TreeFTL: Efficient RAM Management for High Performance of NAND Flash-based Storage Systems

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

***Abstract*—NAND flash memory is widely used for secondary storage today. The *flash translation layer* (FTL) is the embedded software that is responsible for managing and operating in flash storage system. One important module of the FTL performs RAM management. It is well-known to have a significant impact on flash storage system's performance. This paper proposes an efficient RAM management scheme called *TreeFTL*. As the name suggests, TreeFTL organizes address translation pages and data pages in RAM in a *tree* structure, through which it dynamically adapts to workloads by adjusting the partitions for address mapping and data buffering. TreeFTL also employs a lightweight mechanism to implement the least recently used (LRU) algorithm for RAM cache evictions. Experiments show that compared to the two latest schemes for RAM management in flash storage system, TreeFTL can reduce service time by 46.6% and 49.0% on average, respectively, with a 64MB RAM cache.**

I. Introduction

NAND flash memory is the ubiquitous non-volatile memory today. It is widely utilized for secondary storage of embedded systems. It is also starting to replace magnetic hard disks of general-purpose computing systems in the form of solid state drives (SSDs). The management of flash memory is performed by a firmware called the *flash translation layer* (FTL). The FTL provides the traditional block-device interfaces to file systems, and is responsible for all functions of flash management.

The performance of flash is most influenced by two characteristics of NAND flash: its units of access and *out-of-place updating*. Read and write operations of NAND flash are conducted in the unit of a *page*, which is 2KB or more [6]. However, a page cannot be rewritten unless the *block* it is in is erased first. A block consists of multiple pages, and is the unit for erase operations. Hence, to update the data in a page has to be done in an out-of-place way: invalidate the current page and put the updated copy in an erased one.

Upon a read or write request, logical addresses from the file system must be translated to the physical addresses of the flash substrate. Based on the concepts of page and block, there are two basic ways to perform this translation, namely, *page-level mapping* and *block-level mapping*. Read and write requests are serviced with the aid of an internal SRAM [4] [14], DRAM [15], or non-volatile RAM [7] [10] cache that is equipped in flash devices. The byte-addressable RAM cache plays an important role in flash storage system. NAND flash can take advantage of RAM's access flexibility by maintaining entries of the address mapping table. All the mapping entries may be grouped and stored in the *translation pages* [4] [14] of flash memory (as opposed to *data pages* that store real data). On the other hand, the access latency of RAM is much shorter than NAND flash. So caching mapping information or data in RAM can significantly improve the overall performance. Many schemes have been proposed on how to utilize the RAM cache. The emphasis has either been on keeping information for address *mapping* [4] [14], or on *buffering* recently accessed data pages [8] [3].

Recently, there are proposals on how to jointly use the RAM space for both mapping caching and data buffering. JTL [5] proposed a fixed partitioning of the RAM cache for mapping and data. It aims to put the most recently used data pages in RAM. However, JTL's static partitioning makes it inflexible to changing workloads. Shim et al. [15] proposed the *Adaptive Partitioning Scheme* (APS) [15] that dynamically partitions the RAM space. It collects statistical data periodically, and calculates the potential benefits of adjusting either partition. Yet this makes APS less responsive to online workload changes.

In this paper, we propose a novel RAM management scheme that is simple but efficient. We call it *TreeFTL*. The main ideas of TreeFTL are as follows:

- TreeFTL does not cache single mapping entries like previous schemes. It maintains translation pages and data pages in RAM through a tree-like structure.
- A lightweight strategy is devised for LRU eviction. It selects victims at a coarse granularity, and our experiments show that the gain it brings in is significant.

TreeFTL achieves the dynamic partitioning for address mapping and data buffering inherently using the tree structure. We conducted experiments to evaluate the effectiveness of TreeFTL with FlashSim [4], and found that with a 64MB RAM cache, compared to APS and JTL, TreeFTL spends, on average, 46.6% and 49.0% less on service time, respectively.

The rest of this paper is organized as follows. Section 2 shows background and classic algorithms on RAM management. Section 3 details our TreeFTL, including the management algorithm, lightweight victim selection strategy and relevant issues. Section 4 describes evaluations to verify the effectiveness of TreeFTL. Section 5 will conclude the paper.

