Achieving Page-Mapping FTL Performance at Block-Mapping FTL Cost by Hiding Address Translation

Yang Hu†, Hong Jiang§, Dan Feng†✉, Lei Tian†§,
Shuping Zhang‡, Jingning Liu†, Wei Tong†, Yi Qin†, Liuzheng Wang†
†*Huazhong University of Science and Technology*
†*Wuhan National Laboratory for Optoelectronics*
§*University of Nebraska-Lincoln*
‡*Institute 706 Second Academy of CASIC*
✉*Corresponding author: dfeng@hust.edu.cn*
jiang@cse.unl.edu, yanghu@foxmail.com, ltian@hust.edu.cn,
{j.n.liu, weitong}@163.com, {qinyi, lewis110456}@smail.hust.edu.cn

Abstract--**Flash Translation Layer (FTL) is one of the most important components of SSD, whose main purpose is to perform logical to physical address translation in a way that is suitable to the unique physical characteristics of the Flash memory technology. The pure page-mapping FTL scheme, arguably the best FTL scheme due to its ability to map any logical page number (LPN) to any physical page number (PPN) to minimize erase operations, cannot be practically deployed since it consumes a prohibitively large RAM (SRAM or DRAM) space to store the page-mapping table for an SSD of moderate to large size. Alternatives to the pure page-mapping FTL, such as block-mapping FTLs, hybrid FTLs (e.g., FAST) and the latest demand-based page-mapping FTLs (e.g., DFTL), require significantly less RAM space but suffer from a few performance issues. Block-mapping FTLs perform poorly with higher erasure counts, particularly under random write workloads. Hybrid FTL schemes incur costly merge operations that hurt performance and increase the erasure counts. Performances of demand-based FTLs heavily depend on workload characteristics such as access locality, read/write ratio and request arrival interval time.**

This paper proposes a new FTL scheme, called HAT, to achieve the performance of a pure page-mapping FTL at the RAM cost of a block-mapping FTL while consuming lower energy, by hiding the address translation (HAT). The basic idea behind our scheme is to create a separate access path to read/write the address mapping information to significantly Hide the Address-Translation latency by incorporating a low energy-consuming solid-state memory device that stores the entire page mapping table. We implement an SSD simulator, SSDsim, to validate our HAT design and evaluate its performance. The extensive trace-driven simulation results show that the performance of HAT is within 0.8% of the pure page-mapping FTL, while consuming about 50% of the energy.

Ⅰ. INTRODUCTION

With the rapid development of the NAND-based Flash memory technology that has enabled their density and endurance to improve significantly while their price to decline dramatically, Flash memory solid-state drives (SSDs) are poised to significantly change the landscape of modern-day storage systems. Compared with hard disk drives (HDDs), SSDs, without mechanical moving parts, have the potential to provide higher read/write performance, lower power consumption, and higher reliability.

However, an inherent physical limitation of NAND-Flash, known as the *write-after-erase* problem [1], renders out-of-place updates for users' write requests. To alleviate this problem, a Flash Translation Layer (FTL) is required to provide a standard block device interface, and perform logical-to-physical address translations to upper layer file systems or databases. FTL also includes wear-leveling and garbage collection modules to reduce erasures. Thus, FTL plays an important role in SSD's performance. The pure page-mapping FTL scheme [2,3], arguably the best FTL scheme due to its ability to map any logical page number (LPN) to any physical page number (PPN) to minimize erase operations, cannot be practically deployed since it consumes a prohibitively large RAM space to store the page-mapping table for an SSD of moderate to large size. As a result, most existing Flash-memory commercial products (e.g., thumb drives, MP3, digital cameras, etc.) employ the block-mapping FTL scheme that use significantly smaller RAM space than page-mapping FTLs (generally by a factor equal to the number of pages in a block) but incur some performance penalty due to direct block mapping. Whereas, most SSD products adopt hybrid FTLs, in which address mapping is at the block level for the most part except for a very

978-1-4244-7153-9/10/$26.00 ©2010 IEEE

small number of blocks, used to log frequently-updated pages and thus called log blocks, which are page-mapped. When log blocks are full, costly block-merge operations that each involves extra read, write and erase operations, will be triggered to reclaim log blocks [4,5,6]. Thus, under write-dominant workloads, log-block-based hybrid FTLs usually perform poorly. This poor performance of block-level and hybrid FTL schemes stems from the fact that the granularity of address mapping in these schemes is at the block level, which is too large for most write requests that are at the page granularity and thus can benefit the most from page-level fully-associative mapping offered by page-mapping FTLs. To address this problem, Aayush Gupta et al. [7] proposed DFTL (Demand-based FTL) that is based on the page-mapping FTL. DFTL uses a limited RAM (e.g., SRAM) space to store a small fraction of the page-mapping information that is the most popular while keeping the rest in Flash memory. DFTL exploits temporal locality of workloads to decrease merge operations and achieve performance improvement over block-mapping and hybrid FTLs. However, we observe from our analysis and experiments that the performance of DFTL is rather sensitively dependent on the workload characteristics including temporal locality, read/write request ratio and request arrival interval time. During the address translation phase, a read request incurs an extra Flash read operation to fetch the required mapping information if the mapping information is not present in RAM, which can happen more frequently if temporal locality of the workload is very low and adversely impact performance. In write-dominant workloads, on the other hand, a large number of extra Flash write operations to update dirty mapping information in Flash memory and the consequent extra erase operations can significantly lower the performance. While these extra operations may be executed in the background in light workloads to lessen their negative performance impact, the problem can become serious in heavy workloads since background operations will inevitably lengthen response time of intervening external I/O requests. The root cause for this is that data and mapping information are both stored on the same Flash chip and share the critical data path, which leads to the collision of reading/writing of the actual user's data and the mapping information.

To address the problem of DFTL discussed above, we propose a new FTL scheme in this paper, called HAT, to *h*ide extra operations incurred in the *a*ddress *t*ranslation phase, thus achieving the performance of a pure page-mapping FTL. HAT creates a separate access path for the mapping information by integrating a low energy-consuming solid-state chip to store the entire mapping information, thus allowing access to the actual data and the address mapping information to be carried out in parallel. Similar to DFTL, popular mapping information is stored on a small-capacity RAM (e.g., SSD's controller cache) to take advantage of workload locality. Further, in contrast to most existing FTLs that are designed for single-chip SSDs, HAT is designed for multi-chip SSDs, with a potential to further exploit parallelism, improve performance and reduce energy consumption.

This paper makes the following main contributions:

- We propose a new FTL, HAT, based on the ideal page-level mapping, to address the performance problem that the state-of-the-art DFTL scheme suffers when workload lacks strong locality. We identify the root cause of the problem to be the shared critical data path in the Flash memory between the user's data and the address mapping data and create a separate path for the latter by incorporating a low energy-consuming solid-state chip to store infrequently-accessed mapping information and a very small RAM space to store frequently-accessed mapping information, thus hiding the address translation latency.
- We implemented a new SSD simulator (SSDsim) that is validated against a real SSD, which, to the best of our knowledge, is the first reported empirical SSD simulator validation study. The empirical measurements on a real SSD development board and our SSDsim indicate a reasonably high accuracy of SSDsim.
- Using six realistic enterprise-scale workloads and a set of synthetic workloads in our extensive trace-driven evaluation on SSDsim, we show that HAT reduces the average response time of DFTL by an amount ranging from 17.9% to 57% under realistic workloads and from 24% to 52% under synthetic workloads. More importantly, HAT achieves a performance that is 0.79% and 0.8% within that of the pure page-mapping FTL under realistic and synthetic workloads respectively. Further, energy evaluation results show that HAT consumes about 50% of the energy of the page-mapping FTL under each realistic workload.

