15-746 Project 1: myFTL
Handout 2

September 27, 2020

Contents

Overview

The last submission you make for Checkpoint 3 will be your final code submission. Note that there are a couple of days between the final code hand-in and the final project report hand-in. This will leave you a little time to think through and carefully explain your design decision. We strongly encourage you to not wait until the last minute before writing your report, because **your report is also graded**. We will also be **reviewing and grading your code quality (cleaniless, modularity, comments, no magic constants, etc)**. As before, submit your code to Autolab (`http://autolab.pdl.cmu.edu`).

Milestone	Description	Deadline
~~Checkpoint 0~~	~~Setup~~	~~11:59pm EST on Friday September 4, 2020~~
~~Checkpoint 1~~	~~Address translation~~	~~11:59pm EST on Thursday September 17, 2020~~
~~Checkpoint 2~~	~~Garbage collection~~	~~11:59pm EST on Sunday September 27, 2020~~
Checkpoint 3	Design FTL policies	11:59pm EST on Thursday October 8, 2020
Report	Final project report	11:59pm EST on Saturday October 10, 2020

Table 1: Deadlines for the four checkpoints and report.

Please refer to Handout 1 (the write-up you received for the first three checkpoints) for a review of the architecture of Solid State Drives (SSDs) and the Flash Translation Layer (FTL), the SSD emulator we are using in this project, debugging information, and a guide for working with AWS. In this write-up we are describing your objectives for Checkpoint 3, and the final project report. We also describe some guiding principles that you should keep in mind while designing your myFTL, explain the various parameters that form the grading metric we will be using to evaluate your myFTL design, and in Appendix 4 we provide some useful hints on how to optimize the memory footprint of your FTL design.

The Checkpoint 3 handout contains updated code and new tests for Checkpoint 3, so please replace all your code from previous checkpoints with the new code. This does not affect your implementation of myFTL.cpp; the changes are only for purposes of scoring the tests.

1 Checkpoint 3 - Design FTL Policies

Checkpoint 3 is about designing and developing an improved myFTL with wear leveling. Checkpoint 3 allows you to design the components of your myFTL, including the mapping scheme, the cleaning policy and the wear leveling policy, as you see fit in order to improve an overall metric score for your myFTL's design. The overall metric combines endurance, write amplification, and memory usage. In this checkpoint, **you can choose any mapping and garbage collection policies**. That is, if you choose to, you can change the mapping and garbage collection policies that you developed in Checkpoints 1 and 2 in the process of implementing wear leveling so as to improve on the metric. This mysterious "metric" is introduced in detail in Section 1.2.3.

At a high level, the objective of Checkpoint 3 (and indeed, of any reasonable FTL design) is to **maximize the number of possible writes** that can be performed by the SSD. Checkpoint 3 tests will keep issuing writes until your FTL returns the first failure to do so.

You are advised to keep your code modular to allow easy experimenting with different policies (for mapping, cleaning and wear leveling) as you try and achieve the better metric scores.

1.1 Guidelines for designing your wear leveling scheme

Following are some of the factors you may want to consider when implementing your wear leveling policy. Note that the factors mentioned below are by no means a complete list. You can choose to level the wear using any or none of the following factors, but you must describe your wear-leveling scheme in detail in the report and defend your choice of parameters for having done so.

Remaining life of a block. Given the number of erases that an SSD block is capable to withstand, the remaining life of a block is the remaining number of erases that can be performed before we can declare the block dead. In general, the remaining life of an SSD can be considered as the remaining life of its most worn out block. A good wear-leveling policy should try and reduce the number of erases being performed on blocks close to their death ^{RIP}.

Deviation from average remaining life. Since cleaning policies may influence wear leveling significantly, you could envision wear-based choosing of blocks for the purpose of cleaning. A possible metric to choose a block can be based on the deviation of the remaining life of that block from the average remaining life of the SSD (which is simply the average of the remaining life of all blocks of the SSD). Ideally, we want to try and avoid the SSD declaring itself defunct because a few blocks are significantly more worn out than others, i.e., we want to wear the blocks as evenly as possible.

Blocks containing cold data. One way of classifying data is by temperature. Hot data is data that is frequently or recently written, while cold data is data that has been written relatively long ago. You performed a similar exercise in the cost-benefit garbage collection policy in Checkpoint 2, where you identified the age of a block as the timestamp of the latest page written to that particular block. From a wear-leveling perspective, blocks occupied by cold data are usually the blocks that have seen very little wear. And cold data that will never be deleted or rewritten, means fewer opportunities for additional erase/write cycles the LBAs containing that cold data could provide. Choosing to shuffle cold data blocks may be a possible strategy of wear-leveling, but one that might complicate the mapping of LBAs to physical addresses.