II. Background

A. Address Translation

Almost all FTLs use one or a variant of two basic mapping schemes, namely page mapping [1] and block mapping [2].

978-3-9815370-0-0/DATE13/©2013 EDAA

The former is more flexible but requires a large RAM space for its mapping table. DFTL [4] and CDFTL [14] were proposed to load page-level mapping entries into RAM on demand. Block mapping suffers from its coarse granularity since out-of-place updating prevents data from being put into other pages freely in a block. With the increase of flash capacity, the table for block-level mapping also goes bigger, and demand-based mechanisms are needed to lower RAM requirement [13].

Hybrid mapping [9] mixes the two schemes. A logical block is block-level mapped to a *data block*. Some physical blocks construct a *log space* that uses page mapping. Updated data of a logical address are first put into a free page of log space if the page in the data block it is mapped to is used. When there is no free log page left, a victim log block is selected, and valid data from its pages will be *merged* with relevant data blocks. Hence, merging can be a costly procedure.

B. RAM management

To manage RAM is important for FTLs. FTLs use RAM space to hold mapping entries. DFTL loads entries from translation pages on demand. Besides single entries, CDFTL selectively caches translation pages in a two-level structure. Mapping entries form the first level, the *cached mapping table* (CMT). Evicted entries from the CMT are first absorbed by cached translation pages in the second level. The second-level exploits the spatial locality in workloads since neighbouring logical addresses in a same translation page are likely to be accessed. DAC [13] is similar to CDFTL but works at block-level for large-scale flash storage systems.

Data buffering, especially for write requests, is another use of RAM space. A flash page is the buffering unit due to NAND flash's access constraints. BPLRU [8] utilizes a *padding* strategy in hybrid mapping. Unlike RAM management that only writes data to flash upon evictions to free up space, BPLRU may read data from flash to pad a log block and flush it back. Padding can avoid arduous merge procedures. However, reading also costs time. A design named *l-buffer* [3] was proposed to trade off padding for merging, and vice versa.

APS [15] and JTL [5] are two recent proposals that use the RAM cache for mapping and buffering jointly. APS reserves two small areas of RAM as "ghost caches". One is used to keep metadata of evicted mapping entries, while the other maintains the metadata of evicted data pages. They are used to compute the cost caused by not enlarging the cache for mapping and buffering, respectively. Write or read misses in actual cache may hit in ghost cache. A cost-benefit model is built on these hit statistics to estimate the benefits of enlarging either partition. Because APS's estimation is based on values of the past interval, there are delays in adjusting to runtime workload. Moreover, APS uses the *least recently used* (LRU) algorithm at page-level or entry-level to find a victim for evictions in respective parititon. The overhead of frequent LRU selections can be significant since tens of thousands of data pages and mapping entries exist in the RAM.

JTL statically partitions the RAM space into two halves, one for mapping, and the other for buffering. JTL uses a multi-level structure to manage mapping entries. The level n ($n \geq 0$) has 2^n entries. The number of levels is determined by the size of the RAM partition dedicated to mapping, and the size of a single entry. All levels are divided into two groups. As RAM cache is halved for buffering data pages, their mapping entries form Group 0 and take up level 0 to m. Remaining levels fall into Group 1, and their entries correspond to data pages in flash. The entry in the top level corresponds to the most recently used data page. It will move down to level 1 to vacate for the newly accessed entry. An entry at level 1 may need to move to level 2 if no vacancy exists. More moves may follow in next levels. The victim to be moved in each level is randomly selected as entries in the same level is deemed to have similar access recency. When an entry reaches level $m+1$, its cached data page in RAM will be flushed to flash. By doing so, JTL can keep the recently used mapping entries and data pages cached in RAM.

