and the log page is recorded in the log page mapping table. Fig. 2 is adopted from [14]. In Fig. 2 a square

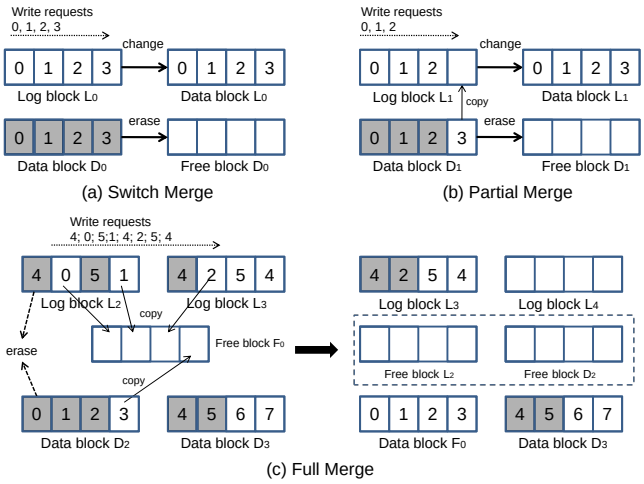

Fig. 2. Three types of merge

is a page and a rectangle of four squares represents a physical block. The number in each square is the logical page number that it maps to. Data in a shaded page are invalid. In Fig. 2(c), logical page 2 is mapped to data block D_2 but cannot be rewritten directly. A page in log block L_3 has to be allocated. Successive updates can be handled by more log pages, and mapping entries are changed accordingly. In Fig. 2(c), three log pages in L_2 and L_3 are used for logical page 4. If all pages of log space are exhausted, a merge procedure must be performed to make space.

Fig. 2 shows three types of merge in FAST. *Switch* and *partial merge* have lower overheads, and are expected in the sequential area. For a switch merge (shown in Fig 2(a)), the log block contains contiguous valid data from the same logical block. It can therefore be simply switched to data space. In a partial merge, the log block will also replace its relevant data block but some valid data in current data block have to transferred to it first, as shown in Fig 2(b). *Full merge* is more complicated. FAST is a *fully associative* hybrid mapping scheme, which means a log block in the random area is not bound to any one data block like BAST but shared by all. Thus, a full merge is costly because each page with valid data in the log block must be (potentially) merged with a different data block. This requires many writes and erasures. FAST and FASTer organize the random area in a FIFO queue (that they called "round-robin"), and the victim for the full merge will be the one at the head of the random area.

A recent proposal, WAFTL [25], considers the issue of workload adaptation. It also combines two basic mapping schemes, but differs from hybrid mapping in its management on buffer space and data blocks. It has a page-mapping *buffer zone* like the log space to hold updates, and data blocks are partitioned into Block-level Mapping Blocks (BMB) and Page-level Mapping Blocks (PMB). When the buffer zone is full, a *data migration* procedure will be called to transfer the data out. WAFTL claims to be workload-adaptive by sending buffered data to either BMB or PMB upon their access frequencies: highly accessed will be sent to PMB and others will be put in BMB. Unlike merging a log block, data migration will flush all data in buffer zone and completely reconstruct the space. It is quite costly to move so many data at a time.

III. Our Proposed FTL Scheme: ADAPT

A. Overview of ADAPT

In this section, we will describe ADAPT. It is a fully-associative hybrid mapping FTL that also utilizes log space to temporarily hold updates. However, ADAPT manages log space in a novel way to adapt to various workloads. Essentially, it adjusts the partitioning of the log space in response to sequential and random write requests met during runtime. By observing online access behavior, ADAPT also avoid premature merges by predicting the likelihood of future references.

B. Online Adaptive Partitioning of the Log Space

How to efficiently handle sequential writes and random writes is an important issue in FTL design. As mentioned before, the log space is partitioned into the sequential and random areas. Hybrid mapping schemes always expect sequential writes to cause switch or partial merge. FAST utilizes one log block as the sequential area [13] while LAST considers multi-tasking environments and employs a fixed number of log blocks to handle sequential requests [14]. On one hand, using one log block tends to result in block thrashing. On the other hand, since the system workload changes from time to time, it is also not optimal to reserve a fixed number of blocks.

Before presenting our design, let us first revisit the issue of identifying sequential writes. FAST uses two conditions to direct a write request to the sequential log block: (1) if its page number is zero within the logical block (data that the log block holds at present will be merged), or (2) the logical block number of the write request is the same to that of the sequential log block and the pages to be written can be simply appended in the log block. The first condition is likely to incorrectly label a random request beginning from page zero to be a sequential one, and may result in frequent merges. In LAST, besides using more blocks for sequential writes to avoid such merges, it also takes into account the size of a write request: if a request writes data to a number of pages, it will be a sequential request. LAST was implemented in PCs of Windows XP operating system, so its threshold was set to be 4KB (2 pages) [14]. For ADAPT, however, we do not use an absolute number of pages accessed in a request to determine whether it is sequential or random. Instead, we will adaptively change the threshold. How this is done will be described below.

We shall now present our area partitioning scheme of ADAPT. Unlike FAST or LAST, the sizes of the sequential and random area are adjusted dynamically. The key idea is that, at runtime, if performance suffers from having insufficient sequential log blocks, blocks will be transferred from the random area to the sequential area, and vice versa. To do these, ADAPT maintains two variables in a time interval. The first

is the *switch and partial merge ratio*,

$$\delta = \frac{\textit{count of switch and partial merges}}{\textit{count of sequential log block allocation}}.$$

This is the count of switch and partial merges over all block allocations from the sequential area. Another one is the *full merge ratio*,

$$\varphi = \frac{\textit{count of merged pages in full merges}}{\textit{count of full merge}}.$$

This is the average number of merged pages in the full merges occurring in the random area.