The rest of the paper is organized as follows. Background and motivation for the HAT research is presented in the next section. Section Ⅲ details the design and implementation of HAT, while Section Ⅳ evaluates its performance. The paper is concluded in Section Ⅴ.

Ⅱ. BACKGROUND AND MOTIVATION

In this section, we introduce and review the necessary background information to facilitate and motivate our HAT research.

A. Flash Memory

In general, Flash memory can be classified into two categories: NOR and NAND [8]. NOR-Flash memory supports byte-level random access, and is typically used in read-only applications such as storing firmware codes. NAND-Flash memory, in contrast, has higher density, larger capacity and lower cost than NOR-Flash, but only supports block-level random access. It is thus typically used for more general-purpose applications, similar to those of HDDs, hence our focus for this study.

There are two NAND-Flash technologies, Single-Level Cell (SLC) and Multi-Level Cell (MLC). While the former stores only one bit per cell, the latter stores two or even more bits per cell. Throughout this paper, we will use the term Flash to refer specifically to NAND-Flash memory.

A Flash package consists of one or more dies (chips). All dies share the same 8/16 bit I/O bus but have separate chip enable and control signals. Each die is composed of multiple planes, where each plane contains thousands of blocks and one or two data or cache register(s) used as the I/O buffer. A block typically consists of 64 or 128 pages, where a page is further divided into multiple 512B sub-pages. Each sub-page has a 16B spare space used to store a variety of information including: error correction code (ECC), logical page number and sub-page state. Figure 1 shows an example of a page structure.

Main field				Spare field			
1st	2nd	3rd	4th	1st	2nd	3rd	4th
512B	512B	512B	512B	16B	16B	16B	16B

Figure 1: **An example of the page structure**

There are three basic operations in Flash: read, write (program) and erase. A read operation fetches data from a target page. A write operation writes data to a target page. An erase operation resets all bits of a target block to "1". A write operation can only change the value of each target bit from "1" to "0". Once a page is written, it must be erased before the next write operation can be performed on the same page, which is the well-known *write-after-erase* problem. In addition, there are two advanced operations in Flash: *copyback* and *interleave*. The former enables a page to be copied within a plane without utilizing the I/O bus of Flash, while the latter provides I/O parallelism among dies and planes.

One major challenge facing Flash is the limited erase cycles per block before it is worn out. After wearing out, a block can no longer store any data. A typical MLC Flash has an erase-cycle limit of about 10K, while a typical SLC Flash has an erase-cycle limit of about 100K with a 1bit/512byte ECC [9].

To hide the above Flash complexities, a Flash Translation Layer (FTL) is required.

B. *Flash Translation Layer*

FTL is a software middle layer between the standard storage interface protocols such as SCSI and ATA/IDE and Flash packages of SSD. FTL emulates a hard disk and exposes an array of logical sectors to the upper-level file or database systems. Its major functions includes: (i) logical-to-physical address translation, (ii) Garbage Collection (GC), (iii) Wear-Leveling (WL). Unlike HDD, writes to the same logical page in Flash are actually directed to different physical pages due to the write-after-erase nature of Flash. Thus, a mapping scheme must be used to map a logical unit address to a physical unit address. The granularity of this mapping can be a Flash page or a Flash block. GC is needed to deal with the aftermath of the write-after-erase problem. To avoid the costly erase operation on each write request, the new data is first written to a clean physical page, while the previously occupied physical page is invalidated by simply updating its state information. In the course of time, a large number of invalidated pages will be accumulated. If the number of invalid page is above a predefined threshold, a GC process is initiated to reclaim all the invalid pages by erasing them. On the other hand, since each block has roughly the same erase-cycle limit, WL is used to evenly distribute erase operations to all blocks to increase the overall life-cycle of SSD. Note that the design choices of specific GC and WL mechanisms generally depend on the schemes of logical address mapping.

When an upper-level component (e.g., file system) issues a read or write request with a Logical Sector Number (LSN) to read from or write to a specific address of Flash, an address translation is triggered. The size of a logical sector is 512B, the typical size of a physical sector in HDD. Assuming that there are *n* sub-pages in a Flash page, the address mapping between LSN and Logical Page Number (LPN) is:

$$LPN=LSN/n$$

Assuming further that there are *m* pages in a Flash block, the address mapping between LPN and Logical Block Number (LBN) is:

$$LBN=LPN/m$$

1) *Page-mapping FTL*

The pure page-mapping FTL is a classic FTL scheme [2, 3]. In this FTL, each logical page can be mapped to any physical page in SSD (i.e., fully associative mapping). A page mapping table is used to store and manage the mapping information between LPN and Physical Page Number (PPN). For performance purposes, the mapping table is usually stored in DRAM. Assuming a 256GB SSD with a page size of 2KB, there will be 2^{27} entries in its mapping table. If each entry consumes 4B space, the mapping table will occupy 512MB DRAM space, which can be too expensive and energy-demanding for a typical SSD product to be cost-effective and energy-efficient.

2) *Block-mapping FTL*

The pure block-mapping FTL is another classic FTL scheme [2, 3]. A block-mapping table is used to store and manage the mapping information between LBN and Physical Block Number (PBN). If there are *m* pages in a block, the size of the block-mapping table is *m* times smaller than its page-mapping counterpart. In a block-mapping FTL, one LPN must be mapped to a fixed page offset in any physical block (i.e., direct mapping). If this page offset has been written before, the LPN can not be written to any other page in this block even if there are free pages in the same physical block. In this case, all existing valid data in the block as well as the data to be written must be copied to a new clean block, and the old block is marked for erase, incurring one erase and a number of read/write operations. Compared with the page-mapping FTL, the block-mapping FTL requires extra operations to serve a request, adversely affecting the performance.

Since both the block-mapping and page-mapping FTLs have their aforementioned disadvantages, they are rarely used in SSD commercial products in their pure forms.

3) Hybrid FTL

A family of the hybrid mapping schemes is introduced to address the aforementioned shortcomings of the page-mapping and block-mapping FTLs. In a typical hybrid FTL, physical blocks are logically partitioned into two groups: data blocks and log blocks. Data blocks account for the vast majority of all physical blocks and are organized and managed by the block-mapping scheme. There are a very small number of log blocks that are page-mapped and invisible to upper-level components (or the users). When a write request arrives, the hybrid FTL first writes the new data in a log block and invalidates the data in the corresponding target data block. Block-mapping information for data blocks and page-mapping information for log blocks are kept in a small RAM for performance purposes. When all the log blocks are full, their data are flushed into the data blocks immediately and they are then erased to generate new free log blocks. More specifically, the valid data in data blocks and the valid data in the corresponding log block must be merged and written to a new clean data block. This process is called a *merge* operation. Further, merge operations can be classified into three types depending on their overhead. *Full merge* occurs, when the log block is selected as a victim block and not written sequentially from the first page to the last page, and all the valid data in it and in its corresponding data block are copied to a new clean block. This process requires *m* read operations, *m* write operations and two erase operations, where *m* is the number of pages in a block. When the log block is written sequentially from the first page to the last page of a logical block, this log block can replace the corresponding data block, a merge operation called *switch merge*. This type of merge requires only one erase operation. *Partial merge* takes place when the log block is written sequentially from the first page to a middle page in a block, and the last part of data will be copied from the corresponding data block. Partial merge requires several read and write operations and one erase operation.