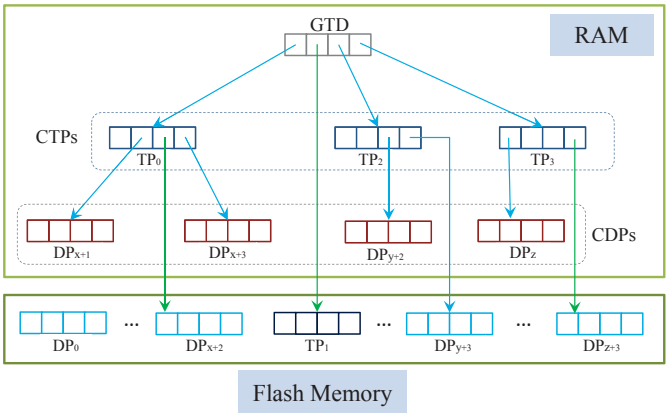


Fig. 1. A Conceptual Structure of TreeFTL

III. TreeFTL

TreeFTL maintains a three-level tree structure in RAM. The first level and second level are used for mapping, while the third level is for caching data pages. TreeFTL dynamically adapts to the runtime workload by adjusting the tree structure. One of TreeFTL's key features is its lightweight LRU victim selection which can significantly reduce spatial and temporal overheads. In following, we shall describe TreeFTL based on page-level address mapping. However, the basic idea of TreeFTL can be easily adopted to a block-level mapping scheme like DAC [13].

A. The Tree in RAM

1) The Three Levels: As is mentioned, all mapping entries of demand-based page-level address translation are stored in the translation pages of flash memory. A structure named *global translation directory* (GTD) is used to record the physical addresses of these pages. The GTD must be resident in RAM as it is the root directory for address translation. Hence, TreeFTL makes GTD the root of its tree structure.

Fig. 1 shows the conceptual tree structure of TreeFTL. In level 1 is the GTD. Level 2 consists of the *cached translation pages* (CTPs), while level 3 holds the *cached data pages* (CDPs). The three levels are connected by unidirectional links.

TABLE I
LATENCIES OF SLC NAND FLASH MEMORY [6]

Read Operation	Write Operation	Erase Operation
25 μs (2KB)	200 μs (2KB)	700 μs (128KB)

TreeFTL treats the RAM cache as a part of storage medium. When a translation page is loaded into RAM, its address in the GTD will be updated to point to a RAM location accordingly. For a mapping entry in a CTP, if the data page is cached in RAM, the record will be the RAM address instead of a physical address in flash memory. CDPs are the leaves of the tree. They are cached upon write requests. TreeFTL emphasizes on write buffer similarly as BPLRU [3], l-buffer [3] and APS [15] do, because write latency is much longer than read latency (Table I). Furthermore, writes may trigger expensive erase operations [8] [15].

Ignoring data buffering, TreeFTL differs from DFTL and CDFTL in that TreeFTL does not cache single entries in a CMT. There are two reasons. First, to load or evict a single entry entails a read or write for a translation page, respectively. Although *batch update* [4] can group evicted entries from a same translation page, it complicates the design, in addition to the space overhead involved. Second, a translation page covers a wider range of consecutive logical addresses, so caching a translation page can benefit from the spatial access locality [14]. CDFTL keeps both single entries and translation pages in its cache. A translation page will be loaded also when one of its entries is fetched into the RAM cache by CDFTL. Thus, eliminating the CMT can avoid duplications, and hence save space. The process of address translation, which will be described below, is also simplified because in CDFTL a miss of the CMT requires consulting the CTPs first.

2) Address Translation With The Tree: Address translation in TreeFTL begins by finding the translation page to locate the desired mapping entry. This can be done by using the logical address as a hash key to look for the RAM location of the relevant translation page, which results in either a hit or a miss. However, in order to show the *growing* of the tree, we will describe the process in another way. The address translation process in TreeFTL can be viewed as a traversal from the root to some leaf of the tree. There are three scenarios for a random write request, as shown in Fig. 2. The first case is when both translation page and data page are cached. In three steps (A-1, A-2 and A-3 in Fig. 2) the data are written to the target CDP. No operation is performed on the flash. The second case is when translation page is cached but data page is in flash. The data page has to be loaded into RAM first. So a read operation (B-3 in Fig. 2) is needed. The third case is neither of the two is in RAM. In such a scenario, two reads have to be conducted (C-2 and C-4 in Fig. 2). So this case is the most costly.

Any CDP or CTP that has been selected as the eviction victim will be flushed back to flash memory. More details will be given in next subsection.

TreeFTL services read requests in a slightly different way. When a read request comes, the translation page will be loaded if it is not already cached. For the target data page, however, it will not be loaded into RAM if not cached. Instead, the flash page is read directly from flash and the data are returned to the file system then.