Algorithm 1: Adjustment of Log Space in ADAPT

```
1  begin
2      reqst_count++;
3      if (reqst_count < INTERVAL ) then
4          return;
5      end
6      else
7          reqst_count := 0;
8          if (δ > 0.4) then
9              victim := GetHeadofRandArea(void);
10             Merge(victim, RW);
11             free_blk := AllocFreeBlock(void);
12             AddtoSeqArea(free_blk);
13         end
14         else if (φ ≥ BLOCK_SIZE/2) then
15             victim := GetVictimofSeqArea(void);
16             Merge(victim, SW);
17             free_blk := AllocFreeBlock(void);
18             AddtoRandArea(free_blk);
19         end
20         return;
21     end
22 end
```

δ and φ represent the situations of recent write requests in a certain period inside the sequential and random areas, respectively. δ varies from 0 to 1. A larger δ means a higher hit rate of block allocation in sequential area. Evidently enlarging the capacity of sequential area is likely to be profitable. Based on our empirical observations, if $\delta > 0.4$ we will do so. On the other hand, a smaller δ implies more requests were incorrectly treated as sequential, and hence the need for sequential log blocks is not high. φ is an integer between 0 and the number of pages in a block, typically 64 [6]. A larger φ means that on average a full merge has to process more valid pages. So having more blocks in the random area may alleviate the pressure. Experiments show that it is time to enlarge the random area when $\varphi \geq 32$, which means on average more than half a block have to be processed in a full merge. A smaller φ implies random requests are handled well by the current random area size, and possibly some blocks can be transferred to sequential area. By measuring δ and φ, ADAPT can adjust the utilization of blocks in both areas. To avoid significant fluctuation on performance, ADAPT will transfer one block every time between two areas. If δ suggests increasing the sequential area, ADAPT will select a victim block in the random area, merge it with its relevant data blocks and reclaim it. A clean block will be allocated to be a sequential log block then. The random area can be adjusted likewise.

In ADAPT, the enlargement of the sequential area has a higher priority than that of the random area. That is to say, ADAPT will consider δ before φ. There are several reasons for this. Firstly, switch and partial merges are less expensive. Secondly, sequential log blocks are managed using block mapping, which consumes less RAM space. Thirdly, the utilization of random log blocks can be optimized with ADAPT's predictive transfer and aggregated movement components, which will be covered in next few subsections.

Unlike LAST's predefined 4KB threshold, the threshold of ADAPT to direct a request to the sequential or random area is also adaptive. In a recent interval, a very small δ, say less than 0.1, means that the sequential area was not very effective. This will cause the threshold to be changed. From our observation on enterprise workloads, over a long period of time, sequential writes tend to access a similar number of pages, either a handful (around 2 pages) or a large number (about 32 pages). Thus, if the threshold is very low, ADAPT will increase it to a large value. Otherwise, the threshold will be decreased. This simple adjustment is quite easy to implement. From our experiments, however, we saw that a latency might be needed to gradually adapt to a specific workload.

Algorithm 1 shows main steps in adjusting the log space. The adjustment is activated every `INTERVAL` requests (line 2 to 5). The impact of the interval length will be discussed in Section 4. We check δ first at line 8. If it is not positive, we will check φ at line 14. The partitions are then adjusted as described above. A victim block is picked from one area and merged with its data blocks (line 9 to 10 or line 15 to 16). A free block will be allocated from the free block pool to replenish the other area (line 11 to 12 or line 17 to 18). The way to select a victim in the random area is the same as a common merge procedure. ADAPT organizes the random area as a FIFO queue like FAST and FASTer, and the head will be the victim each time (line 9). For the sequential area, however, it is better to find a block that will make a full merge or partial merge, which is computed by the function at line 15.

Note that our adaptation is different from previous works [19] [10]. They adapt by changing the degree of associativity between the random log space and the data space. In other words, a log block may be changed from being shared by many data blocks to being bound to one. ADAPT's adaptivity focuses on the partitioning of log space to service either type of write request efficiently. ADAPT also differs from WAFTL whose adaptation is in the transfer of data from the buffer zone to either the page or block mapping areas of the data space.

C. Predictive Transfers

The second chance scheme is the main feature of FASTer. FASTer gives valid data in the victim log block a second chance thereby preventing premature merges. With the second chance scheme, valid data from the head block of the random area will be moved to the block at the rear of the queue. FASTer performs well for OLTP systems because they frequently access little data from some very hot logical pages and not too many data would be left in the victim log block for movement. For other classes of workloads, however, such movements seem wasteful. While the second chance scheme can reduce the number of erasures, it may significantly increase the amount of data copying. Table II shows typical latencies of write and erasure of NAND flash [6]. It can be deduced that moving five pages will reverse the gain of an avoidance of an erasure. This is especially detrimental if a page given a second chance turn out not to be accessed during the time it is in the log space. This leads to the idea that if we can predict the likelihood whether a page in the random area will be used, we can make a better decision on whether to delay merging this page or not. In general, a page at the front of the random area that is likely to be accessed again should be given a second chance in a merge process. Otherwise, if it is unlikely to be updated in the near future, then it should be directly merged.

TABLE II
LATENCIES OF LARGE-BLOCK SLC NAND FLASH

Read	Write	Erase
130.9 μs (2KB)	405.9 μs (2KB)	2 ms (128KB)

As with most forms of prediction, the principle of temporal locality can be applied here. Particularly, a page that has been written recently is most likely to be updated again. We utilize the historical write information of a page to predict its future access possibility. Hence, on deciding if a page should be given a second chance, we shall examine whether its data were recently updated.

The data structure for prediction of ADAPT is the *historical access table* (HAT). HAT records a history of recent writes to logical pages at runtime. It is a hashed queue maintained in RAM. The key used for hashing is the logical base address in a write request, i.e., the concatenation of the logical block and page numbers. Each entry consists of the key and the number of pages that was written within a historical request. Hashing allows for queries about the existence of an entry to be answered quickly. Entries in the HAT are updated dynamically and managed via a queue discipline. On a coming write request, if its logical base address and the size to be accessed are already cached in the HAT, it will be moved to the rear; otherwise, a new entry will be enqueued in the HAT. If the HAT is already full, the entry at the head of the queue (the least recent one) will be deleted to make room for the new entry. In this way, we maintain a history of the most recent writes for the purpose of prediction.

The overhead of HAT is not significant. It is resident in RAM together with address mapping tables, thus having a much shorter access latency than flash memory. The HAT does not need to be persistently stored since access behaviors are always changing and not correlated over a long time. The HAT is also small. As shown in Fig. 3, each entry of the HAT has two fields, the base page number (4 bytes), and the number of pages accessed (2 bytes). Thus, 1KByte of RAM can hold about 170 requests. Our experiments actually showed that a 1KByte HAT could perform well. More discussions of the size of HAT will be given in Section 4 with various configurations.