Overprovisioned space wear leveling. Throughout Checkpoints 1 and 2, you have divided the flash capacity space into data blocks and overprovisioned blocks, subdivided into log-reservation blocks and cleaning-reservation blocks. The overprovisioned space itself can get more worn out than the data blocks. Relocating the overprovisioned blocks to a less worn out area can be a possible wear leveling strategy. Also, within the set of overprovisioned blocks, the log-reservation blocks might wear at different rates than the cleaning-reservation blocks. Shuffling blocks inside the overprovisioned space is another way to reduce wear. You can change the designation of blocks as data, log and cleaning as you see fit.

Selection of garbage collection policy. Keep in mind that the chosen garbage collection policy will result in different wear statistics for the SSD. As a result, the blocks chosen by your wear-leveling scheme may differ for different garbage collection policies. You get to choose which cleaning policy to use for Checkpoint 3, and you can design a new one, if you want. The garbage collection policy mentioned in config file of test cases can be ignored for checkpoint 3.

TRIM. We introduced TRIM in Checkpoint 2 to get you thinking about the ways in which it can be used for Checkpoint 3. We did not require you to implement any actual functionality for TRIM commands in checkpoint 2 and still do not require you to do so. However, in checkpoint 3 tests, we will be issuing TRIMs along with READs and WRITEs, and it is in your best interest to correctly implement TRIM command handling. Think about how handling TRIMs intelligently and exploiting discarded LBA and help you improve on the metric score.

Note/Hint: It is possible to obtain full credit on Checkpoint 3 with a version of the log-block scheme from Checkpoints 1 & 2, but we recommend exploring different options such as a page-mapping scheme if you are having trouble. Think about whether a page-mapping scheme is possible given the memory constraints detailed in Section 1.2.3.

1.2 Evaluation

1.2.1 Metric Parameters

The evaluation of your FTL's mapping, cleaning and wear-leveling policy is based on the following metrics:

- **Write amplification:** Write amplification is the ratio of the number of page writes performed by the SSD to the number of LBAs written by a workload. The ideal write amplification is 1. But, since we perform garbage collection and wear leveling, write amplification is usually >1. We will run various workloads against your SSD to understand the write amplification factor. The closer to 1 it is, the better.
- **Maximum writes observed:** Having wear leveling, the SSD should ideally be capable of performing more writes than in checkpoint 1 and 2. We will run various workloads on your SSD until it declares wear our failure and measure the number of writes accepted by the SSD. Usually, the larger the number of writes, the more effective your wear leveling algorithm.
- **Memory usage:** As mentioned throughout this handout, there is usually very little memory available on an SSD. A small memory footprint is another very important aspect you should keep in mind while designing your wear leveling scheme. Our tests will measure the total amount of memory used by your data structures while running a variety of workloads. In general, the lower the memory usage, the better.

1.2.2 Tests

All tests for Checkpoint 3 are **stress tests**. These are essentially infinitely long running tests, i.e. tests that will stop once they encounter the first failure return code from your implementation. The logs for these tests will also show the memory usage and write amplification.

These stress tests require that your FTL not fail to service a write request until at least one block has no erases remaining. In such a case, the score will be 0 for the test. A reasonable FTL would not fail to perform garbage cleaning when every block is still able to be cleaned.

1.2.3 Grading Function

Each test for checkpoint 3 will generate a score (out of 100). There are a total of 10 tests, which means that the maximum score you can achieve is 1000. Your goal is to maximize your combined score. Your grade from the tests is $total\_score - 900$ (out of 100). So a total score of 1000 gives full credit, and a score of 975 gives 75%.

The score for each test is composed of three subscores, based on the metrics described above. The three subscores are computed as following:

- **Write amplification score**: Ratio of "ideal" write amplification (e.g., 1.2 for a given test), to the write amplification of your myFTL design.
- **Endurance score**: Ratio of the number of write requests successfully completed by your myFTL, to the maximum number of writes possible with a "reasonable" implementation, which we have profiled for each individual test.
- **Memory score**: Ratio of your myFTL's memory usage[1], to the maximum usage for a "reasonable" myFTL design (= 320 KB for all our test cases).