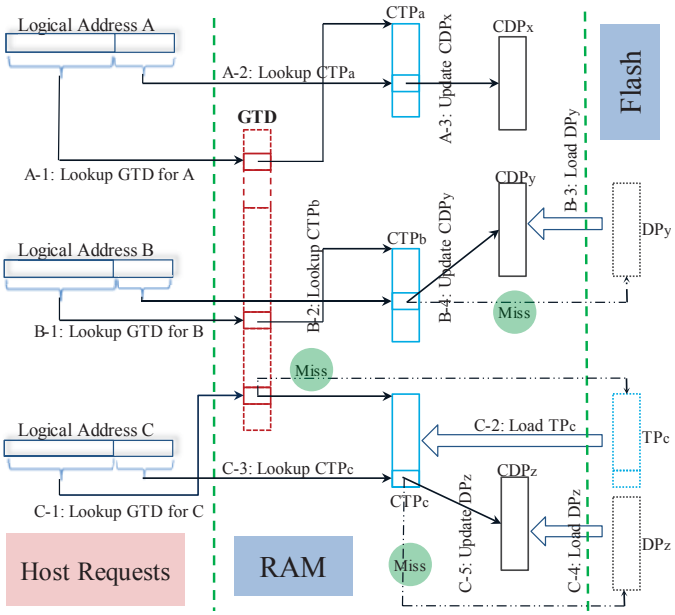


Fig. 2. Address Translation Process in TreeFTL

Note that the write or read request mentioned above is random access request. TreeFTL deals with sequential requests in a "write-through" manner. Data are written to or read from flash memory in a bulk, and RAM cache is bypassed. This is similar to what JTL does [5]. It is based on the assumption that data which are sequentially requested are likely to be infrequently accessed. There are many methods to identify a request to be random or sequential. For example, deciding based on the access size is a simple but effective approach. TreeFTL deems a request to be sequential if it tends to access more than half a block, i.e., 32 pages of 64KB data as in [6].

B. Lightweight Pruning

When RAM space is exhausted, a victim has to be selected and evicted. The victim ought to be the one that is the least recently used (LRU). APS performs LRU selection at the level of entries and pages among cached mapping entries and data pages, respectively. JTL's multi-level structure helps it to find the LRU mapping entry or data page easily as less frequently accessed ones are moved down from RAM to flash memory. However, both APS and JTL suffer from LRU selection. Assuming that all 64MB of a RAM cache is used for APS's data buffering, there would be 64MB/2KB = 32768 data pages in total. It is not trivial to find the LRU page each time in such a large number of pages. For JTL, its multi-level structure may have to be adjusted on each arriving request.

TreeFTL exploits its tree structure and utilizes a lightweight victim selection policy. Since TreeFTL uniformly caches pages which are just nodes in the tree, the eviction process is like *pruning* the tree. To do so, TreeFTL introduces the concept of a *caching group* (CG). A CG is a group which includes a CTP and its relevant CDPs. A CG is just a branch (sub-tree) of the tree. There are three CGs in Fig. 3.

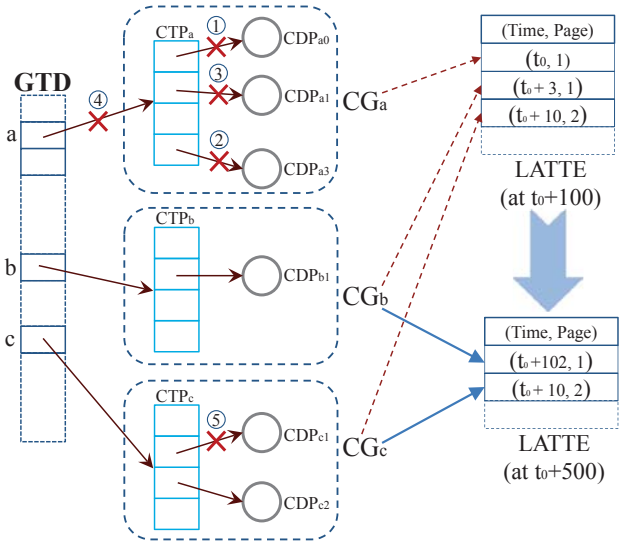


Fig. 3. The Sketch of TreeFTL's Victim Selection

TreeFTL maintains a hash table called *Last Access Time Table for Eviction* (LATTE) that records the last access time for each CG. Note that while we use timestamp as a metric to implement LRU, other implementations of LRU can also be used. The hashing key of LATTE is the number that identifies each CTP. This number subsequently identifies a CG. Each entry in LATTE is a two-tuple. The first element is the time when any CDP of that CG is last accessed. The second element is the page number of the last accessed CDP, which ranges from 0 to 511 (2KB a translation page and 4B for a mapping entry [4]). Hashing enables LATTE to be swiftly updated after access requests. A sketch of the LATTE is shown in Fig. 3.