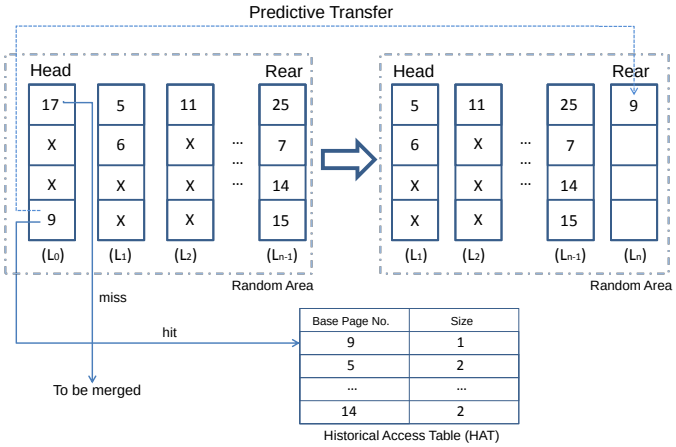

Fig. 3. Predictive transfer with historical access table

Fig. 3 gives an instance of merging with prediction. A rectangle is a physical block, and each has four squares for four pages. The number in a physical page represents the mapped logical page. An 'X' means invalid data that will be skipped in merging. Suppose the random area space runs out of free pages. A merge procedure will be performed. Firstly, a new block (L_n) will be allocated from the free block pool, and enqueued to the rear of the random area. In return, block L_0 will be removed and examined. In Fig. 3, there are two valid pages in block L_0, namely page 0 and page 3. Page 3 corresponds to logical page 9. Its access record exists in the HAT, and so it is given a second chance, i.e., it will be copied into block L_n. However, the record for page 0 (which maps to logical page 17) cannot be found in the HAT, and it will be merged immediately with its relevant data block.

ADAPT's predictive transfer is different from the *adaptive merge* of a recently proposed hybrid mapping FTL named MAST [23]. MAST uses 2D-striping to access data. When a merge has to be performed, MAST will also migrate valid log pages to other log blocks. However, in merging or migrating a log page, MAST will consider the logical block it is related to. If that logical block is cold, and its total number of related log pages is bigger than a predefined threshold, the log page will be merged. Otherwise it will be migrated. In other words, MAST's criterion is the number of log pages that a logical block is using, while ADAPT utilizes the recent access history of the logical page of the corresponding log page.

D. Aggregated Data Movement

As we have observed in our experiments, with the workloads from media and file servers, the victim log block to be merged may still have a lot of valid pages. I/O requests of non-OLTP systems may not be that small, as is shown in Section 2. Especially in multi-tasking environments, the access to storage may be switched to other applications frequently, thereby leaving many log pages valid even when they are to be merged. It is inefficient to process these pages one at a time. Therefore, we propose an *aggregated data movement* scheme to solve the problem. The example in Fig. 4 will be used to explain this scheme. When the random area runs out of free pages, the merge procedure will be called. Unlike before, we shall first examine the two blocks at the head of the random area, i.e., L_0 and L_1 in Fig. 4. If the number of valid pages in L_0 does not exceed an *aggregated move threshold*, τ, or if *both* L_0 and L_1 exceed τ, we will just merge L_0 with its relevant data blocks using the predictive transfer described above, i.e., a situation similar to Fig. 3. The only remaining case is when L_0 exceeds τ, but L_1 does not. Then we will instead merge L_1, but move L_0 to the back of the log space, just ahead of the newly allocated block that is resulted from the merging of L_1, as shown in Fig. 4.

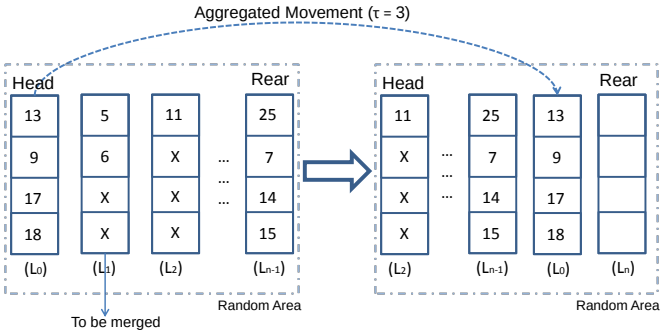

Fig. 4. Aggregated data movement

Here we only consider two blocks at the front of random area. It is because scanning too many blocks will cause performance degradation. Another reason is that we do not want to change too much the FIFO manner of random area.

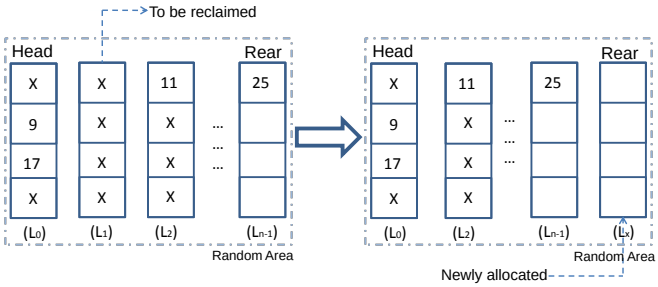

Fig. 5. Early reuse of log blocks

Now that most pages in a log block can be valid, it is also possible that all pages in a block become invalid even though it has not reached the head. FAST and FASTer leave such blocks there until they are merged. To merge a block without valid pages is trivial: erase it and allocate a clean one.

Unlike previous hybrid mapping, ADAPT proposes to *early* reuse the space of blocks that are full of invalid pages. Note that each log block's number of valid pages is recorded. If this number of a block decreases to zero, ADAPT will immediately remove it from log space and allocate another clean one to the random area. Fig. 5 gives an example. L_1 is detached and L_x is appended to bring in more free space. The performance then benefits from the early reuse of log blocks (ERL), because merging the current head like L_0 in Fig. 5 can be delayed since free space has been made by L_x. Yet the time for blocks behind L_1 to be merged is not affected by the removal of L_1. Take, for example, L_2, which would be merged when two blocks that are allocated for merging L_0 and L_1 are exhausted at the rear. Now it only waits for L_0's replenishing block to be used, but L_x has been introduced for L_1. Thus, L_2 still has to wait for two blocks' exhaustion to be merged. However, the effect of ERL cannot be very significant. The first reason is that switch among multi-tasks will scatter data across log space and there is little chance to find a block without any valid data. Secondly, one log block of pages help marginally to satisfy continuous allocation requests.

To support ADAPT's aggregated data movement and ERL modules, we need to know the number of valid pages in each log block. We assume that this is also stored in a table in RAM. It is possible to store this information in the spare area of blocks, but the access latency will be longer. The space requirement for such a table is also comparatively low. A block typically comprises of 64 pages, and one byte is sufficient to store the total number of valid pages. Since log blocks typically take up 3% of the overall capacity, an xGBytes flash SSD with the standard 128KBytes large block configuration, will require a total of $0.24x$KBytes of RAM to store the per-block valid page counts. A 64GBytes SSD, for instance, will require a table of less than 16KBytes. This is quite small compared to the main block mapping table which is about 2MBytes (assuming each entry has a 3-byte physical block number and one byte for mapping status).