A number of variations of the hybrid FTL schemes have been proposed recently, including BAST [10], FAST [4], LAST [5], Superblock [6], Reconfigurable FTL [23]. They attempt to address the problems arising from the costly full merge operation in different ways. However, since the merge operations can only be reduced to a certain degree, none of these hybrid FTLs is able to achieve the desired performance of a pure page-mapping FTL.

More recently, Demand-based FTL (DFTL) [7] was proposed to address the RAM-capacity problem of the page-mapping FTL by storing only the "hot" mapping information in RAM based on temporal locality of workloads. DFTL is shown to significantly outperform hybrid FTLs.

C. *Research Motivation for HAT*

There are several factors affecting the performance and reliability of SSD, such as the capacity of data buffer, buffer management algorithms, request scheduling algorithms, I/O parallelism exploitation in SSD, FTL schemes and so on. In this paper we focus on what we believe to be the most important one: FTL schemes.

Although the page-mapping FTL can provide arguably the best performance and reliability, the huge demand on the DRAM (as SRAM density does not meet the requirement) space to store the mapping table leads to a significant increase in cost and energy consumption. Therefore, as far as we know, most, if not all, current SSD commercial products employ the hybrid FTL schemes. While DFTL is a promising alternative to the hybrid schemes with a comparable performance to the page-mapping FTL scheme, it has a few limitations as elaborated below.

The essence of DFTL is its effective exploitation of the temporal locality of workloads. In the ideal case scenario, since each piece of the required mapping information belongs to a "hot" request and thus exists in RAM, DFTL performs exactly the same as the pure page-mapping FTL scheme since no extra Flash read/write operation is required. In a real environment where a request misses the "hot" portion of the mapping table residing in RAM, however, DFTL's performance is significantly affected by the following operations: reading/writing the mapping information from/to Flash during the address-translation phase and extra erase operations resulting from updating the mapping information in Flash (for a write request). Different characteristics of workloads, such as temporal locality, request arrival interval time (i.e., arrival rate or I/O intensity) and read/write ratio, can either exacerbate or mitigate the negative impacts of these extra operations on DFTL performance. This is clearly evidenced by Table 1 in which we list the amounts of performance deviation of DFTL from the ideal case scenario (i.e., the pure page-mapping FTL) under four workloads based our trace-driven simulations on SSDsim.

Table 1: Performance deviation of DFTL from page-mapping FTL

	Web1	Fin2	Fin1	Openmail
Temporal locality	2.5%	76.7%	65.9%	47.1%
Read ratio	99%	82.3%	23.2%	45.4%
Int. arrv. time(ms)	2.99	11.08	8.19	<1 (96%)
Perf. deviation	18.8%	8.3%	21.8%	57.08%

Under the read-dominant workload, the Web1 trace [11] with a read request ratio of 99%, read requests do not update the mapping information, thus avoiding much of the extra Flash write operations (to write the mapping information back to Flash) and extra erase operations. On the other hand, its temporal locality is only 2.5%, which incurs a large number of extra Flash read operations. Each extra Flash read operation directly lengthens request response time, as show in Figure 2. It results in an 18.8% performance deviation of DFTL from the page-mapping FTL.

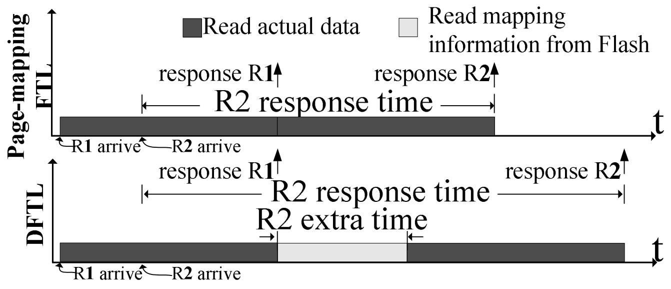

Figure 2: Extra Flash read operation increases request response time.

For the write-dominant workload, the Fin1 trace [11] with a write ratio of about 73%, the negative impact of extra Flash read operations on performance is relatively low since a write request needs not read the mapping information before writing actual data to Flash. Write requests lead to a large number of mapping information update operations, resulting in a large number of extra Flash write operations and many extra erase operations. Consequently, subsequent requests will be delayed, as illustrated in Figure 3, causing a 21.8% performance deviation of DFTL from the page-mapping FTL for the Fin1 trace.

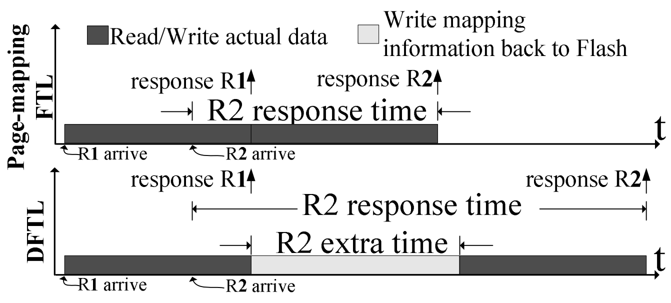

Figure 3: Extra Flash write operation increases request response time.

Clearly, the higher the I/O request intensity (or request rate), represented by the average arrival interval time, the higher the impact the extra Flash operations will have on the performance deviation of DFTL. In the Openmail [12] workload that has the highest request intensity, the extra operations lead to a 57.08% performance deviation of DFTL from the page-mapping FTL.

In DFTL, both the actual user's data and mapping information are stored on the same Flash chip, sharing the same I/O path to/from Flash. The sharing of the Flash I/O path in fact is the root cause for DFTL's performance deviation from page-mapping FTL, as explained next. If the required mapping information for a read request is not present in RAM, for example, DFTL can not read the actual user's data until the mapping information is fetched from the page-mapping table in Flash. On the other hand, if the miss on the page-mapping table in RAM results in a replacement of a dirty entry in the RAM-resided table, the dirty entry must be written back to Flash to update the Flash page-mapping table. This Flash write operation and the potential erase operation will delay a subsequent request.

Thus, an obvious solution to the above problem is to store the Flash page-mapping table away from the Flash, in a different and independent chip. This solution, shown in Figure 4, can potentially allow the actual user's data and the mapping information to be read/written in parallel to hide the impact of the extra operations during the address translation phase, thus reducing the request response time.

These important observations and analysis motivate us to propose the HAT scheme, detailed in the next section.

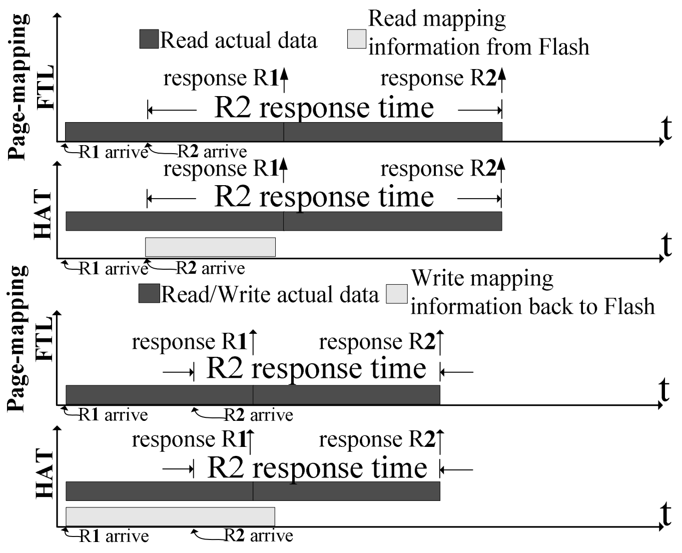

Figure 4: Hiding the cost of address translations in HAT.