Victim selection is performed upon an eviction request. It first finds the *victim CG* that has the smallest timestamp. In Fig. 3, at time $t_0 + 100$ the victim CG is CG_a. Then the selection inside a CG starts. The CDP that has the smallest offset in a CG will be the victim page. If it is the one recorded in the LATTE entry, however, it will be skipped unless there is no other CDP left. In Fig. 3, a circled number is the eviction sequence of a CDP. CDP_{a0} is firstly evicted and CDP_{a3} will follow. On the next eviction request, CDP_{a1} will not be skipped again since it is the last CDP of CG_a. This way all the CDPs of CG_a would be pruned. If no new data page joins CG_a (otherwise the LATTE will be updated) before next eviction request, CTP_a will be flushed back to flash as a victim page, and CG_a's entry in the LATTE will be removed.

TreeFTL's pruning policies can be summarized as follows:

- With the LATTE, the LRU selection is conducted at the level of a CG, not page.
- CDPs are preferred for eviction. The one that is the most recently accessed in a CG, i.e., the recorded one in the LATTE, would be the last to be picked.
- If a CG has no CDP left, and it has the eldest timestamp in the LATTE, the CTP will be evicted.

The first rule makes TreeFTL "lightweight" as the granularity of CGs is much coarser than that of CDPs, since a CTP can point to hundreds of CDPs. Spatial locality dictates that consecutive logical pages are likely to be accessed in a short interval of time. Hence, the timestamp of the last accessed CDP can be used to approximate a group's recency. This approximation saves RAM space and reduces processing time. It certainly suffers from the lack of detailed information about each CDP. However, our experiments (in Section 4) show that such trade-off is worthwhile to make.

The second rule states to evict CDPs is preferred. It is because a CDP is a leaf of the tree, and a CTP yet connects to tens or even hundreds of CDPs. Moreover a miss of a translation page needs two read operations, while a miss of a data page requires only one read. In addition, the CDPs recorded in LATTE should be the last one to be evicted. Based on temporal locality, this CDP is the one that is the most likely to be accessed again. Other CDPs will be selected in the sequence of their offsets in their CTP.

The third rule dictates when a CTP is to be flushed back to flash. When all CDPs are removed from a CG, the CTP can be evicted. However, TreeFTL's eviction is based on demand. Only when a request is raised for free space, will TreeFTL act. This also gives a CTP a second chance to stay for a while.

In the worst case, each CG just has a CTP and a CDP. The temporal overhead of the lightweight selection would be half that of a page-level selection, since a timestamp is used for two pages (a CDP and a CTP). The spatial overhead of LATTE is the maximum in this case too. It is less than that of a page-level strategy. The second element of a two-tuple in LATTE needs less space than a timestamp, and a two-tuple stands for two pages while two pages of a page-level strategy need two timestamps. Such extreme case is rare. Since a CG may have many CDPs, at most 512, it can be expected that the overhead of maintaining and searching at CG-level is significantly less than that of a page-level policy.