The ADAPT FTL scheme that we propose consists of the online adaptive partitioning of log space, the predictive transfer, the aggregated data movement, and ERL described above. We shall now give more details about the implementation of ADAPT, especially during the full merge to make decisions.

E. Merge or Move Decision Procedure

Algorithm 2 outlines the decision making procedure that is executed in the merge procedure in ADAPT. It first locates the victim log block to be merged, the new head and rear of random area, as well as the numbers of valid pages of the victim and the head block (line 2 to line 6). At line 7, it checks whether aggregated movement needs to be performed. If so, it will append the block to the rear of the random area (line 9), set the corresponding flag (line 8), and attempt to merge the next block to create the space (line 10 to 11).

If the condition for aggregated movement is not met, each valid page of the log block will be checked (line 14 to 28). At line 19, the HAT is queried to see whether the page has been

Algorithm 2: Merge decision procedure in ADAPT

```
1  begin
2      victim := GetHeadofRandArea(void);
3      head_log_blk := RenewRandAreaHead(void);
4      rear_log_blk := GetRearofRandArea(void);
5      vp_no_vic := GetValidPageNo(victim);
6      vp_no_hd := GetValidPageNo(head_log_blk);
7      if ((vp_no_vic ≥ τ) && (vp_no_hd < τ)) then
8          AG_MOV := true;
9          Insert(rear_log_blk, victim);
10         new_head := RenewRandAreaHead(void);
11         MergeBlock(head_log_blk, AG_MOV);
12         return;
13     end
14     else
15         page_no := 0;
16         while (page_no < BLOCK_SIZE) do
17             state := GetPageState(victim, page_no);
18             if (state == VALID) then
19                 flag := IsHATHit(victim, page_no);
20                 if (flag == true) then
21                     MoveData(victim, page_no,
                       rear_log_blk);
22                 end
23                 else
24                     MergePage(victim, page_no);
25                 end
26             end
27             page_no++;
28         end
29     end
30     return;
31 end
```

accessed recently. If so, it will be moved to the block at the rear (line 21). Otherwise, it will be merged with corresponding data block (line 24).

Note that in the implementation, we have two levels of merges, one at the block level (line 11) and another at the page level (line 24). This adds flexibility to resource management at runtime.

IV. Experiments

A. Configurations and Assumptions

In this section we will evaluate the effectiveness of ADAPT using a number of workloads. The experiments were conducted using the FlashSim [9] simulator. We implemented FASTer, DFTL [4], WAFTL and ADAPT in FlashSim for comparison. DFTL is a demand-based page mapping scheme. All the parameters of the NAND flash, including the latencies of read, write and erasure which are shown in Table II, were obtained from [6].

To assess ADAPT's performance on various workloads, we utilized 10 traces from three sources. SPC1 is a trace that was collected at a large financial institution [20]. Another trace is a typical OLTP trace from the TPC-C database benchmark [24], TPC-C_20. The other traces are the MSR-series from Microsoft's data centers [18]. The I/O characteristics of these traces have been presented in Table I. We believe these traces are representative of various workload scenarios. The number of write requests in these traces is at least a million. We did not use other shorter traces found in some previous works.

There are also several assumptions in our experiments. Firstly, as with earlier works [15], we assume that the FTL has sufficient DRAM buffer to hold all mapping tables required by FASTer. DFTL and WAFTL were configured to have the same capacity of RAM as FASTer. ADAPT needs less RAM space than FASTer for mapping tables because more log blocks are managed using block mapping for sequential writes. Secondly, the traces used were collected from different machines. Therefore, we had to assigned a capacity configuration to each one based on their access patterns and lengths so that they are of more or less the same scale.

We evaluated each scheme by the elapsed time to complete the simulation in FlashSim, together with counts of write and erasure. FlashSim has a module that accumulates the time caused by reads, writes and erasures as well as bus competitions on the chip. However, because the absolute value varies significantly with each trace, we chose to present the results in a normalized form. For ADAPT, the HAT size was set to 1KB and the aggregated move threshold τ was 56 by default. The length of the interval to measure δ and φ was 4000 write requests. There will be more discussions about the values of τ, δ and φ later. As with previous works, log space was set to be 3% of the overall data capacity [13][15]. The buffer zone of WAFTL also took up 3% of data capacity as originally proposed [25]. Since FAST used one block [13] and LAST used 1/16 of the log space [14] for the sequential area, the lower and upper limits of ADAPT's dynamic sequential area were one block and 1/16 of all log blocks by default, respectively.

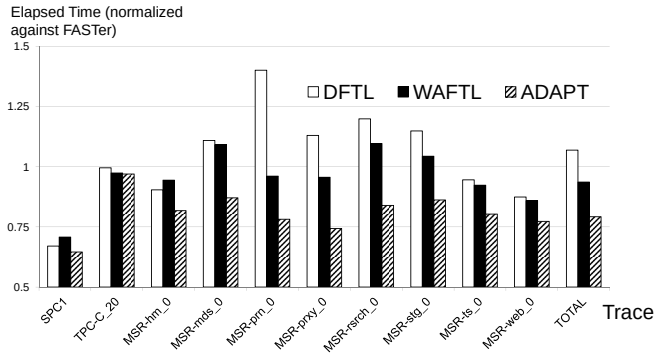

Fig. 6. Normalized performance (elapsed time)

B. Performance Evaluation

Fig. 6 shows the elapsed time for simulating each trace under DFTL, WAFTL and our proposed ADAPT, normalized against that of FASTer. Fig. 7 and Fig. 8 show the erase and

