Non-Volatile Memory Impact on Storage Systems

Course: CSCI 8980 - Fall 2013

Instructor: Dr. David Du

Submitted on December 10, 2013 By Group 11

Ziqi Fan (4629598), and Xiaoxiao Jiang (4281338)
I. Background and Introduction

In recent years, non-volatile memory (NVM) technologies have gained an increasing attention. Traditional systems use a two-level storage model, where DRAM is a kind of volatile memory that when system suffers crashes or power unexpected fails, the data stored in there would be lost. To remedy this kind of awkwardness, even flash is a kind of mature NVM, comparing to DRAM, it is too slow to replace the main memory. As a result of these needs, many researches are focusing on alternative non-volatile memory, which can provide low power, low latency, high capacity, and the ability to retain the data over long time periods. Basically, three types of NVMs are under fast development, which includes PCRAM, Memristor and STT-RAM. Comparing to DRAM, PCRAM has lower speed on write speed, while Memristor and STT-RAM have similar read and write latency with DRAM. STT-RAM also has better endurance like DRAM. Figure 1 shows the detailed characteristic comparisons of DRAM, PCRAM, Memristor and STT-RAM.

![Figure 1. Characteristics of different storage technologies](image)

Generally, there are five architectures have been designed based on NVM as shown in Figure 2. The first one is an hybrid architecture that DRAM and NVM work together as main memory (Figure 2.a). The second type is directly attaching CPU with a Storage Class memory, which is as quick as DRAM and meanwhile as large as HDD (Figure 2.b). In the traditional storage hierarchy, the additional overhead is ignorable since the latency to access storage devices is much higher than that to access memory. However, when the storage device is attached directly to the memory bus and can be accessed at memory speeds,
these overheads can substantially impact performance. The third and fourth types are using NVM to replace the DRAM in HDD and SSD system respectively (Figure 2.c, Figure 2.d). Last but not the least, NVM is inserted inside the SSD (Figure 2.e). As we known, a small DRAM is deployed inside SSD to accelerate address mapping access speed. Adding an even smaller NVM into SSD for address mapping update can extend lifespan of SSD.

![System Architecture](image)

**Figure 2. System Architecture**

The main objective of most previous works is to improve mechanisms designed for DRAM based system to fully explore the advantage of the non-volatile characteristic of main memory. In this paper, we focus on the fifth type that NVM is inserted inside SSD. We aiming at adding a smaller NVM inside SSD that works with SRAM to absorb frequent address mapping update and provide better system reliability and faster recovery.
II. Problem Definition and Motivation

Since most operating systems expect a block device interface even though flash memory in SSD does not support it, a software layer called FTL (Flash Translation Layer) is employed between OS and flash memory. In this way, a HDD can be simulated by SS. The role of FTL is manipulating the mapping tables and other data structures, which are stored in a small, fast DRAM. Basically, FTL mapping schemes can be classified into three types: page-level, block-level and hybrid FTL scheme. For the page-level FTL scheme, the logical page number can be mapped into any page within the flash. Therefore, it provides efficient utilization of blocks within the flash device. However, on the downside, such translation requires a large mapping table to be stored in DRAM. The block-level FTL scheme translates the logical block number into a physical block number using the mapping table. The size of the mapping table is significantly reduced comparing to page-level FTL. However, since a given logical page may now be placed in only a particular physical page within each block, the possibility of finding such a page decreases. As a result the garbage collection overheads grow. In order to remove the disadvantages from both page and block level scheme, a hybrid strategy have been designed in FTL. Most of the blocks are mapped at the block level while a small number updated blocks are mapped at the page level and are used for recording updates to pages in the data blocks.

Among all the FTL schemes we mention, DFTL [4] stands out. DFTL uses a pure page-leveling mapping mechanism that overcomes the large mapping table problem by the help of double layers of mapping instead of one.