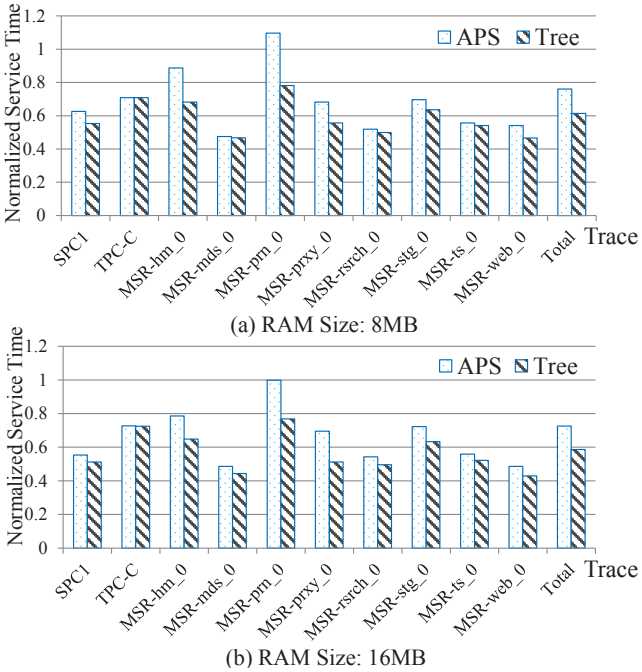
C. Partitioning and RAM Space Utilization

Adaptive partitioning is inherently achieved by TreeFTL. The tree naturally grows or is pruned on access requests. The partitions for mapping and buffering are accordingly adjusted.

A possible issue of TreeFTL is the utilization of RAM space. A CTP has many entries, and usually not all of them are connected to CDPs. So unused "holes" scatter across CTPs. The benefits of caching a translation page on spatial locality have been addressed in Section 3.2. In terms of RAM utilization, caching a page for a requested entry risks taking up more space, but the potential use of other entries in this page can save valuable time. The lower utilization of RAM cache is more likely to be caused by outdated mapping information and data pages. A RAM management module ought to efficiently identify and move them out.

D. Reliability and Garbage Collection

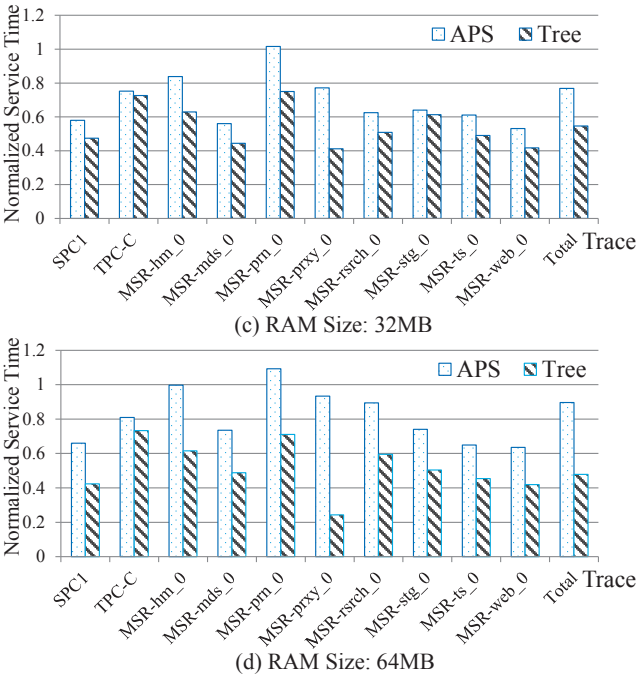
Reliability is an important issue of data storage, especially when RAM is used as a storage medium. DRAM or SRAM is volatile memory, and would lose data if power supply is unexpectedly off. This problem has been addressed by using non-volatile memory [4] [7] [10]. A backup battery can



(a) RAM Size: 8MB

(b) RAM Size: 16MB

Fig. 4. Normalized Service Time for Traces (1)



(c) RAM Size: 32MB

(d) RAM Size: 64MB

Fig. 5. Normalized Service Time for Traces (2)

otherwise be equipped. Moreover, CTPs and CDPs can be copied to flash memory when the storage system is idle.

Garbage collection is another important area in NAND flash management due to out-of-place updating, and the time-consuming write and erase operations. If data cached in RAM can be frequently updated, it will alleviate the pressure on the garbage collection module since less data will be sent to flash. This is an area we plan to work on.

IV. Performance Evaluation

A. Experimental Setup

We evaluated TreeFTL using FlashSim [4] simulator running on a Linux 64-bit system to simulate a 32GB NAND flash storage system. The compiler was GCC 4.6. The traces we used are from three public sources. SPC1 is from Storage Performance Council (SPC) [12]. TPC-C was collected within TPC-C database benchmark [16]. MSR traces are from Microsoft data centers [11]. We believe that they represent various workloads in the real world. The parameters of the flash memory used for the evaluation, as shown in Table I, were obtained from a recent datasheet [6]. In previous works, RAM access time was either ignored [3] or unclear [5] [15]. We assumed one RAM operation over a 2KB page costs $2\mu s$, which is the same as in [14]. The RAM capacity were multiply configured, and will be shown below.