[1]Remember, as described in the previous writeup, that our calculation of your myFTL's memory usage is accurate within +/-1 KB. So you might get a slightly different memory score on different runs with the same code. We do not expect a significant variance in your memory scores with the specified accuracy, so if that happens for you, please let us know.

We compute a weighted sum of the three scores and scale the final score to 100, which becomes your final score of the test. The weights associated to each score are as follows:[2]

Score type	**Weight for stress tests**
Write amplification	40%
Endurance	40%
Memory	20%

Table 2: Weights of the scores for each test. Scores of individual tests are simply summed to obtain your final score.

1.2.4 Code Review

Along with the the above mentioned metric, we will also be **reviewing and grading your code**. We encourage you to have a modular code design with well-named functions and variables and add detailed comments explaining your myFTL design, in addition to describing the same comprehensively in your report. We expect you to follow the 213 guidelines (`https://www.cs.cmu.edu/~213/codeStyle.html`). The report should also provide justification for the trade offs you made (see Section 3).

2 Submission

Your code should be submitted to the **myFTL Checkpoint 3** project in Autolab in the same way you submitted your code for Checkpoints 1 and 2. You can only modify the `myFTL.cpp` file. When you want to submit the code on autolab, you should only submit this file. Checkpoints 3 has **25 unpenalized submissions**. For each additional submission (after the initial unpenalized ones), you will be penalized 10% of your checkpoint grade. Note that tests are not backward compatible; after you develop Checkpoint 3, some of the tests for previous checkpoints might fail. As before, you can use maximum of 2 grace days for this submission (out of 3 total grace days that you have for the entire semester). After you have used up your grace days, you will be penalized 15% of you checkpoint grade for each late day.

3 Report

The report should be a PDF file no longer than 3 pages in a single column, single-spaced 10-point Times Roman font with the name `AndrewID.pdf`. Your report should be a self-contained document that does not depend on the writeups that we provided. That is, you should briefly describe the aim of the project in the report, rather than assuming that the readers know it because they have read the writeup. You should also add your Name and Andrew ID in the report.

The report should justify the design choices you made for Checkpoint 3. You should describe the policies you used for different functionalities, i.e. mapping, garbage collection and wear-leveling, that you designed and implemented.

Remember to include a discussion on how you think your myFTL design would fare in the face of different workloads, e.g., which is the best workload for your design, and which is the worst? The report should also describe the key data structures and code structure; this would help us with your code review. Feel free to add any specific optimizations that you used so as to maximize your score on the grading metric.

You can also include additional evaluation results from your own test cases, in addition to the ones we gave out and those we used to mark your Autolab submission. Lastly, we would also appreciate any feedback you want to share about the project (so we can lessen the suffering of those that come after you!). If you used any of the optional features mentioned in the previous writeup's appendix, such as memory tracing, SSD visualization, and performance testing, make sure to mention that. Feel free to let us know of other features you think should be part of the simulator to help better design an FTL.

[2]If you are interested, you can checkout the exact grading function in 746FlashSim.h.

4 Appendix: Notes on optimizing memory usage

In this appendix, we share some hints on how to optimize the memory usage of your myFTL design. We also note some pitfalls which you might encounter while accounting for your myFTL's memory usage.

Example configuration: Let us begin with an example. Consider an SSD with the following configuration: 5 packages, 2 dies per package, 2 planes per die, 32 blocks per plane and 64 pages per block. Let us do some back of the envelope calculations for the memory space you would need for (say) the hybrid log block mapping scheme that you have implemented in Checkpoint 1 and 2[3]. For the hybrid log block mapping scheme, you need two mappings: first from data blocks to the log blocks and second from pages in the data blocks to pages in the log blocks.

Memory usage for a naive implementation: Let us start with a naive implementation and then optimize it for memory. In a naive implementation, there would be a mapping from each data block to each log block. In the above example, there are 640 blocks (some of which would be overprovisioned blocks and hence wouldn't need a mapping, but lets ignore that for now as we are considering a naive implementation). For each of these (data) blocks, there would a a corresponding 4-tuple which identifies the address of the corresponding log block. Assuming that each member of the tuple can be represented with a 4 byte integer (i.e. the number of packages, dies per package, and so on all fit in 4 bytes, which is true in the example here), the size of each 4-tuple is 16 bytes. With a 16 byte mapping for 640 blocks, the space required for the data to log blocks mapping is 10240 bytes or 10 KB. Using similar logic for the data block pages to log block pages mapping (4 byte per tuple member, 5 tuple address to identify a page and a mapping for 40960 pages), the space required for the same would be 800 KB. The total memory usage for the two mappings combined is thus 810 KB.

