myFTL

Cundao Yu <cundaoy>

Overview

The aim of this project was to design and implement a Flash Translation Layer (FTL) called myFTL, which efficiently handles read and write operations, garbage collection, and wear-leveling on a solid-state drive (SSD). The focus was to create an FTL that maximizes performance, optimizes memory usage, and minimizes write amplification, while considering the challenges posed by different workloads.
Let's first talk about the overall control flow of myFTL.
For read, a flow chart Figure 1 is shown as below. First, LBA will be translated to PBA. If it is a valid PBA, we return this PBA and do read.

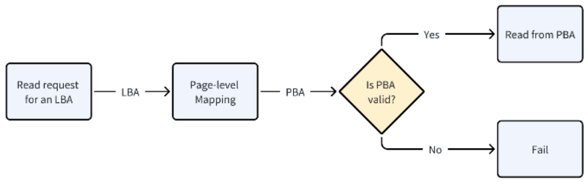

Figure 1 Read

For write, a flow chart Figure 2 is shown as below. It also first needs translation from LBA to PBA. And all writes go to the same block – writing block. And if the writing block is full and there's still unused initially empty block, we set writing block to the next initially empty block, otherwise we trigger garbage collection and get a new writing block. If there's older version for this PBA, related metadata also needs to be updated.

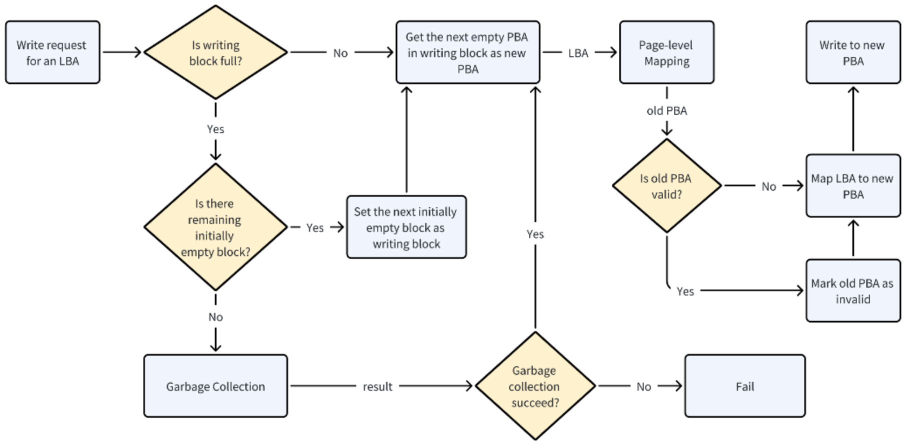

Figure 2 Write

For trim, a flow chart Figure 3 is shown as below. After translating LBA to PBA, we modified related metadata to indicate this LBA is trimmed.

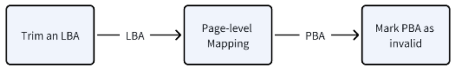

Figure 3

Before more detailed introduction of different components in following chapters, let's make some definition:

1. **Block_size:** Number of pages in one physical block.
2. **Block id:** identifiers for physical blocks, from 0 to total number of blocks on SSD.
3. **Page Index:** indices for physical pages in one physical block, from 0 to number of pages in one physical block (block_size).

Also, there're some variables that're used in many components:

1. **writing_block_id_:** Block id of the current block for writing. All writes go here. Initialized to 0 at the start of the program.
2. **cleaning_block_id_:** Block id of the block reserved for cleaning. Initialized to 1 at the start of the program.
3. **next_init_empty_block_id_:** Block id of the next initially empty block. Initialized to 2 at the start of the program.

Page-level Mapping

MyFTL uses a page-level mapping scheme for translating LBA to PBA. To achieve this, myFTL uses the following data structures or metadata:

1. **struct MyAddress:** Stores one uint16_t value to represent LBA or PBA. If it is PBA, use bit operation to extract block id and page index of the PBA. If it is LBA, it is just a compact form of LBA (only use 2 bytes). If the value is the max value of uint16_t ($2^{16} - 1$), then this MyAddress object represents an invalid address.

