Solid State Disk Flash Translation Layer

HingOn Miu (hmiu)
Carnegie Mellon University

Introduction

Simply put, my translate function is intend to map a virtual address to physical address on a SSD during a read/write operation. Also, my algorithm does not rely on any certain SSD configuration, and it runs properly across all reasonable SSD configurations. The deeper purpose of my translate function is to prolong the life of SSD. Given that each block in SSD has a certain life span (erase limit), it is very essential that my algorithm does a good job to serve as many read/write as possible. Once a write fails, it signifies the end of life of the SSD. To handle this task, I have experimented and tested many different schemes.

1. Code Organization

First off, let's review the translation process of my algorithm. For a given SSD configuration, I have some range of physical address that belongs to the data blocks. The rest belongs to over provisioning blocks. Within the over provisioning blocks, there are only log blocks. Please note that I have decided to remove cleaning blocks, detailed reasoning is provided in a later section.

1.1. Algorithm

To initialize the translation process, I have to make sure the erase count of each blocks is zero. The logical address to physical address is mapped directly (evolves to dynamic mapping later on). All data blocks have no mapped log blocks, and all pages in SDD are in empty state. At last, the over provisioning area is partitioned into blocks are properly stored for later use as log blocks.

To service a read operation, my algorithm checks if the page is written. If the page is empty, the operation fails. Otherwise, my algorithm calculates the corresponding physical data block address. Then, it checks if there is a log block mapped to this data block. If there is not a mapped log block, I know that the latest page must be in the data block. If there is a mapped log block, it first check in the log block for the page. If the page is in the log block, I know that the latest page must be in log block. Otherwise, the latest page must be in data block.

To service a write operation, my algorithm first check if the page is empty. If the page is empty, it calculates the data block address and makes sure the first written page must go to the data block. Otherwise, I know that the page in data block must be written, and so it checks whether there is a mapped log block. If there is not a mapped log block, it attempts to map a free log block to the data block. If it fails to do so, my algorithm officially declares the SSD is over. If there is a mapped log block, it then attempts to find a free page in the log block. If it fails, then it attempts to clean the data block and the log block. If the cleaning fails, then my algorithm declares the SSD is over.

Before cleaning, my algorithm always attempt to remap data block and to remap log block when those blocks are close to erase limit. The remapping scheme is a bit complicated and will be discussed in a later section. For cleaning, my algorithm always attempt to find an empty data block to serve as a cleaning block. Essentially, latest copy of the pages from both data block and log block are moved to the empty data block, and eventually moved back to the data block. Detailed reasoning about the cleaning procedure will be provided in a later section.

The decision of declaring the SSD is over is not easy to make. This only happens when my translation process cannot find a space to write the new page. It is obvious to see that the

threshold of my algorithm lies in the amount of remaining free log blocks. Hence, the key of my algorithm is to generate more log blocks when free log blocks are running out.

1.2. Data Structure

My algorithm uses an unsigned int array with each bit indicating whether a page is empty or not. Another unsigned int array to record the erase count for each physical block. Also, I have a std vector to store all the remaining over provisioning blocks. More interesting, I have a signed int array that stores the mapping from logical block to physical data block, where each element is an offset such that the offset indicates the signed difference of the position between the logical block and physical data block. Similarly, I have a signed int array that stores the mapping from data block to log block, where each element is an offset such that the offset indicates the signed difference of the position between the data block and log block.

2. Policy Decisions

Below I would discuss a few essential decisions in my implementation.

2.1. No Cleaning Blocks

The original purpose of the cleaning blocks is reserved for moving pages around in the cleaning procedure. I see it as stealing over provisioning blocks away from log blocks. Cleaning blocks are useless and when there is no cleaning involved. To maximize the number of writes serviced, I decided to remove cleaning blocks. Then, what to use for moving pages around during cleaning? I decided to locate the cold data, which are the empty data blocks lying around. I use them as cleaning blocks to move data and erase them afterwards. Since they are originally empty, it does not increase write amplification of the cleaning procedure. It is true that this will increase the erase count of the cold data. Since I want to distribute the number of erase fairly among the physical blocks, it seems only fair to make use of the cold data.