III. DESIGN AND IMPLEMENTATION OF HAT

A. *The HAT Architecture*

Similar to DFTL, HAT is based on the page-mapping FTL, but it differs from DFTL in that it stores the user's data and page-mapping table separately to create independent access paths to these two types of data, thus significantly hiding the address-translation latency. A low energy-consuming solid-state chip is used to store the entire page-mapping table while a very small fraction of the mapping information is stored in DRAM to exploit locality, as shown in Figure 5. In our current design, we choose Phase-Change Memory (PCM) [13,14] as this low energy-consuming solid-state chip. PCM is emerging as a promising high-performance nonvolatile memory technology that supports byte-access and in-place update and is very suitable for mapping-information updating. Detailed descriptions and discussions of PCM can be found in [13,14,15,16]. Recent studies [17,18,19,20] have proposed some hybrid SSD architectures that use PCM as a write buffer. The basic assumption made by these studies is that the latency of the PCM write operation is equal to the theoretical latency, while in fact the write latency in practice is longer than the theoretical PCM latency or even longer than the Flash write latency [13]. Our design of HAT is based on the measured real latency of the PCM write operation.

Frequently-accessed mapping information is stored in a small-capacity RAM (e.g., controller cache). Because the access speed of RAM is higher than PCM, the smaller page-mapping table stored in RAM is called Quick Mapping Table (QMT), whereas the entire page-mapping table, stored in PCM, is called Slow mapping Table (SMT). QMT is a small subset of SMT.

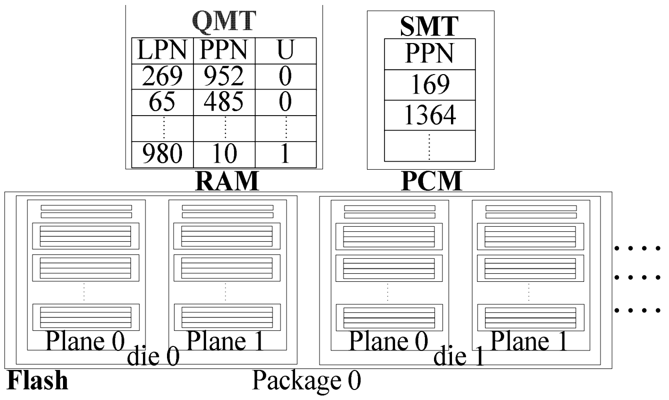

Figure 5: The architecture of HAT

QMT: Quick Mapping Table

Each entry of QMT consists of three parts, LPN (Logical Page Number) used as the search index for QMT, PPN (Physical Page Number) used to index the actual data in Flash, and the update bit (U). The update bit is used to indicate whether the mapping information stored in this entry has been updated (“1”) or not (“0”).

If the required mapping information exists in QMT, a QMT hit happens; otherwise a QMT miss occurs. We use the LRU algorithm for the replacement of entries in QMT.

SMT: Slow Mapping Table

All of the logically consecutive mapping information is sequentially stored on PCM. HAT uses LPN as the offset address to find the entry belonging to LPN in SMT on a QMT miss.

Each entry in SMT contains a single field: PPN, which is used to index the actual data in Flash.

B. Read/Write operations in HAT

The procedure by which HAT services a request is described in Algorithm1.

Algorithm 1 HAT Algorithm

Input: Request R with logical page number R_{LPN}, request type R_{type} and request size R_{size}

Output: NULL

```
1.  size ← R_size
2.  LPN ← R_LPN
3.  while size > 0 do
4.      entry(lpn).lpn ← LPN
5.      entry(lpn).ppn ← NULL
6.      entry(lpn).update ← 0
7.      if R_type ≡ read then
8.          if QMT hit then
9.              entry(lpn).ppn ← search_QMT(entry(lpn).lpn)
10.         else {/* QMT miss */}
11.             if QMT is full then
12.                 detele_QMT_Tail()
13.             end if
14.             entry(lpn).ppn ← Search_SMT(entry(lpn).lpn)
15.         end if
16.     else {/* R_type ≡ write */}
17.         entry(lpn).ppn ← get_ppn()
18.         entry(lpn).update ← 1
19.         if QMT is full then
20.             detele_QMT_Tail()
21.         end if
22.     end if
23.     move_QMT_head(entry(lpn))
24.     size ← size − 1
25.     LPN ← LPN + 1
26. end while
27. return
```

When a write request arrives, HAT calls the *Get_PPN* function to get a clean physical page for serving the request. On a QMT hit, the PPN field of the entry is updated to the new value, and the update bit of the entry is changed to “1”. Finally, the entry is moved to the head of QMT; On a QMT miss, on the other hand, HAT invokes the *Detele_QMT_Tail* function if QMT is full (as shown in Figure 7). Finally, the new entry is added to the head of QMT.

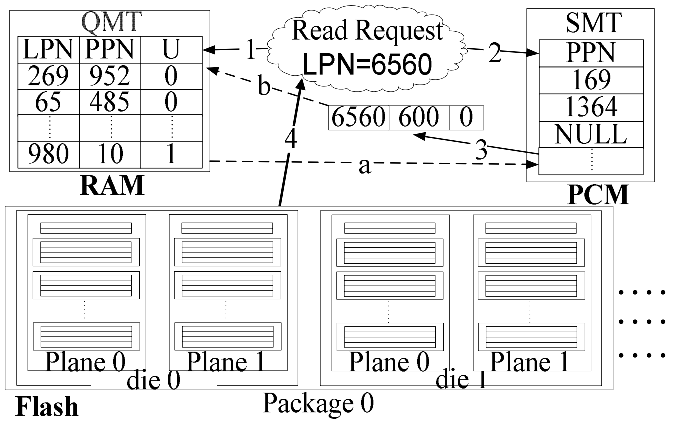

Figure 6: An example of HAT serving a read request

on a QMT miss (1) searches QMT with LPN=6560 as the index value, and misses; (2) searches SMT with LPN=6560 as the offset address; (3) fetches PPN=600; (4) reads the target page in Flash and transfers the data to the controller; (a) writes the last entry of QMT back to SMT; (b) adds the new mapping information to the head of QMT. Note that step_(a) is executed simultaneously with step_(4), thus hiding the latency.

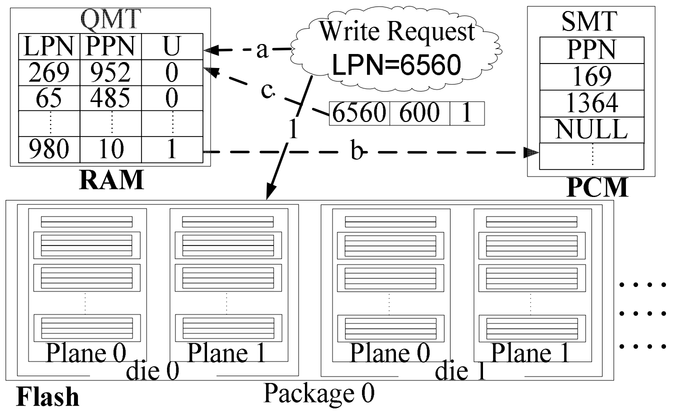

Figure 7: An example of HAT serving a write request on a QMT miss. (1) writes the valid data to the target page (PPN=600); (a): searches QMT with LPN=6560 as the index value, and misses QMT that is also full; (b) writes the last entry of QMT back to SMT; (c) adds the new mapping information to the head of QMT. Note that step_(a) and step_(b) are executed simultaneously with step_(1).