2. **struct PhysicalBlock:** Stores metadata of one physical block, which includes:
    a. **std::vector<MyAddress> mapped_logical_pages_:** One vector with size of block_size, indicates which LBA is mapped for each physical page. This helps modifying page mapping when the page is moved elsewhere or trimmed. If this MyAddress object is invalid (as described above), it indicates that this physical page is empty (not programmed/written yet) or doesn't contain valid data (outdated or trimmed).
    b. **next_empty_page_idx_:** One uint8_t value indicates the next empty page index in this physical block.
    c. **erases_:** One uint8_t value represents remaining erase count.
    d. **valid_page_count:** One uint16_t value indicates the number of valid pages (written and the newest pages) in this physical block.
    e. **lru_prev_block_id_:** One int16_t implicit LRU (Least Recently Used) doubly linked list "prev" pointer. Instead of being the memory address of the previous PhysicalBlock object, it is the block id of the previous physical block. -1 represents "nullptr". Will be discussed more in the later chapter "Wear leveling".
    f. **lru_next_block_id_:** One int16_t implicit LRU (Least Recently Used) doubly linked list "next" pointer. Instead of being the memory address of the next PhysicalBlock object, it is the block id of the next physical block. -1 represents "nullptr". Will be discussed more in the later chapter "Wear leveling".
    g. **ts_:** One size_t timestamp. It is the timestamp for last write or erase operation on this block.
3. **lba_to_physical_page_:** One std::vector<MyAddress> with size of total number of LBA, stores the newest PBA for each LBA. Use LBA as index to access this vector. If the MyAddress object for one LBA is invalid, this LBA is still not be written or trimmed.
4. **physical_blocks_:** One std::vector<PhysicalBlock> with size of total number of physical blocks, stores the PhysicalBlock object (which stores metadata for one block) for each physical block. Use block id as index to access this vector.

Then we can easily translate LBA to PBA by using LBA as index to access lba_to_physical_page_. After that, we can simply use PBA for read or use block id extracted from the PBA to access physical_blocks_ to get and update related metadata.
A Figure 4 is shown as below to illustrate this.

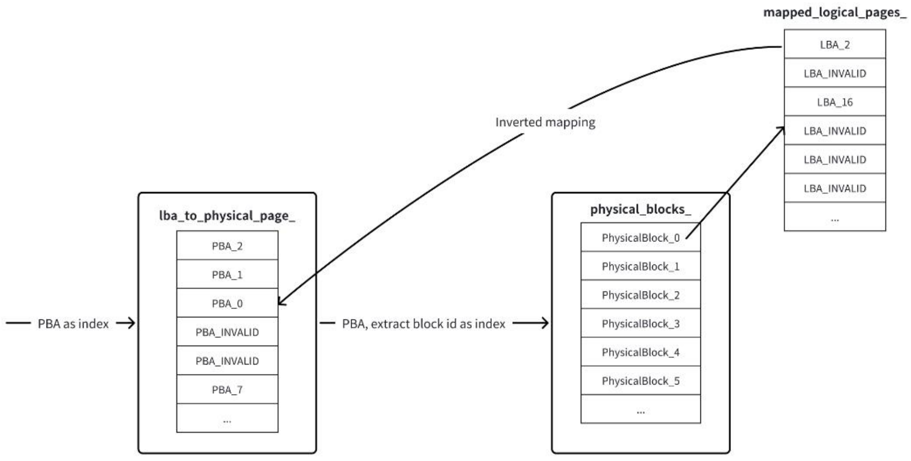

Figure 4 Page-level Mapping

Garbage Collection

MyFTL uses cost-benefit ratio as garbage collection policy. It iterates physical_blocks_ vector to calculate the cost-benefit ratio for each physical block. To calculate this ratio, we first need to calculate $utilization = (number\ of\ valid\ pages\ in\ block)/(total\ number\ of\ pages\ in\ block)$. Then we can calculate (cost benefit ratio) $= \frac{1-utilization}{1+utilization} * (current\ timestamp - block\ timestamp)$.
We choose the block with highest cost-benefit ratio as the victim block. We first move all valid pages in the victim block to the clean block and modify related metadata. Then erase the victim block. After that, we set the cleaning block as new writing block (since it still has space for writing) by setting writing_block_id_ to the id of the cleaning block. And we also need set the victim block as new cleaning block by setting cleaning_block_id_ to the id of the victim block.
A flow chart Figure 5 is shown in the next page to illustrate this. Cold data migration in this flow chart will be introduced in the next chapter.
By using such policy, blocks with low utilization will have relatively high probability to be chosen as victim. Low utilization means the block contains more invalid spaces thus is a good garbage collection target considering reducing write amplification. Also, such policy has can do some wear leveling, which will be talked about more in detail in the next chapter.

Wear Leveling

Cost-benefit Ratio policy achieves some wear leveling since the more frequently used blocks (has higher current_ts_ – block_ts_, probably has fewer remaining erases) have relatively low cost-benefit ratio thus is less probable to be chosen as victim. And the less frequently used blocks (has lower current_ts_ – block_ts_, probably has more remaining erases) have relatively high cost-benefit ratio thus is more probable to be chosen as victim. Besides that, myFTL uses two extra mechanisms to achieve better wear leveling:
1. **Cost-benefit ratio scaling based on remaining erases**
2. **Cold data migration**