Figure 3. Overview of DFTL
As shown in Figure 3, DFTL works as follows:

<table>
<thead>
<tr>
<th>Case 1: requested LPN hits in Cache Mapping Table (CMT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Done. Retrieve the mapping directly</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 2: a miss in CMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>If (CMT is not full)</td>
</tr>
<tr>
<td>then look up Global Translation Directory (GTD)</td>
</tr>
<tr>
<td>read the translation page</td>
</tr>
<tr>
<td>fill in CMT entry</td>
</tr>
<tr>
<td>goto case 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 3: a miss in CMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>If (CMT is full)</td>
</tr>
<tr>
<td>then select CMT entry to evict (LRU)</td>
</tr>
<tr>
<td>write back dirty entry</td>
</tr>
<tr>
<td>goto case 2</td>
</tr>
</tbody>
</table>

However, DFTL depends on SRAM to store its mapping table. SRAM is volatile, leading to reliability issue. If using low frequency of synchronization, it will lead to data loss with the occurrence of unexpected power outage. If using high frequency of synchronization, it will lead to extra SSD write which wears it out faster. We proposed to add an even smaller NVM into SSD that works along with SRAM to a) absorb address mapping update and b) provide better reliability.

**III. Related work**

Existing caching algorithms for flash memory can be classified into two main categories. One category operates on the page-level and another category operates on the logical block-level. The first category includes CFLRU algorithm [1] and the second category includes BPLRU [2], FAB [3] and CLC [5] algorithms. In the rest of this section, we will discuss each algorithm in detail.

Clean First LRU (CFLRU) [1] buffer management scheme exploits the asymmetric read and write speed of the flash memory. It divides the host buffer space into two regions: working region and eviction region. Victim buffer pages are selected from the eviction region. The size of these two regions varies based on the workload characteristics. CFLRU reduces number of write operations by performing more read operations. It chooses a clean page as victim instead of a dirty page, as write operation is more expensive than the read operation. When all the pages in the eviction region are clean, the victim is selected in the Least Recently Used (LRU) order.
Block Padding LRU (BPLRU) [2], Flash Aware Buffer (FAB) [3] and Coldest and Largest Cluster (CLC) [5] cache management schemes group the cached pages that belong to the same erase block in the SSDs into single block. BPLRU uses a variant of the LRU policy named Block Padding LRU (BPLRU) to select a victim in the cache. BPLRU manages these blocks in a LRU list. Whenever any single page inside a block gets hit, the entire block is moved to the Most Recently Used (MRU) end of the list. When there is not enough space in the cache, the victim block is selected from the LRU end of the list. In contrast to BPLRU, FAB considers block space utilization, which refers to the number of resident cached pages in a block, as the sole criteria to select a victim block. FAB evicts a block having the largest number of cached pages. In case of tie, it considers the LRU order. Since BPLRU only considers recency (i.e., LRU order) to select a victim block, for some write workloads where there are no or marginal recency, for example completely random write workload, FAB outperforms BPLRU. However, in general BPLRU outperforms FAB for the workloads which have even moderate temporal localities.

However, considering the shortage of FAB, Kang and et al. has proposed a novel write buffer-aware flash translation layer algorithm called CLC in [5]. In the CLC policy, both the temporal locality and cluster size are considered concurrently. Similar to FAB, the replacement unit of the CLC policy is also a page cluster with the largest cluster size among the cold clusters. To accommodate both the temporal locality and cluster size, there are two kinds of cluster lists: size-independent cluster list and size-dependent cluster list. The size-independent cluster list is sorted using LRU fashion and explores the temporal locality. The size-dependent cluster lists works similar to the FAB that is sorted by cluster size and explores the spatial locality for cold clusters. Pages will be inserted into size-independent list initially and move to size-dependent list if it is not frequently used. When the size-independent LRU cluster list is full and a new page cluster arrives, the page cluster in the LRU position of the list is evicted from the list and inserted into the size-dependent LRU cluster list with a corresponding cluster size.

Our proposed algorithm is different from all the above-mentioned work, because all their work focuses on data and our work focuses on metadata, which is mapping table. So in their cases, the cache size inside SSD is comparably bigger than ours. However, their algorithms can apply to ours without confliction.

**IV. Our Approach**

The architecture of our design is adding a small NVM into SSD that working along with a larger SRAM (or DRAM). SRAM is mainly used to accelerate access of mapping table. NVM is used as a write buffer for mapping table update as well as providing better reliability and faster system recovery.