A read request is directly serviced through the Flash page read operation on a QMT hit and the corresponding mapping information is then moved to the head of QMT. On a QMT miss, HAT calls the *Search_SMT* function to retrieve the mapping information from SMT using LPN as the offset address. At the same time, HAT invokes the *Detele_QMT_Tail* function to reclaim an entry space for the new mapping information when QMT is full (as shown in Figure 6). The function of *Detele_QMT_Tail* determines if the last entry of QMT will be deleted directly or written back to SMT by checking the update bit of the last entry. If the update bit is "1", HAT writes the last entry back to SMT, or deletes it directly otherwise. Finally, HAT adds the new mapping information to the head of QMT.

C. The Advantages and Limitations of HAT

We discuss the advantages and limitations of HAT from the following three aspects.

1) Performance: In the hybrid mapping FTLs, random writes lead to a large amount of merge operations that incur significant performance penalties. Since HAT is based on the page-mapping FTL, which provides fine-granularity fully associative page-mapping to effectively handle random writes, it minimizes merge operations and thus improves random write performance. Since HAT separates the IOs of actual user's data and the mapping information and executes them in parallel, it is able to provide improved performance over DFTL in which these types of data share the same critical data path.

2) Energy Consumption: HAT uses low energy-consuming nonvolatile memory (e.g., PCM) to replace DRAM to store the page-mapping table. Compared with non-volatile memory, DRAM consumes much more energy due to its *refresh* operations. Therefore HAT can provide better energy-efficiency than the page-mapping FTL that uses DRAM to store the entire mapping table. Other existing FTLs, due to the lack of a dedicated chip and separate data path, incur many extra operations, such as the erase operations in hybrid FTLs and the extra Flash read/write operations to read/write mapping information and erase operations in DFTL, thus consuming much more energy than HAT.

3) Cost: We list the price/capacity ratios of three types memory chips in 2009 in Table 2. We can see that the ratio for PCM is the highest at the present. Low volume production and shipment of PCM lead to its high average cost. In the near future, with the increase in production volume and shipment of PCM, its price will likely decline steadily, and the price/capacity ratio is expected to decrease for PCM.

Table 2: Unit price of Flash, PCM and DRAM (Since prices fluctuation according to density, quantity and other factors, we list the range of prices)

Unit: $	**Flash**	**PCM**	**DRAM**
Price/Mb	10^{-4}	10^{-2}~10^{-3}	10^{-3}

Ⅳ. PERFORMANCE EVALUATIONS

A. Experimental Setup

1) SSDsim Simulator

We design and implement an SSD simulator, called SSDsim, which is event-driven and modularly designed. SSDsim is a single-threaded program written in C, which has about 10 thousands lines of C code. SSDsim is capable of simulating most SSD hardware platforms, mainstream FTL schemes, buffer management algorithms and request scheduling algorithms. The tri-tiered SSDsim structure consists of the buffer and request-scheduling module at the top, the FTL module in the middle, and the hardware module at the bottom. The buffer and request-scheduling module simulates the write-buffer management and request scheduling. The FTL module simulates many state-of-the-art FTL schemes. According to the request type and request size, the FTL module chooses suitable standard Flash commands and sends them to the hardware module in the third level. We have implemented several typical FTL schemes, such as block-mapping FTL [2,3], FAST [4], DFTL [7], page-mapping FTL [2,3] and our HAT scheme. The hardware module simulating the behaviors of all of Flash operations is based on Open NAND Flash Interface Specification (ONFI) 2.2 [21]. In the hardware module, there are two sub-modules, the time-consumption sub-module that tracks the amount of time taken by different operations and the energy-consumption sub-module that tracks the amount of energy the hardware consumes. By feeding the block-level traces files and configuring with the parameter files to SSDsim, we can obtain the processing time and energy consumption of each request, along with other results.

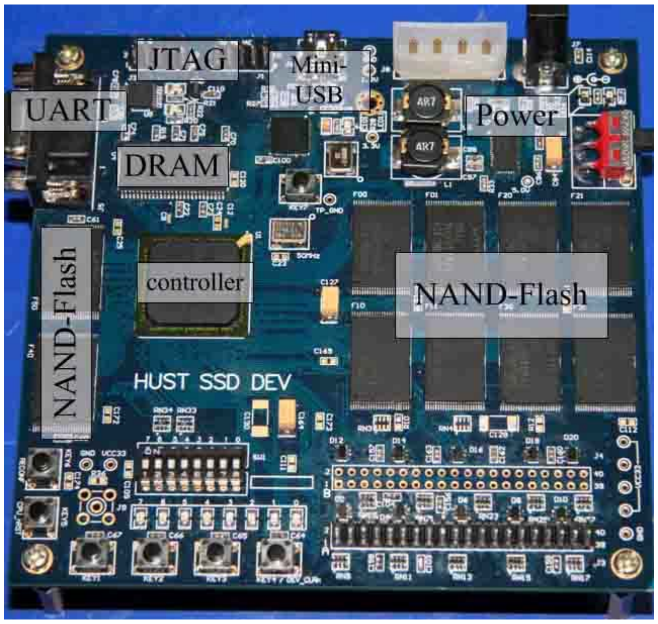

Figure 8: Prototyped SSD board. We implemented this prototype with 10×2GB SLC NAND-Flash and 16MB DRAM.

At the same time, we have also implemented an SSD hardware prototype, as show in Figure 8. We employ the page-mapping FTL in the prototype. Using this prototype, we validate our SSDsim preliminarily. In sequential read-dominant workloads, the measurements obtained from SSDsim are within 0.3% of those from the real SSD prototype. In random write-dominant workloads, the measurements obtained from SSDsim are within 11.5% of those from the real SSD prototype. In other mixed workloads, the measurements from SSDsim are within 0.3%~11.5% of those from the real SSD prototype. These results indicate a reasonably high accuracy of SSDsim. After completing a comprehensive validation and robustness test of SSDsim in the near future, we plan to release the SSDsim source code to the public domain.

2) Workloads

We use a set of realistic and synthetic traces to study the impact of different FTLs on performance and energy consumption. Fin1 and Fin2, and Web1, Web2, and Web3 were collected at a large financial institution and at a popular Internet web search machine respectively [11]. The Openmail trace [12] was obtained from a centralized email server used by a large number of users.

Table 3: Workload characteristics of the traces.

Workloads	**Avg. req. size**(512B)	**Read**	**Int. arrv. time**(ms)
Fin1	6	23.2%	8.19
Fin2	4	82.3%	11.08
Openmail	11	45.4%	<1 (96%)
Web1, 2, 3	30	99.9%	2.99/3.36/4.57(97%)
Synthetic 1	4/8/16/32/64/128	100%	0.05/0.1/0.3
Synthetic 2	4/8/16/32/64/128	0%	0.1/0.2

3) Simulator parameters

The SSDsim in our experiment is configured with its key parameters listed in Table 4.