2.2. No Cleaning Policy

With the remapping and shuffling scheme properly implemented, the cleaning policy is proven to be redundant by repeated testing as it only causes necessary write amplifications.

3. Optimizations

Through repeated testing, here are the optimizations in my algorithm.

3.1. Dynamic Mapping

There are two types of dynamic mappings in my algorithm. The first type is same block type remapping. My algorithm identifies the swap the cold and hot data blocks and swap the cold and hot log blocks. The main theory is to evenly distribute the erase count among all the physical blocks. Therefore, physical blocks with uneven erase counts will be remapped to prolong the life span of the SSD.

The second type is different block type remapping. My algorithm attempts to identify the cold data blocks and swap them with hot log blocks. Because of the limited amount of over provisioning blocks, it is essential to always maintain a sufficient amount of log blocks. Therefore, while the erase count of the log blocks are rising, it is wise to swap them with the data blocks with low erase count such that my algorithm only asks for a free log block unless absolutely necessary.

3.2. Write Amplification

To reduce unnecessary writes, I have decided to remove the cleaning policy. There are test results below to show that my algorithm's write amplification is usually very close to 1.00.

3.3. Wear Leveling

To keep track of wear leveling, I have managed to remap and shuffle physical blocks as described in an earlier section. The main theory is to distribute the number of erases among physical blocks as evenly as possible.

3.4. Memory Size

I have put many thoughts into reducing memory usage in my implementation. For starter, I do not use any structs that may cause fragmentations in memory if there are many of them with unaligned size. Also, I do not use pointers because pointers may be 64 bits in certain systems. Moreover, I do not dynamically allocate memory unless it is absolutely necessary (which is almost never). I always declare local variables such that they either on the stack or in registers and so they are freed automatically when that function returns.

More obviously, I use a bit to indicate the emptiness for each page. Besides, I have managed to reduce the size of my pages configuration in a log block by allocating it only as needed. Observe that I use std string for the pages configuration, and so my algorithm appends a new page to the string whenever a new page is written to log block. Hence, I do not pre-allocate the page configurations for all the log block initially, and so if no log block is used at all, there will be no memory consumption in pages configurations.

4. Test Results

Tests	# of W	W.A.	# E	Mem	Score
1	4276	1.000	5	1KB	1.037
2	38946	1.000	5	1KB	1.991
3	38946	1.000	10	1KB	2.000
4	38912	1.000	5	4KB	1.000
5	10305	1.034	5	1KB	0.023
6	77697	1.949	5	4KB	1.194
7	38946	1.000	5	1KB	1.991
8	10305	1.034	5	4KB	0.023
9	10274	1.000	5	136KB	0.895
10	38946	1.000	5	1KB	1.991
11	17654	2.855	5	1KB	0.057
12	77959	1.000	5	1KB	1.907
Total					14.1

5. Conclusion

I am most proud of my algorithm's memory usage since it is surprisingly low. It proves that those strategies I mentioned in previous section do work. With the unnecessary cleaning policy removed, I have managed to reduce the write amplification as well.

Through repeated testing, I have identified the weakness of my algorithm lies in the fact that the log blocks run out while there is no available data blocks to shuffle back. Looking back, I would develop a more sophisticated remapping scheme such that the shuffling is done in a more appropriate time and location.

The strength in my algorithm definitely lies in the remapping and shuffling scheme. Most tests provided issue writes randomly and evenly on all blocks in SSD. However, this is rarely the case in reality. If only a certain area of the SSD is written repeatedly, my algorithm can always identify those empty data blocks lying around and shuffle them to re-supply log blocks. Therefore, the remapping and shuffling scheme of my algorithm can prolong the life span of the SSD much greatly.