I'm a responsible programmer. Can I consume *less* memory?: Now let's think about optimizing the memory usage. An obvious optimization is based on the observation that at any given point of time, neither of the mappings (say, for example, the data to log block mapping) need to be hold values corresponding to all the possible entries (i.e. the data to log block mapping doesn't need to be defined for all the data blocks) because at no given point of point would all data blocks have a corresponding log block. This in itself could drastically reduce the memory usage depending on the overprovisioned space.

Another possible optimization, independent of the first one, is based on the observation that storing 4-tuple and 5-tuple addresses, for blocks and pages respectively, is not required. In the configuration of our running example described above, the total number of blocks is 640. If these blocks were to be numbered and addressed sequentially from 0 to 639, all the blocks can be address using just 2 bytes. Even ignoring the first optimization described above, the mapping for 640 blocks with 2 bytes per block is just 1280 bytes or 1.25 KB, down from 10 KB in the earlier naive implementation. Similarly, you don't need to use a 5-tuple to identify a page. In fact, in hybrid log block mapping scheme, a single byte can be used for mapping each page in the data block to the page it maps to in the log block (since there are only 64 pages per block, which can be numbered and addressed using a single byte). Hence, again without using the above optimization and assuming that the mapping holds all the 40960 entries, with 1 byte per entry (per page), the total memory required is just 40960 bytes or 40 KB, down from 800 KB in the earlier naive implementation. Note that in order to fully benefit from this optimization, you should **use the smallest possible data type for the variables. Think along the lines of: do I need an `size_t` for this variable or would a `short` suffice?**

Using an optimized representation, for example, a 2 byte representation for the SSD blocks, does not lead to any loss of information. The linear addressing can always be translated to the 4-tuple address using simple division and mod operations, which are commonly used in any storage system. You are encouraged to spend some time thinking about how you can use this optimization. Also note that the optimization has been discussed here in the context of hybrid log block mapping scheme but there is nothing special or exclusive about this particular mapping scheme. Indeed, similar optimizations can be used for other mapping schemes that you might want to use or design.

I'm obsessing about memory consumption a little. Can a real FTL consume *so little* memory?: While the above discussion points to possible optimizations, it is also worth noting that things are different in a real implementation; your memory usage is affected by other factors too. In particular, the actual memory usage of you implementation is almost surely going to be more than what you compute using back of the envelope calcualtions like shown above. And there are many reasons for the same. For example, one possible reason for why your memory usage can be higher than what you would expect from such calculations is internal fragmentation in malloc

[3]Note again that we do not require you to stick to hybrid log block mapping scheme for Checkpoint 3 (however, you are free to do so if you choose to). We are simply using this scheme in the example because we expect everyone to be familiar with this mapping scheme by now.

(remember malloc lab?). Every call to malloc can potentially lead to more memory being accounted towards your myFTL than what it asked for in the call to malloc. Another potential reason for the memory usage bloating is C++ libraries. The libraries (for e.g. std::vector or std::map) are not necessarily optimized for memory and can be lead to more memory being used than expected by back of the envelope calculation. **Be careful and cognizant of the memory usage while choosing the data structures to use for your myFTL design. Consider using data structures with fixed upfront sizes as opposed to data structures that grow dynamically and can lead to inefficient memory utilization (think along the lines of using a std::vector or array of a fixed size declared upfront as opposed to a std::map).** Lastly, in an actual implementation, you would also be storing additional metadata (such as valid/invalid status) and those would also contribute towards myFTL's total memory consumption. You would want to be cognizant of all such effects while thinking about your myFTL's memory usage. Also, you might find the memory tracer and SSD visualization feature of Flash Simulator useful in analyzing your FTL (These were explained in previous writeup's appendix).

Maximum values of SSD configurations: Implementing the optimizations discussed above is possible only if the maximum value of, say, the number of blocks or pages is known in advance. In order to allow such optimizations, we are letting you know the maximum values of the following SSD parameters:

- Maximum number of packages in the SSD (`SSD_SIZE`) is 4
- Maximum number of dies per package (`PACKAGE_SIZE`) is 8
- Maximum number of planes per die (`DIE_SIZE`) is 2
- Maximum number of blocks per plane (`PLANE_SIZE`) is 10
- Maximum number of pages per block (`BLOCK_SIZE`) is 64