We implemented APS and JTL as comparisons to TreeFTL. Their implementations are referred to as `APS`, `JTL` and `Tree`, respectively. `APS`'s interval length was 1000 requests which is the same as in [15], and the two partitions had equal capacity in the beginning. The metric to measure access performance is the *service time* needed to process a trace using a management scheme. For a scheme, the shorter the service time is, the higher its performance.

B. Experimental Results

Fig. 4 and 5 present the results for each trace with the RAM cache configured as 8MB, 16MB, 32MB and 64MB. The rightmost bar in each diagram is the sum of the results of all ten traces. Because service time of ten traces varies over a wide range, we normalized values of `Tree` and `APS` to that of `JTL`. From Fig. 4 and 5 we can see `Tree` always has the least service time under all four configurations, which means it consistently achieves the highest performance. Take the 64MB RAM cache for example. `Tree`'s average time over all traces is less than that of `APS` and `JTL` by 46.7% and 49.0%, respectively. The service time of `Tree` is at best 73.9% and 72.3% less than that of `APS` and `JTL` on MSR-prxy_0.

From Fig. 4 and 5 we can see the gap between results of `JTL` (normalized as 1) and those of `Tree` is significant. It is because JTL statically partitions for mapping and buffering into two halves. Evidently, the buffering partition needs to take up more RAM space. One entity in a translation page is only 4 bytes but an entity in the buffering partition is a 2KB data page. This means that the total number of distinct entities held in JTL's mapping partition far exceeds that in its buffering partition. Misses for data pages cause frequent loading and eviction between RAM and flash, while most of the space dedicated to mapping entries is infrequently used.

From the two figures we can also find that with a small RAM capacity, `Tree` outperforms `APS` marginally. They both can adaptively adjust partitioning. A small capacity cannot effectively cache mapping information or data pages, and evictions and loading dominate the performance. However, owing to the delayed estimation of `APS`, `Tree` is still a little faster. With the increasing of RAM cache, the overhead of LRU selection becomes significant for `APS`.

Table II shows the hit ratios of three schemes for mapping

TABLE II
HIT RATIOS (%) OF THREE SCHEMES

Trace	Address Mapping			Data Buffering		
	APS	JTL	Tree	APS	JTL	Tree
SPC1	97.5	97.5	99.8	65.3	24.9	70.5
TPC-C	99.5	97.5	100.0	99.5	99.1	99.5
MSR-hm_0	92.4	94.2	99.4	45.8	18.1	64.1
MSR-mds_0	95.6	98.1	99.7	64.5	31.7	70.3
MSR-prn_0	70.1	96.1	99.6	48.0	26.3	77.9
MSR-prxy_0	98.9	98.6	99.9	53.2	33.9	92.6
MSR-rsrch_0	97.4	98.2	99.5	59.0	34.0	63.3
MSR-stg_0	97.4	98.2	99.6	61.5	23.1	64.8
MSR-ts_0	95.4	97.1	99.6	59.4	21.6	68.2
MSR-web_0	95.1	98.0	99.6	64.6	20.1	75.3

and buffering with the 64MB RAM cache, respectively. Tree hardly has any miss for mapping. We ascribe this to the spatial locality of a real workload. APS's and JTL's ratios are a little lower because of their policy of caching single entries. In Table II generally the hit ratios of buffering are much lower, which is due to the mentioned asymmetry between mapping information and data pages. Yet JTL suffers more than APS and Tree due to its fixed partitioning.

We experimented with the RAM size as 128MB and 256MB also. They are not presented here due to space limitation. We did not evaluate with an even bigger RAM cache, as excessive RAM space makes it possible to accommodate everything needed in RAM [5]. This will not correctly highlight the effectiveness of the RAM management schemes.