Like DFTL, SRAM and NVM only store mapping tables or entries and don’t store actual data pages. We believe file systems have their own buffer cache to accelerate data page accesses. Due to limited space, SRAM will hold all of T1 mapping entries and partial T2 mapping entries.
Here, we define the Global Translation Directory used in DFTL as Tier One (T1) mapping and Mapping Table as Tier Two (T2) mapping. In our design, when an upper layer (e.g., file system) passes a Logical Page Number (LPN) to SSD to read a page out, we need to get where the page is located (i.e., get the page’s Physical Page Number (PPN)). So we need to check whether its T2 mapping entry stays in SARM. If it is, we can simply retrieve the PPN and return the SSD page to the upper layer. If the corresponding T2 mapping entry is missing from SARM, we need the help from T1 mapping table and fetch the corresponding T2 mapping page holding that T2 mapping entry into SRAM. Then we can get the PPN and return the SSD page to upper layer. However, due to the limited space of SRAM, we could only cache certain number of T2 mapping entries before it gets full. When it’s full we need to evict a page entry out. We can borrow LRU algorithm to do the job, which evicts the lease recently used entry out of SRAM. SSD read operation is shown in Figure 4.

![Figure 4. SSD read operation.](image)

When the upper layer passes an LPN to SSD to update a page, we need to find a free page to write data into. Then the previous SSD page becomes invalid so as its mapping entry. We need to link the LPN to the new PPN and form a new T2 entry and store in NVM instead of flash to save an SSD write. When the T2 entry is written into NVM successfully, we make a copy into SRAM. In this way, all the read requests will go to SRAM only. Here, we add a bit called @NVM in each SRAM’s T2 entry to denote whether each entry has a copy in NVM. If there is not, we can evict that entry from SRAM without any problems. Otherwise, we cannot evict that entry until its copy in NVM is synchronized to NVM and evicted. The reason is if we evict a T2 entry from SRAM with its copy at NVM, the next time we want to read that T2 entry, we will fetch it from flash, but it is an invalid entry as it is out of date.
When SRAM is full, we start to evict from LRU position. If the victim page’s @NVM is set, we need to sync and evict the copy on NVM before evicting SRAM’s. But due to sync the copy on NVM might take a long time, the victim at LRU would be skipped for now and check the next page towards MRU position. The process will be continued until one page is evicted.

In NVM, we reserve a large portion for T2 entries and a small portion for T1 entries. When T2’s portion is above a high watermark, say ninety percent, we start evicting entries. We can try different algorithms here to gain the best performance. Currently, we will cluster T2 entries which belong to a single T2 page. Then evict the largest cluster followed by the next largest cluster until T2’s utilization reaches low watermark, say sixty percent. In this way, we can save SSD write to the most. However, this algorithm does not consider recency and frequency, which is a future research direction. After T2 entries in NVM are evicted, we will set its corresponding T2 entry’s @NVM to 0 as its copy on NVM has gone.

After a T2 page is written into flash, its corresponding T1 mapping needs to be changed accordingly. The same as T2 entry update, we update T1 entry in the NVM first, then copy it to SRAM. When T1’s portion in NVM is above a high watermark, we start to evict and synchronize T1 to flash. SSD write operation is shown in Figure 5.

In our design, all the read requests will go to SRAM and write requests will go to NVM. One of the benefits is SRAM’s read access speed in general is faster than NVM’s. On the other hand, even though NVM’s write access speed in general is slower than SRAM’s, write requests do not need quick response as bad as read requests.
When power goes off unexpectedly, all the data stored in SRAM will be gone. But it will not hurt system’s reliability as they are all clean entry. Then, to recover from failure, we can simply copy all the entries from NVM into SRAM. Then SSD will work correctly afterwards. SSD recovery is shown in Figure 6.

**IV. Conclusion**

Among several FTL designs, DFTL stands out due to its flexibility of address mapping. However, DFTL relies on SRAM to accelerate mapping table access, which suffers from system inconsistency and frequent SSD write due to the volatile nature of SRAM.

In this paper, we propose adding a smaller NVM inside SSD that works with SRAM to absorb frequent address mapping update and provide better system reliability and faster recovery. In the future, we will investigate more on the eviction policy of T1 and T2 entries resided on NVM. We plan to use FlashSim and SSDSim to evaluate our proposed algorithm’s performance.
References