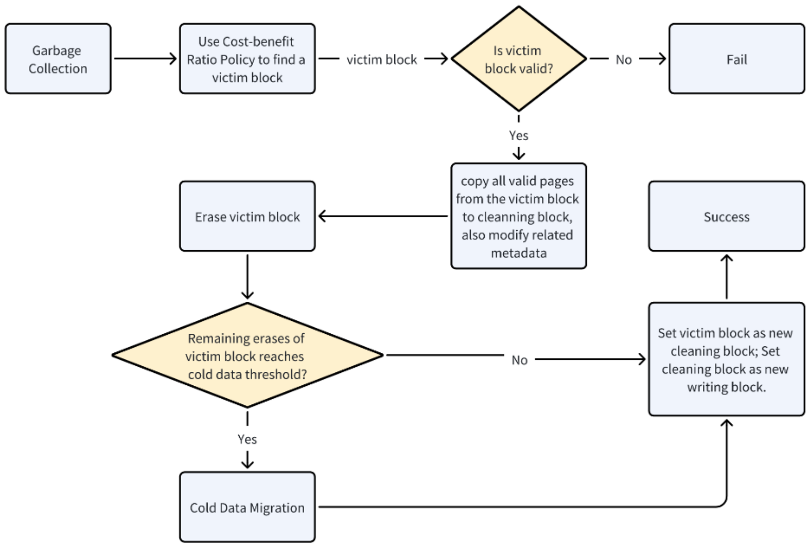
Garbage Collection
Use Cost-benefit Ratio Policy to find a victim block
victim block
Is victim block valid?
No
Fail
Yes
copy all valid pages from the victim block to cleanning block, also modify related metadata
Erase victim block
Success
Remaining erases of victim block reaches cold data threshold?
No
Set victim block as new cleaning block; Set cleaning block as new writing block.
Yes
Cold Data Migration

Figure 5 Garbage Collection

When calculating cost-benefit ratio, myFTL actually scales the ratio based on remaining erases: $\text{(scaled cost benefit ratio)} = \text{(cost benefit ratio)} * \frac{(remaining\ erases\ of\ block)}{(total\ erases\ of\ block)}$. By doing so, blocks that has been erased more will have smaller cost-benefit ratio thus has lower probability to be chosen as garbage collection victim. Relatively, blocks that has more remaining erases will be more likely chosen as victim.

Also, when the remaining erases of one block reach a lower threshold, myFTL will try to do cold data migration. This should happen when this block is garbage collection victim and is erased. MyFTL first try to find a suitable cold block, then move valid pages from the cold block into the victim block. After that, erase the cold block and set the cold block as new cleaning block by setting cleaning_block_id_ to the id of the cold block (As described in chapter Garbage Collection, the victim block will be the new cleaning block, but here the cold block replaces the victim block to become the new cleaning block).

So how do myFTL find a suitable cold block? Recall the data structure struct PhysicalBlock, it has a lru_prev_block_id_ and a lru_next_block_id_. These two block ids can be used as index to access physical_blocks_ vector to get the prev or next block. Then, we can maintain a LRU list for all physical blocks. When a block is written or erased, it will be moved to the head of the LRU list. Thus, the blocks at the tail of the LRU list will be the "coldest" blocks.

Then we can traverse the LRU list from tail to head to find the block that satisfies these requirements:

1. The block can not be worn out (It still has remaining erases).
2. The number of valid pages in this block should be greater than a threshold. This means if one block contains too few cold pages, such migration has low wear leveling effect and can incur high write amplification.
3. The block is "cold". A block is "cold" when its age (which is: $current\ timestamp - block\ timestamp$) is larger than a threshold.
4. Also, when two blocks both satisfy above requirements, we choose the one with more remaining erases.

When one block isn't "cold" during the traverse, we can just break the traverse since we are traversing from tail (least recently used) to head (most recently used) in a LRU list so later blocks should also be not "cold". If there's no suitable cold block, myFTL will give up doing cold data migration.

Some Discussion

Best and Worst Workloads

The best workload pattern should be that each LBA is written with the nearly same frequency. The worst workload pattern should be that some LBAs is written with high frequency, but some LBAs is only written with low frequency and are cold for a long time.

With page-level mapping, we don't care if LBAs are consecutive and can fill whole blocks, because all writes go to the same place and the LBA-PBA can be remapped easily. However, we do care that there are not too many cold pages since that will make it hard to find a block with many invalid spaces. If myFTL has to clean many half-full blocks, write-amplification will increase. Also, though myFTL has cold data migration and other mechanisms to do wear leveling, cold data migration will only be triggered once for each physical block and other mechanisms are just improving but not solving unevenly worn out. Thus, too many cold pages still have bad effect on wear leveling.

Metadata Compaction

As described in chapter Page-leveling, myFTL uses the minimally required spaces to store different metadata. Like since we only have at most 4 packages per SSD, 8 dies per package, 2 planes per die, 10 blocks per plane and 64 pages per block, we can use two bytes (uint16_t) to represent LBA or PBA.

Implicit LRU Doubly Linked List

MyFTL chooses to use an implicit form of LRU doubly linked list because memory address takes 8 bytes on 64-bit machine, which is a huge burden on memory usage. And since we have a static physical_blocks_ vector ("static" means block id will always be the vector index and there won't be element change), replacing memory address as block id can also construct a LRU doubly linked list. Block id only takes 2 bytes, so such implicit approach would save a lot of memory.