We evaluate the advantages and limitations of each kind of FTLs. In particular, we offer a DRAM chip for the page-mapping FTL to store the page-mapping table. FAST and block-mapping FTL store the entire block-mapping table in the controller cache. DFTL and HAT store the popular part of the page-mapping table in the controller cache.

4) Performance and energy consumption metrics

The *average response time* measure is a good metric for estimating FTL performance since it captures all the overheads associated with servicing a request, including I/O driver queue time, device service time, read/write mapping-information time and garbage collection time. The *Input/Output Per Second (IOPS)* measure is another good metric for evaluating the maximum read/write performance of each FTL scheme.

The *total energy consumption* measure is a good metric for evaluating the impact of FTLs on energy consumption. Since the choice of FTLs does not affect the controller energy consumption while the energy consumption of other components is small, we only consider the energy consumption of Flash, PCM and DRAM in our estimates of total energy consumption.

Table 4: Configuration parameters of SSDsim (The specification parameters are obtained from [9,13,22])

Organization		**Timing characteristics** (us)		**Electrical characteristics** (mA, V)	
Channel	4	t_{BERS}	1500	DRAM r/w	125
Package	1/channel	t_{PROG}	200	refresh	3
Die	4/package	t_R	20	Flash r/w/e	25
Plane	4/die	t_{WC}	0.025	PCM read	8
Block	2048/plane	t_{RC}	0.025	PCM write	35
Page	64/block	PCM r	0.115	Supply Voltage (V)	3.3
Sub-page	4/page	PCM w	90		
Over-provisioned (Redundant space)					0.1
Controller cache (KB)					128
GC threshold					0.2

B. Analysis of the Evaluation Results

1) Macro-benchmark Performance Analysis

We evaluate five FTL schemes: page-mapping FTL, block-mapping FTL, FAST, DFTL, and HAT. Figure 9 and Figure 10 illustrate the average response times, erasure counts and performance deviation from the the page-mapping FTL under Fin1, Fin2, Openmail and Web1, Web 2, Web 3.

Due to the large numbers of write requests in Fin1, Fin2 and Openmail, FAST and block-mapping FTL induce the numbers of erase operations that are one order of magnitude larger than those in page-mapping based FTLs. Compared with the page-mapping FTLs, the performance degradation of FAST is up to a factor of 64, and the performance degradation of block-mapping FTL reaches a factor of 1519.

The Fin2 trace has an 82.3% read radio and its arrival interval time is about 11 milliseconds, suggesting a light load. Thus, the extra Flash read operations triggered to fetch mapping information are the most important factor responsible for the increased average response time. For a page read request, the time to read the actual data is about 72.8us (t_R +2112× t_{RC}). If the required mapping information is not present in RAM, DFTL spends more than 20us to read mapping information from Flash. In the Fin2 trace, the average request size is 4 sectors, which is equal to the page size in our experiment. Thus, if the required mapping information of a read request is not present in RAM, request response time will be increased by about 27% (i.e., 20/72.8). Because about one quarter of the requests incur extra Flash read operations, along with some extra Flash write operations due to some of the write requests, the average response time in DFTL is increased by 8.3%. Whereas, the discrepancy in average response time between HAT and the pure page-mapping FTL is only 0.79%. The main reason behind HAT's superior performance are twofold. First, when a read request waits in the I/O driver queue and its required mapping information is not present in QMT, HAT can fetch the mapping information from PCM while Flash is serving other requests (recall Figure 4). Second, when the evicted mapping information is written back to PCM, Flash can serve other subsequent requests (recall Figures 4, 6, and 7). In both cases, the address translation latency is completely hidden.

Fin1 is write-dominant and its mapping information hit ratio is 65.9% (see Table 1). In DFTL, a large number of write requests produce many mapping information updates, which in turn lead to many extra Flash write and erase operations. Compared with the pure page-mapping FTL, DFTL incurs 21.8% extra response time cost. HAT has the same erase number as the page-mapping FTL since it is based on the page-mapping scheme. Although HAT experiences the same number of misses on the RAM page-mapping table as DFTL, its average response time deviation from pure page-mapping FTL is only 0.38% because of its ability to completely hide the latency in accessing the PCM page-mapping table.

The Openmail trace is the heaviest workload among all the traces considered in this study, as more than 96% requests have arrival interval times below 1 millisecond. In DFTL, a large number of background operations, including extra Flash write and erase operations, significantly increase the waiting time in the I/O driver queue of subsequent requests. It results in an average response time deviation of up to 57% from the page-mapping FTL. In contrast, the performance deviation of HAT is only 0.5%, again due to its ability to completely hide address translation latency incurred in accessing the PCM page-mapping table.

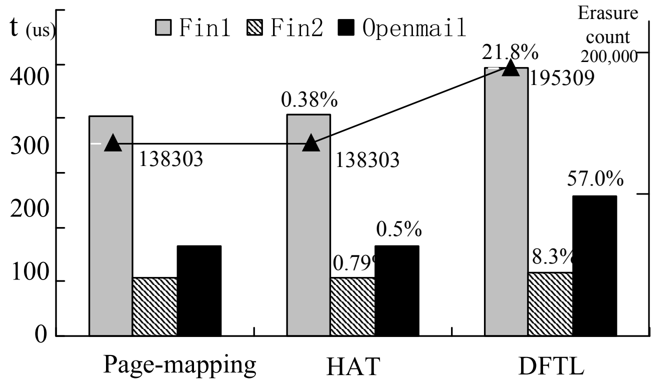

Figure 9: The average response time of Fin1, Fin2 and Openmail (In this figure, the average response time of FAST and block-mapping FTL are significantly larger than others and thus not displayed here. The percentages indicate average response time deviations from the page-mapping FTL. The integers denote the erase operation counts of the FTLs under Fin1)

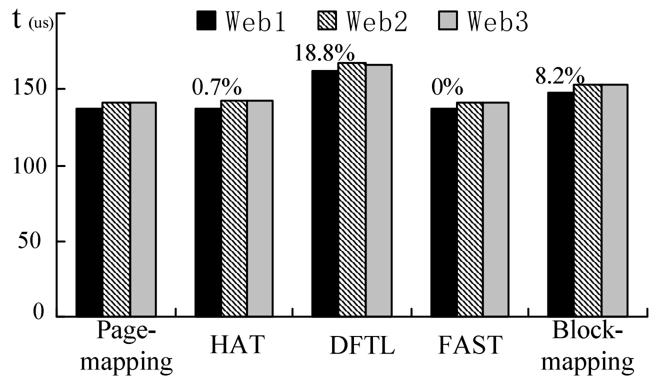

Figure 10: The average response time of Web1, Web2, Web3

The Web1, Web2, and Web3 traces are read-dominant workloads. The page-mapping FTL and FAST have the same average response times, because the small-capacity log blocks in FAST can serve all the write requests and hence there is no merge operation. Block-mapping FTL has an average response time deviation of 8.2% from the page-mapping FTL, since write requests lead to a significant number of erase operations. In DFTL, the mapping information hit ratio is 2.5% (see Table 1), which leads to a large number of extra Flash read operations. Therefore, the average response time of DFTL deviated from the page-mapping FTL by about 18.8%. HAT only has a 0.7%

performance loss, since it can read the mapping information from PCM and the actual data from Flash in parallel.