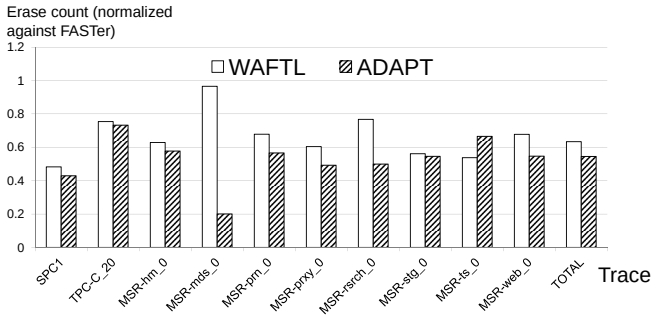

Fig. 7. Normalized erasure counts

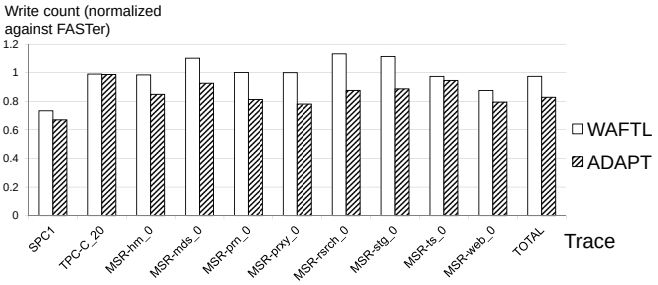

Fig. 8. Normalized write counts

write counts, respectively, of WAFTL and ADAPT normalized against those of FASTer. FASTer, WAFTL and ADAPT combine page mapping and block mapping in a similar way, and utilize parts of flash blocks for buffering. On the other hand, DFTL does page-level mapping, and its overheads include loading and evicting mapping between flash and RAM. Thus our comparisons using write and erase counts exclude DFTL. The rightmost bars in the three figures represent the sum of ten traces' results normalized against the total for FASTer. Let us first compare ADAPT with FASTer and WAFTL since they share similar designs on flash management. It is evident from Fig. 6 that ADAPT outperforms them, consuming 35.4% less time than FASTer at best for the SPC1 workload, and 23.5% less than WAFTL for MSR-rsrch_0 workload. In all, ADAPT is 20.7% and 15.3% on average faster than FASTer and WAFTL respectively.

There is an interesting phenomenon in the case of the TPC-C_20 trace. FASTer was developed for OLTP applications. Even so, from Fig. 6, we can see that for the TPC-C_20 trace, ADAPT is still marginally better than FASTer. Since I/O requests are predominantly random and small in this OLTP workload, with access severely skewed, there is little opportunity for ADAPT's mechanisms to exact its maximum impact.

Fig. 7 and Fig. 8 are the results for write and erase counts, respectively. From the two figures we can see that in every trace, ADAPT performs less writes and erasures than FASTer and WAFTL. However, there is something interesting to note in the results. In Fig. 7 we can see that for MSR-mds_0, FASTer needs four times more erasures than ADAPT, but the results in Fig. 6 show FASTer is merely 13.9% slower than ADAPT. This is because the performance is mainly dominated by the number of write operations at runtime. For MSR-mds_0 in Fig. 8, ADAPT has only 7.4% less writes than FASTer. Consequently, the overall improvement of performance is not as significant as the reduction on erase counts would suggest. The situation is also the same for WAFTL and ADAPT executing MSR-ts_0. WAFTL has less erasures, but slightly more writes making it worse than ADAPT. Moveover, WAFTL was designed to flush all data in the buffer zone because it wants to take advantage of the integration of logical blocks that are buffered. However, to move all data is not trivial, and it will take too many writes and erasures. On the other hand, ADAPT attempts to leave data in the buffer for a longer time to avoid unnecessary movements.

We also implemented a state-of-the-art page mapping scheme. Since lazyFTL was said to have a similar performance to DFTL [17], we have selected DFTL for comparison. The results are also presented in Fig. 6, normalized against those of FASTer. From the figure, we can see ADAPT is faster than DFTL by as much as 44.2% for the case of MSR-prn_0. Unlike FASTer or WAFTL that considers characteristics of one or more types of workload, DFTL merely loads page-mapping entries to RAM on demand, and handles sequential and random writes in the same way. For MSR-prn_0, 9.46% of its requests would write more than 64KB (32 pages) at a time. ADAPT could respond well to such access patterns. However, these continual large writes from multi-tasks would cause DFTL to reclaim blocks frequently for clean pages as well as load and evict mapping entries, thereby badly degrading overall performance.

TABLE III
PREDICTION HIT RATES AND AGGREGATED MOVES

Trace	Prediction Hit Rate	Aggregated Moves
SPC1	79.50%	132
TPC-C_20	100.00%	0
MSR-hm_0	95.68%	233561
MSR-mds_0	96.49%	1727
MSR-prn_0	99.93%	124608
MSR-prxy_0	99.72%	8323
MSR-rsrch_0	98.75%	2050
MSR-stg_0	93.24%	1045
MSR-ts_0	95.16%	1165
MSR-web_0	96.99%	5408

Table III shows the prediction hit rates and the number of aggregated move for each trace. The hit rate is high for most traces. For SPC1, even with a relatively lower hit rate, good performance can still be achieved by the cooperation of all modules in ADAPT. From Table III, we can also see there is no aggregated movement for the OLTP TPC-C_20 trace, and the prediction hit rate is 100%. This agrees with our earlier analysis in Section 2.

Aggregated movement and predictive transfer affect each other. If an aggregated move is performed on a block, then its pages will stay longer in the log space. This will result in the block at the rear of the random area having many valid pages. If the heuristics are correct, many of the pages will

be accessed again soon, leaving the remaining pages to be processed by predictive transfer when this block again reaches the head of the random area. Therefore, aggregated movement and predictive transfer complement each other.

C. Effects of Log Space Capacity

The impact of the log space capacity was also investigated. We performed experiments where the log space was provisioned as 3%, 5% and 10% of the overall capacity. The results are shown in Fig. 9. We normalize the results for provisioning 5% and 10% of space as log space against that for 3%. It can be seen that generally performance improves with larger log spaces. However, the extent of effect varies. For some traces, the impact on performance is significant, but for others, such as TPC-C_20, it is not. We believe this is particularly noteworthy for SSD users to utilize resources.

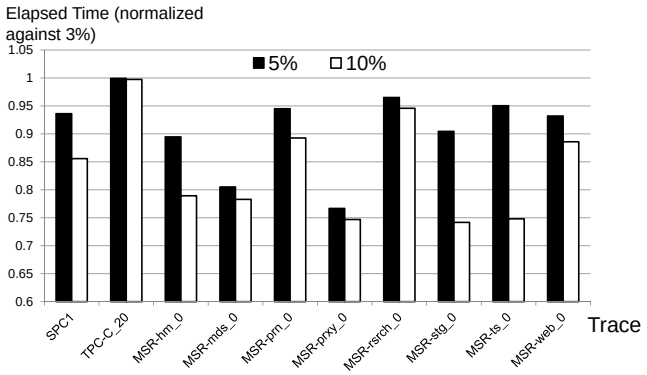

Fig. 9. Effects of different log space capacities

D. Effects of Log Space Partitioning

We also did experiments to verify the effects of adaptively adjusting the partitioning of the log space. In Fig. 10, ADAPT and ADAPT-sw had the same configuration including predictive transfer and aggregated movement except that ADAPT-sw used only one log block for sequential writes, which is the same as FAST and FASTer. All results are normalized to those of ADAPT-sw. From Fig. 10 we can see ADAPT can be faster by as much as 31.9% in the case of SPC1. However, TPC-C_20 is still special because it has almost no sequential writes as shown in Table I.