The effect of TreeFTL's lightweight victim selection was also measured. We implemented Tree-PL, which is the same as Tree except that LRU victims are selected at the page-level. Without loss of generality, MSR-hm_0 and MSR-prxy_0 were picked as examples. Fig. 6 are their results in six cases of "RAM Configuration+Scheme", respectively. It clearly shows the contributions of RAM operations ("RAM"), flash operations ("Flash") and LRU overheads ("LRU") to the overall service time with the RAM cache configured to be 32MB and 64MB using APS, Tree-PL and Tree, respectively. JTL differs from APS and Tree in LRU victim selection, so it was excluded. The results in Fig. 6 support our claim that as the capacity of the RAM cache increases, LRU selection overhead will be an issue. We can see for APS and Tree-PL, the overhead of the page-level selection contributes significantly to the worsening of performance as RAM cache scales up. The CG-level LRU selection also suffers from a larger RAM size, but the overhead increases more gradually due to its coarser granularity.

V. CONCLUSION

The RAM cache is an important resource of a NAND flash storage system. Managing it efficiently will yield significant performance improvements. TreeFTL proposed in this paper is capable of utilizing the RAM space jointly for caching information of address translation and buffering data pages. Cached translation pages and data pages are organized in a tree-like structure that can be adapted by TreeFTL for changing workload. To minimize the overhead of cache evictions,

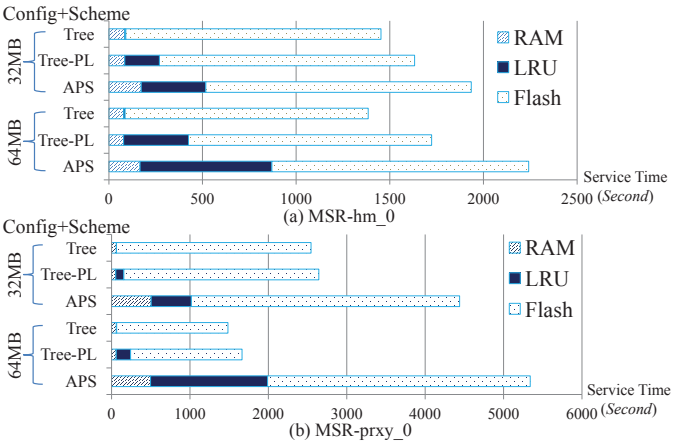


Fig. 6. Effect of Lightweight Victim Selection

TreeFTL uses a lightweight LRU selection algorithm. The victim selection is done at a coarse level, but this trade-off in precision results in the significant reduction in processing time. Experimental results show that TreeFTL can outperform previous schemes on various workloads. As for the future work, we plan to integrate TreeFTL with the garbage collection module of flash management for higher performance.

ACKNOWLEDGEMENT

This paper is supported by the Ministry of Education of Singapore under the grant MOE2010-T2-1-075.

REFERENCES

[1] A. Ban. Flash file system, 1995.
[2] A. Ban. Flash file system optimized for page-mode flash technologies, August 1999.
[3] L.-P. Chang and Y.-C. Su. Plugging versus logging: a new approach to write buffer management for solid-state disks. In *DAC '11*, 2011.
[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*, 2009.
[5] P.-C. Huang, Y.-H. Chang, and T.-W. Kuo. Joint management of RAM and flash memory with access pattern considerations. In *DAC '12*, 2012.
[6] Micron Technology, Inc. NAND flash memory datasheet (MT29F16G08AJADAWP), Feburary 2012.
[7] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.*, 58(6):744–758, June 2009.
[8] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST '08*, 2008.
[9] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. on Consumer Electronics*, 48, 2002.
[10] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao. A block-level flash memory management scheme for reducing write activities in PCM-based embedded systems. In *DATE '12*, 2012.
[11] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4, November 2008.
[12] Storage Performance Council. SPC traces. http://traces.cs.umass.edu/, December 2009.
[13] Z. Qin, Y. Wang, D. Liu, and Z. Shao. Demand-based block-level address mapping in large-scale nand flash storage systems. In *CODES/ISSS '10*, 2010.
[14] Z. Qin, Y. Wang, D. Liu, and Z. Shao. A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems. In *RTAS '11*, 2011.
[15] H. Shim, B.-K. Seo, J.-S. Kim, and S. Maeng. An adaptive partitioning scheme for DRAM-based cache in solid state drives. In *MSST '10*, 2010.
[16] BYU trace distribution center. TPC-C database benchmark traces. http://tds.cs.byu.edu/tds/, 2001.