Table 5: The range of average response time deviation from page-mapping FTL

FTLs	Min deviation	Max deviation
HAT	0.38%	0.79%
DFTL	8.3%	57.0%
FAST	0	64 times
Block-level	8.23%	1519 times

From the above analysis of performance results under the six workloads, it is clear that HAT outperforms all other FTL schemes except for the pure page-mapping FTL scheme that stores the entire mapping table in DRAM. More importantly, HAT's performance comes very close to that of the page-mapping FTL scheme, by a maximum margin of only 0.79%. Table 5 lists the maximum and minimum of average response time deviation of each FTL scheme from the page-mapping FTL scheme.

2) Micro-benchmark Performance Analysis

We use a set of micro-benchmark workloads to evaluate random IOPS measure of the page-mapping FTL, DFTL and HAT schemes. In these workloads, the mapping information hit ratios are all within 6%. Figure 11 shows the IOPS measurements of HAT and DFTL as a function of the request size, normalized to the IOPS of the page-mapping FTL scheme. Clearly, HAT consistently outperforms DFTL and maintains a performance that is almost equal to that of the page-mapping FTL.

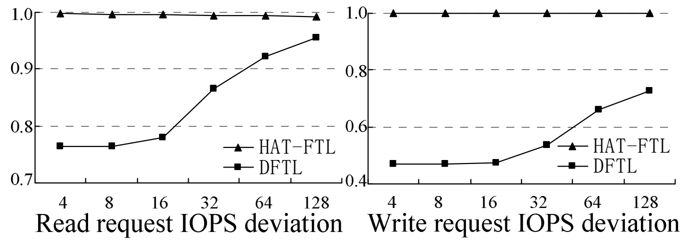

Figure 11: IOPS as a function of the request size, normalized to the IOPS of page-mapping FTL (the unit of the request size is sector (512 byte))

In random read workloads, we can see that IOPS of HAT is very stable, with a deviation from page-mapping FTL of no more than 0.8%. The DFTL performance is rather unstable, resulting in an IOPS deviation of between 23.3% and 4.5%. In DFTL, one extra read operation can fetch 512 mapping entries (i.e., a Flash page is 2KB and each mapping entry consumes 4 Byte space.). As the request size increases, the fraction of the request response time consumed by fetching the mapping information in DFTL slowly diminishes, which explains the climbing trend of the DFTL curve in the Figure 11.

In random write workloads, HAT shows the same write performance as the page-mapping FTL, while the DFTL write performance deviates from the page-mapping FTL scheme by amount between 47.4% and 72.6%. For each write request, HAT writes the actual user's data to Flash and at the same time writes the evicted mapping information back to PCM in parallel. When the actual data write operation completes, mapping information update operation is also finished. While in DFTL, after writing the actual data to Flash, the next write request can not be serviced immediately until the extra Flash write operation for the evicted mapping information is finished.

3) Microscopic Analysis

In what follows, we conduct a microscopic analysis on the impact of Flash reading/writing mapping information on instantaneous response time. Figure 12 depicts the same set of 50 consecutive requests being served by HAT and DFTL under the Fin1 trace. It illustrates how the Flash write operation for mapping information impacts the response time of a subsequent request.

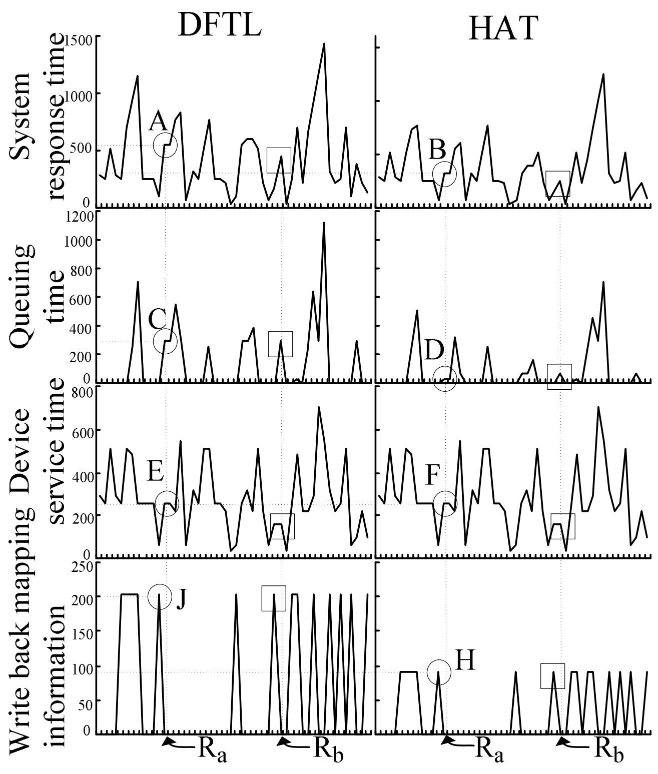

Figure 12: Microscopic analysis of the HAT and DFTL schemes under the Fin1 trace.

Since HAT and DFTL are both based on the page-mapping FTL scheme, their service times for the same request (R_a) are the same (E and F, as shown in circles). However, the request has different response times (A and B).

In DFTL, there is an extra Flash write operation to write mapping information back to Flash triggered by an earlier request (J), which incurs a high latency in I/O driver queue of R_a (C). The write-back operation for the mapping information of the earlier request increases the request response time in DFTL (A). In contrast, while the earlier request also leads to mapping information write-back in HAT (H), it does not incur I/O driver queuing time of R_a (D) in HAT. The mapping information write-back operation for the earlier request and the read/write operations on the actual data for request R_a are executed in parallel in HAT. R_b has a similar experience, as shown in square points in Figure 12.

4) *Energy Consumption Analysis*

During the runtime, DRAM refreshes each cell continuously to maintain its data. Refresh operations consume significant amount of energy, which fortunately are not required in Flash and PCM. In Flash, read, write and erase operations consume energy. For example, given that the time consumption of a Flash erase operation is 1.5 milliseconds, the supplying voltage is 3.3 volt and operating current in an erase operation is 25 milliampere (see Table 4), each erase operation in Flash consumes 123.75 micro-joules.

Figures 13 and 14 illustrate the energy consumption of each FTL under the Fin1 and Web1 traces.

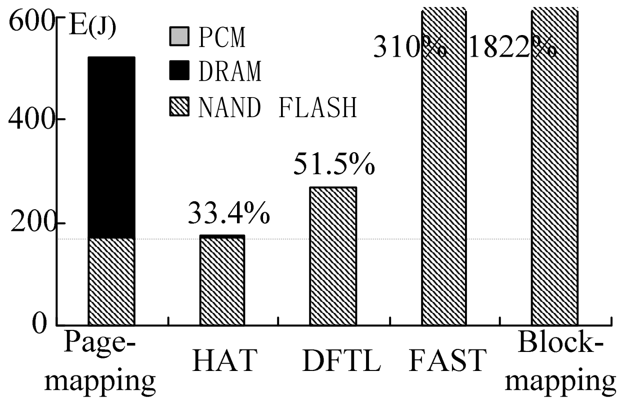

Figure 13: Energy consumption under Fin1

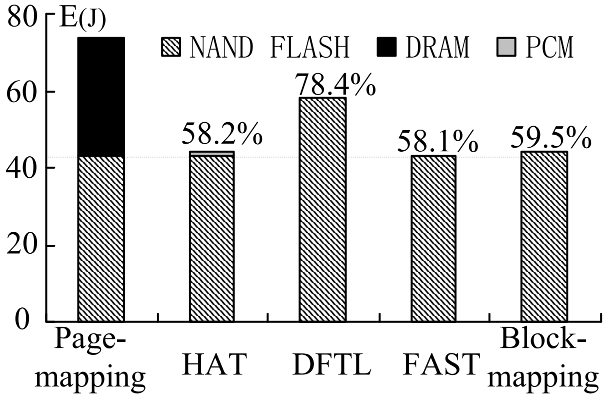

Figure 14: Energy consumption under Web1

Under Fin1, a large number of erase operations in the FAST and block-mapping FTL scheme lead to a large amount of energy consumption. DFTL consumes 51.5% of the energy consumed by the page-mapping FTL scheme, which is 54% more than what HAT consumes. This is because DFTL incurs many extra Flash read/write and erase operations. The fact that PCM supports byte-access and in-place write, which make PCM far more energy efficient than SSD since no extra read/write and erase operations are needed, enabling HAT to be more energy efficient than DFTL and consuming 33.4% of the energy consumed by the page-mapping FTL scheme.