Fig. 11 shows the result for different thresholds used in the identification of sequential writes. We used three configurations. The first, ADAPT-2, has a threshold of 2 pages (4KB) which is the same as LAST. The threshold of the second, ADAPT-32, is 32 pages (64KB). The last one is the full version of ADAPT that adaptively adjust the threshold based on δ. The lower and upper bounds of ADAPT are set to 2 and 32, respectively. Results of the ADAPT-32 and ADAPT are normalized against those of ADAPT-2 and presented in Fig. 11. From Fig. 11 we can see that ADAPT is faster than ADAPT-2 and ADAPT-32 most of the time. But with some workload like MSR-ts_0, ADAPT had to spend 12.7% more time to finish the trace. We analyzed MSR-ts_0, and

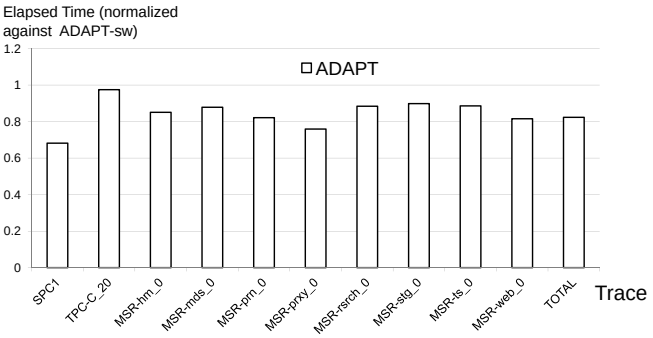

Fig. 10. Performance impact of log space partitioning

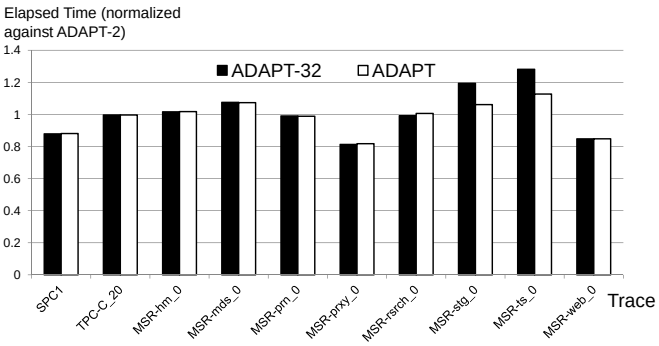

Fig. 11. Impact of different sequential write identification thresholds

found that the feedback in current interval does not accurately reflect the access behaviors. The results in next subsection will address this by showing how performance can be significantly improved with longer intervals.

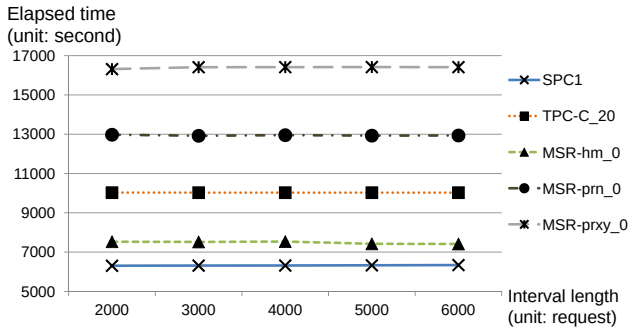

Fig. 12. The effects of the interval length (A)

E. Effects of the Interval Length on Adaptation

ADAPT needs to observe and calculate δ and φ in an interval. By default, we used an interval of 4000 requests in the experiments. We also experimented with intervals of 2000, 3000, 5000 and 6000 requests. Their results are shown in Fig. 12 and Fig. 13. From the results, we can see that in most cases, the length of the interval hardly affects the performance. For MSR-stg_0 and MSR-ts_0, however, their results may fluctuate a little more. That means current configuration is too short to reflect the online behaviors. By prolonging the

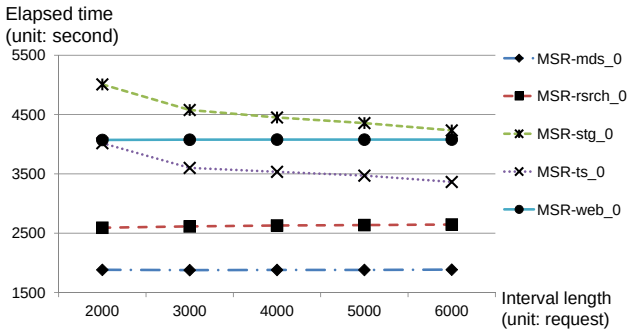

Fig. 13. The effects of the interval length (B)

interval, better performance can be achieved, as shown in Fig. 13. This agrees with results in Fig. 11.

F. Effects of HAT Size

The HAT is an important data structure for ADAPT. Fig. 14 presents four results of each trace when the size of HAT was configured to be 512, 1024 (1K), 1,536 and 2,048 (2K) bytes. The results for 512 bytes are used to normalize the other cases. It can be concluded from these results that the optimal size of the HAT depends on the workload. Recall that the HAT is used to record the recent write history which is then used for prediction. Obviously, keeping too long or too short a history may result in wrong predictions. If the HAT is too big, it would store outdated access records, causing pages that should be merged immediately to stay too long in log space. If the HAT is too small, the prediction would not get a full view of the locality, and unnecessary merges may be performed.

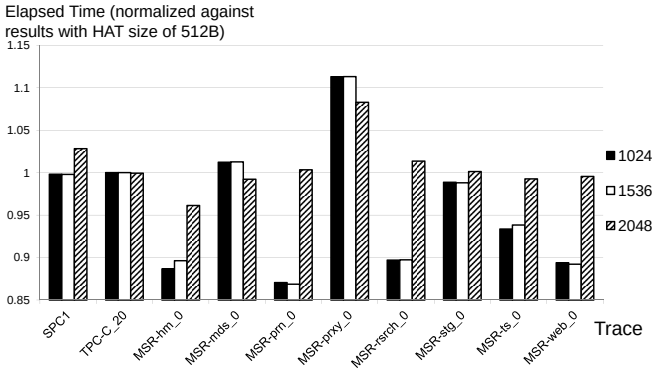

Fig. 14. Effects of different HAT sizes

The performance of ADAPT on TPC-C_20 trace changes slightly with different HAT sizes. This can also be attributed to its access patterns. Due to the skewness of the accesses in the OLTP workload, a small HAT suffices. It therefore makes little difference in enlarging the HAT. The results also suggest that due to the differences in locality, the size of the HAT should be tuned for each workload.

G. Tuning of Aggregation Threshold

The threshold τ to trigger aggregated data movement is an important parameter in ADAPT. For ease of reading, we have separated the results as Fig. 15, Fig. 16 and Fig. 17.

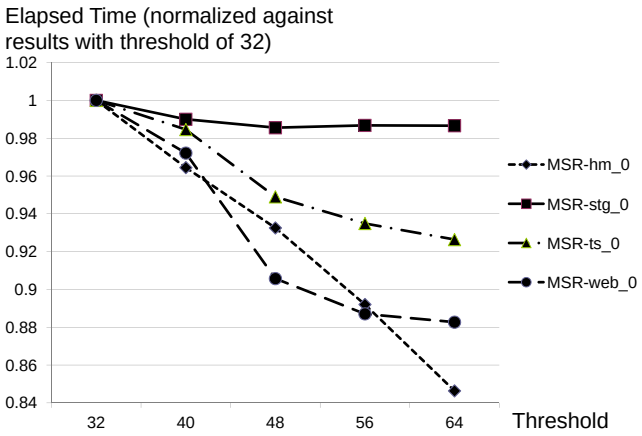

Fig. 15. Performance of aggregated movement (A)

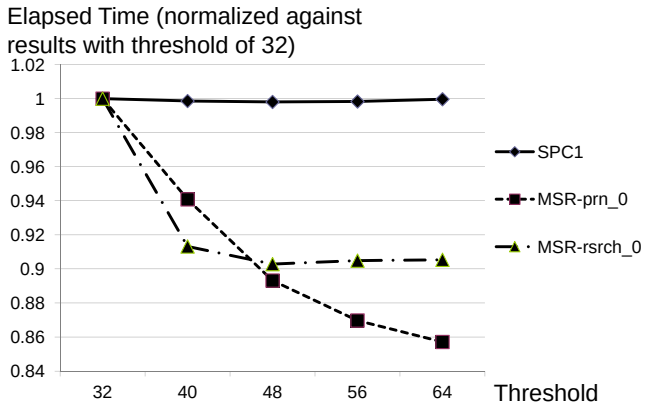

Fig. 16. Performance of aggregated movement (B)

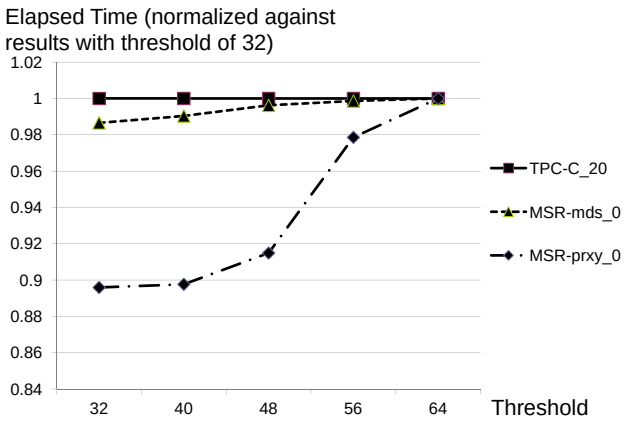

Fig. 17. Performance of aggregated movement (C)

In accordance with [6], each block has 64 pages in our simulations. In general, when the number of valid pages in the block to be merged reaches the aggregated move threshold, i.e., τ, the block would be moved, and the one next to it will be merged instead. If τ equals to 32, aggregated move will be performed if 50% or more pages of the block are valid. If it is 64, all the pages in a block will have to be valid in order for an aggregated move to be activated. Fig. 15, 16 and 17 show the impacts of various values of τ on performance.

We know that TPC-C_20 has no aggregated movements from Table III. For other traces, we can see from the figures that for some of them, including MSR-ts_0, MSR-web_0, MSR_hm, and MSR-prn_0, the results are better with a higher τ. For others, such as SPC1, MSR-rsrch_0, MSR-mds_0 and MSR-stg_0, τ does not affect performance. However, for the MSR-prxy_0 trace, performance degraded with larger τ. Again, we attribute this to the access patterns of the traces. Traces in the first category generally have more valid pages in the log block to be merged than others. Hence, a higher τ improves the performance. For traces in the second category, the number of valid pages is moderate and stays fairly constant throughout the execution, and different thresholds showed little impact.

We analyzed MSR-prxy_0, and found out more about its access behaviors. As discussed in Section 2, requests with 2 pages (4KB) are considered to be small. But in MSR-prxy_0, there is a huge number of requests that are even smaller. 77.8% of the requests in MSR-prxy_0 only write to one page and data in these pages are frequently updated and scattered. Thus the log block to be merged may have dozens of valid pages. However, a higher τ gives aggregated movement little chance to show off its advantage, leaving valid pages to be processed by predictive transfer. A larger log window would absorb more small requests. From Fig. 9 we can see it is MSR-prxy_0 that improves the most with larger log space.

H. Impact of Early Reuse of Log Blocks

We also investigated the impact of the early reuse of log blocks. Experiments were conducted with the ERL module enabled or disabled, respectively. When it is disabled, if a log block has no valid data at runtime, it would be left in the log space until it reaches the head of the log space. Fig. 18 shows the results. ADAPT-nd is the configuration without ERL module. The results agree with our expectation in Section 3 that ERL may help marginally. Compared to ADAPT-nd, ADAPT could require less time to finish each trace, at most 4.3% with the case of MSR-web_0.