Under the Web1 trace, FAST consumes the least amount of energy, since there is no any extra operation and component. The block-mapping FTL scheme involves some erase operations, which makes it less energy efficient than FAST. DFTL consumes 78.4% of the energy consumed by the page-mapping FTL scheme, since 98.5% the requests each incurs one extra Flash read operation in DFTL. While the same 98.5% of the requests each incurs one extra PCM read operation, HAT only consumes 58.2% of the energy due to the aforementioned salient feature of PCM.

From above observations and analysis, we can conclude that HAT is very energy-efficient for two reasons. First, HAT does not induce any extra operation in Flash itself. Second, all the extra operations for address translation are served by PCM that is very energy efficient and supports byte-access and in-place write.

V. CONCLUTION

We argued that existing state-of-the-art FTL schemes exhibit poor performance and energy efficiency in realistic and synthetic workloads. We propose a new FTL scheme called HAT based on the page-mapping FTL, to separate mapping information and data I/O path to hide the latency due to read/write operations on the mapping information during the address translation phase. Our experimental evaluation using SSDsim with realistic and synthetic workloads demonstrates that HAT offers high performance and energy efficiency. In essence, HAT achieves the performance of the pure (ideal) page-mapping FTL scheme at a cost of the block-mapping FTL scheme.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and insights. This work was supported in part by Microsoft Research Asia Project under No. FY09-RES-OPP-011, the National High Technology Research and Development Program (863 Program) of China under Grant No. 2009AA01A401, 2009AA01A402, Changjiang Innovative Group of Education of China No. IRT0725, NSFC No.60933002, US NSF under Grants NSF-IIS-0916859, NSF-CCF-0937993 and NSF-CCF-0621526, and Wuhan National Laboratory for Optoelectronics Innovation Foundation under No. Z080006.

REFERENCES

[1] An Introduction to NAND Flash and How to Design It In to Your Next Product. http://download.micron.com/pdf/technotes/nand/tn2919.pdf

[2] Eran Fal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, Volume 37, June 2005.

[3] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee and Ha-Joo Song. System Software for Flash Memory: A Survey. In *Proc. of EUC'06*, Augeust 2006.

[4] Sang-Won LEE, Dong-Joo Park, Tae-Sun Chung, Dong-Ho LEE, Sang-Won Park and Ha-Joo Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. In *ACM Transactions on Embedded Computing Systems*, Volume 6, July 2007.

[5] Sungjin Lee, Dongkun Shin, Young-Jin Kim and Jihong Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. In *Proc. of SPEED'08*, February 2008.

[6] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim and Joonwon Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proc. of EMSOFT'06*, October 2006.

[7] Aayush Gupta, Youngjae Kim Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-Level Address Mapping. In *Proc. of ASPLOS'09*, March 2009.

[8] M-System. Two Technologies Compared: NOR vs NAND. In *White Paper*, 2003.

[9] K9XXG08UXA datasheet. http://www.samsung.Com/products/semicondutor/flash/technicallinfo/datasheets.htm.

[10] Jesung Kim, Jong Min Kim, Sam H.Noh, Sang Lyul Min and Yookun Cho. A Space-Efficient Flash Translation Layer for Compact Flash Systems. In *IEEE Transactions on Consumer Electronics*, Volume 48, May 2002.

[11] UMass Trace Repository.http://traces.cs.umass.edu

[12] The Openmail Trace. http://tesla.hpl.hp.com/open source /openmail/.

[13] Numonyx Phase Change Memory (P8P) 128-Mbit Parallel Phase Change Memory. http://www.numo nyx.com.

[14] The Basics of Phase Change Memory (PCM) Technology. White Paper. http://www.numonyx.com.

[15] S. Raoux, G. W. Burr, M. J. Breitwisch, C.T. Rettner, Y.-C. Chen, R. M. Shelby, M.Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, C. H. Lam. Phase-Change Random Access Memory: A Scalable Technology. In *IBM Journal of Research and Development*, 2008.

[16] Cho. Beak-hyung, Lee. Kwang-jin, Park. Mu-hui. Phase Change Random Access Memory Device. United states patent 7639558. December 2009.

[17] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi and Kyoung Il Bahng. A PRAM and NAND Flash Hybrid Architecture for High-Performance Embedded Storage Subsystems. In *Proc. of EMSOFT'08*, October 2008.

[18] Jin Hyuk Yoon, Eyee Hyun Nam, Yoon Jae Seong, Hongseok Kim, Bryan S. Kim, Sang Lyul Min and Yookun Cho. Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture. In *IEEE computer architecture letters*, Volume 7, January-June 2008.

[19] Guangyu Sun, Yongsoo Joo, Yibo Chen, Yiran Chen and Hai Li. A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement. In *Proc. of HPCA'10*, January 2010.

[20] Youngwoo Park, Seung-Ho Lim, Chul Lee and Kyu Ho Park. PFFS: A Scalable Flash Memory File System for the Hybrid Architecture of Phase-change RAM and NAND Flash. In *Proc. of SAC'08*, March 2008.

[21] Open NAND Flash Interface SpecificaRion. revision2.2. http://onfi.org/wp-content/uploads/2009/02/ONFI%202_%20Gold.pdf

[22] Synchronous DRAM MT48LC64M4A2. http://www. micron.com.

[23] Chanik park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh and Wonhee Cho. A Reconfigurable FTL Architecture for NAND Flash-Based Applications. In *ACM Transaction Embedded Computing Systems*, Volume 7, July 2008.

[24] Nitin Agrawal, Vijiayan Prabhakaran, Ted Wobber. Design Tradeoffs for SSD Performance. In *Proc. of USENIX'08*, June 2008.

[25] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng and eng-Hsiung Hsu. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *Proc. of ISC'09*, June 2009.

[26] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proc. of FAST'08*, February 2008.

[27] Feng Chen, David A. Koufaty and Xiaodong Zh- ang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proc. of SIGMETRICS' 09*, February 2009.

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

[29] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proc. of FAST'10*, February 2010.

[30] Li-Pin Chang and Tei-Wei Kuo. Efficient Manangement for Large-Scale Flash-Memory Storage Systems with Resource Conservation. In *ACM Transactions on Storage*, Volume 1, November 2005.

[31] Luc Bouganim, Bjorn Por Jonsson and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proc. of CIDR'09*, January 2009.