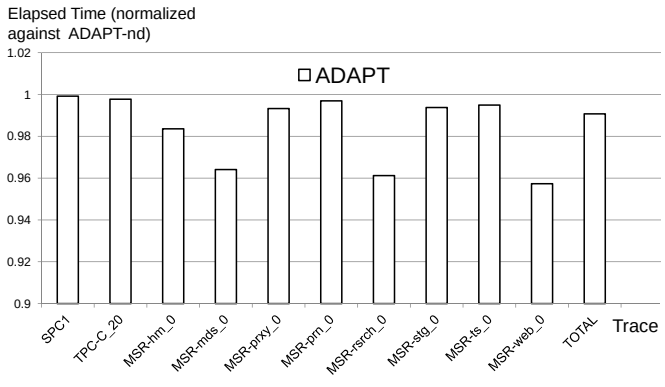

Fig. 18. The impact of early reuse of log blocks

V. Related Works

Page mapping and block mapping are two basic mapping schemes. The mapping table used for page mapping is significantly larger. To alleviate this issue, DFTL [4] was proposed by caching parts of entire table that are in use in RAM. This RAM space can be managed using Least-Recently-Used (LRU) eviction. DFTL achieved better performance compared to FAST and LAST. The latest LazyFTL [17] is similar in design and performance as DFTL, but its focus is on data reliability.

Block mapping lacks flexibility due to its coarse granularity. Write requests to a page will cause block-level copying because of out-of-place updating. Recently, an improved block mapping scheme, DAC [21], was proposed. It is similar to DFTL, and caches mapping entries and pages on-demand in RAM in two levels.

Mapping schemes based on other granularities have also been developed. One example is a set-based mapping strategy [3]. Each set contains multiple blocks. Logical sets are mapped to physical sets with another table used to store the mapping of logical block to physical block in a set. There are also other schemes that try to strike a balance between page mapping and block mapping, such as Janus-FTL [11].

There are many hybrid mapping schemes. The log space can be viewed as a cache of data blocks. FAST, FASTer and ADAPT are fully associative, and BAST is block associative. More complicated N-way associative schemes of log blocks have also been designed. Physical blocks are grouped together, and they are associated to a set of log blocks where the size of the set may be dynamically changed at runtime [19][10]. Other mapping schemes, such as the superblock [7] and KAST [2], give more considerations to garbage collection and real-time systems, respectively.

Recently, content-aware FTLs that attempt to reduce duplicate writes have been proposed too. Examples include CAFTL [1] and CA-SSD [5]. They can potentially benefit from improvements in content detection and reduction.

VI. Conclusion

Address mapping of the flash translation layer is central to the performance of flash-based devices. In this paper, we proposed *Aggregated Data movement Augmenting Predictive Transfers* (ADAPT), a hybrid mapping FTL scheme that adjusts to various workloads by exploiting their access behaviors and temporal locality. ADAPT can handle both sequential and random writes efficiently by dynamically tuning the partitioning of the two areas of log space that process the respective types of writes. To do so, ADAPT collects statistics on how log blocks in the sequential and random area are used, and then utilizes these statistics to adjust its parameters. ADAPT also explores the locality to reduce unnecessary data movements in full merges. It employs a prediction mechanism to decide whether a log page should be merged, or given a second chance. In addition, ADAPT records the number of valid pages in each random log block at runtime. If a block to be merged has more than a certain threshold of valid pages, the entire block would be kept in the log space. However, if a block is found to have no valid page, it will be immediately reclaimed and replaced even if it has not

got its turn for merging. Our experiments show that ADAPT can outperform FASTer by as much as 35.4% with a modest additional RAM space requirement of less than 16KBytes for a 64GBytes SSD. ADAPT is also faster than DFTL and WAFTL by as much as 44.2% and 23.5% respectively. We believe by being adaptive to various workloads, ADAPT will enhance the general performance of SSDs.

REFERENCES

[1] F. Chen, T. Luo, and X. Zhang, "CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960481

[2] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 507–512. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874620.1874745

[3] Y.-S. Chu, J.-W. Hsieh, Y.-H. Chang, and T.-W. Kuo, "A set-based mapping strategy for flash-memory reliability enhancement," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 405–410. [Online]. Available: http://portal.acm.org/citation.cfm?id=1874620.1874717

[4] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2009, pp. 229–240.

[5] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960482

[6] M. T. Inc., "Small-block vs. large-block nand flash devices. technical report (tn-29-07): Small-block vs. large-block NAND flash devices." Tech. Rep., May 2007.

[7] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*. New York, NY, USA: ACM, 2006, pp. 161–170.

[8] 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*, vol. 48, pp. 366–375, 2002.

[9] Y. Kim, B. Tauras, A. Gupta, and B. 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 125–131.

[10] D. Koo and D. Shin, "Adaptive log block mapping scheme for log buffer-based FTL (flash translation layer)," in *IWSSPS 2009: International Workshop on Software Support for Portable Storage*, 2009.

[11] H. Kwon, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Janus-FTL: finding the optimal point on the spectrum between page and block mapping schemes," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '10. New York, NY, USA: ACM, 2010, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1879021.1879044

[12] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory SSD technology for enterprise database applications," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 863–870. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559937

[13] 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 Trans. Embed. Comput. Syst.*, vol. 6, no. 3, p. 18, 2007.

[14] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.

[15] S.-P. Lim, S.-W. Lee, and B. Moon, "FASTer FTL for enterprise-class flash memory SSDs," *Storage Network Architecture and Parallel I/Os, IEEE International Workshop on*, vol. 0, pp. 3–12, 2010.

[16] S. E. C. Ltd, "NAND flash spare area assignment standard," Tech. Rep., April 2005.

[17] D. Ma, J. Feng, and G. Li, "LazyFTL: a page-level flash translation layer optimized for NAND flash memory," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989325

[18] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *Trans. Storage*, vol. 4, pp. 10:1–10:23, November 2008. [Online]. Available: http://doi.acm.org/10.1145/1416944.1416949

[19] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 4, pp. 1–23, 2008.

[20] S. Performance Council, "Storage Performance Council (SPC) storage traces," http://traces.cs.umass.edu/, December 2009. [Online]. Available: http://traces.cs.umass.edu/

[21] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "Demand-based block-level address mapping in large-scale nand flash storage systems," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 173–182. [Online]. Available: http://doi.acm.org/10.1145/1878961.1878991

[22] M. Sanvido, F. R. Chu, A. Kulkarni, and R. Selinger, "NAND flash memory and its role in storage architectures," vol. 96, no. 11, pp. 1864–1874, Nov. 2008.

[23] G. Shim, Y. Park, and K. H. Park, "A hybrid flash translation layer with adaptive merge for SSDs," *Trans. Storage*, vol. 6, no. 4, pp. 15:1–15:27, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1970338.1970339

[24] B. trace distribution center, "TPC-C database benchmark traces," http://tds.cs.byu.edu/tds/, 2001. [Online]. Available: http://tds.cs.byu.edu/tds/

[25] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada, "WAFTL: A workload adaptive flash translation layer with data partition," *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, vol. 0, pp. 1–12, 2011.