FTL with Oracle

Shangda Li
Carnegie Mellon University
Pittsburgh, PA, 15213
shangdal@cs.cmu.edu

Sheng Xu
Carnegie Mellon University
Pittsburgh, PA, 15213
shengx@cs.cmu.edu

Tianjun Ma
Carnegie Mellon University
Pittsburgh, PA, 15213
tianjunm@cs.cmu.edu

ABSTRACT

The flash translation layer (FTL) of an SSD is responsible for mapping logical LBAs to physical pages, and performing garbage collection and wear leveling. A key performance metric of an FTL is Endurance, the number of write requests endured by the FTL until it declares failure. Current state-of-the-art FTL algorithms are not workload-aware and they do not achieve workload-specific optimal page-allocation that can maximize the Endurance of SSDs. We conjecture a superior workload-aware FTL algorithm that can maximize SSD lifetime exists, though devising such an algorithm is difficult and likely involves the use of machine learning. To prove the feasibility of such a workload-aware algorithm, we implement an oracle with complete knowledge of a workload, which can provide hints to the FTL to help the FTL make smarter decisions. We mainly explore two types of hints: various hints to tell the FTL the hotness/coldness of LBAs, and various hints about the optimal internal parameters used by a FTL, such as the block threshold for cleaning. We also implemented an FTL baseline that combines approaches from multiple state-of-the-art FTL algorithms. By showing that the FTL baseline with oracle achieves 20% to 25% higher Endurance than just the FTL baseline, we demonstrate that it is possible for a workload-aware algorithm to outperform state-of-the-art FTL algorithms. We conjecture that a workload-aware algorithm can substitute hints from the oracle with hints deduced on its own, possibly using methods such as machine learning.

1 INTRODUCTION

Background

Solid-state drives (SSDs) based on NAND Flash memory have long been the state-of-the-art non-volatile storage medium. Compared to hard-disk drives, SSDs have faster random I/O speed and can serve multiple requests in parallel. SSDs are widely used in smart phones, personal laptops, data centers, and high-end computing. Internally, an SSD is a collection of physical pages, organized in a hierarchy of packages, dies, planes, and blocks. These layers are hidden to the upper layer, such as file systems and user programs, which view the SSD as a series of Logical Block Addresses (LBAs). Each LBA corresponds to the size of a physical SSD page. Several key characteristic of SSDs are

(1) No overwriting on a SSD page is possible, so every new write must occupy a new page.
(2) Erases can only be done on the granularity of blocks, so valid pages must be migrated before erasing a block.
(3) Each SSD block can only tolerate a limited number of erases, called its lifetime. If a particular block is updated and erased frequently, it will wear out before other blocks and make the SSD inoperable.

Therefore, using direct mapping between LBAs and physical pages is not possible, and SSDs come with a flash translation layer (FTL). The FTL is responsible for address translation (logical-to-physical mapping that maps LBAs to physical pages), garbage collection (erasing blocks to free up space when the SSD is filled up), and wear leveling (load balancing writes across blocks to extend the lifespan of an SSD). Thus the FTL policy for SSDs is very similar to the Segment Cleaning Policy we read about in the Log-Structured File System [1]. Similar to LFS, an FTL cleans with block granularity, migrates pages, and keeps track of page locations.

Motivation

There are numerous varying FTL algorithms. Even though some of them contain dynamically adjusting parameters, e.g. maximum distance between the healthiest and the most worn out block, their core algorithms and the way their parameters adjust is the same across different workloads, where each workload is an online stream of LBA requests sent to the FTL, including READ, WRITE, and TRIM. However, we conjecture that there are different patterns that modern computer programs exhibit in their LBA traces, and for each of these patterns some FTL algorithms perform better than others, and within the same algorithm some combinations of the parameters perform better than others. Such a superior workload-aware algorithm that outperforms state-of-the-art FTL algorithm can extend the lifespan of SSDs, which is desirable given the wide use of SSDs.

However, devising such a workload-aware algorithm is difficult. We speculate that the algorithm is not deterministic and might require machine learning. In this paper, our goal is less ambitious: we aim to show that a superior workload-aware algorithm is feasible. We implement an *oracle*, which knows the entire workload upfront, and uses this complete knowledge to provide *hints* to the FTL. If we show that FTL algorithms can perform better with *hints* from an oracle, then this can serve as a proof-of-concept that there is room for a

superior workload-aware algorithm. The actual workload-aware algorithm will not have help from an *oracle*, and would need to deduce *hints* on its own based on the part of the workload it has already seen. This deduction of *hints* can possibly be done by a machine learning algorithm, which can use the stream of LBA requests as the input and the decrease in lifetime of SSD as the error function, to continuously adapt itself to the workload.

Note that since a workload-aware algorithm cannot modify the workload, the *oracle* is not allowed to modify the workload either. If the *oracle* is allowed to modify the original workload to an equivalent workload, it could analyze the final value of each updated LBA after running the original workload, and devise a new workload to include only a single update to each of these LBAs with its final value. This new workload, though trivially optimal, is not allowed.

Apart from being a proof-of-concept for a superior workload-aware algorithm, the *oracle* might also be useful if a layer above the SSD ,such as the operating system, has some degree of knowledge about the workload and can act as a weaker version of the *oracle* to provide *hints* to the FTL.

Contribution

In this project, we will explore the feasibility of a workload-aware, self-adjusting FTL algorithm that utilizes machine learning, by demonstrating the endurance and lifetime improvement to FTL brought about by *hints* from an *oracle* with complete knowledge of the workload. Specifically, our contributions will include:

(1) Building a FTL baseline that combines the techniques from multiple state-of-the-art FTL algorithms
(2) Show that the FTL baseline can perform better with *hints* from the *oracle*. Explore different levels of *hints* and their corresponding performance improvements.
(3) Evaluate the performance of the FTL baseline with and without the *oracle*, under multiple real world traces and practical scenarios, such as with SSD warmup and TRIMs.

The outline of this paper will be: in section 2 we will discuss related work in FTL algorithms, in section 3 we will discuss our FTL baseline and *oracle* design and implementation in detail. In section 4 we will discuss evaluation of our FTL baseline with and without *oracle*, and with different *oracle hint* types, under several different workloads and different scenarios. In section 5 we will discuss future work and extensions to our project. And section 6 is the conclusion.

2 RELATED WORK

There are numerous algorithms proposed for FTL garbage collection and wear leveling. Earlier algorithms were mainly block-level schemes that maintained mapping from logical blocks to physical blocks, the most notable being [2], which redirected overwrites in data block to reserved log blocks. However, if requests arrive evenly over many data blocks, log blocks need to be swapped for data blocks regularly and erased even when they are quite empty, causing high write amplification. [3] proposed FAST, which maintained page-to-page mapping that allowed for more flexible allocation of pages. Since then, many page-level schemes have been proposed to reduce wear-leveling and improve SSD lifetime, including [4], [5], and [6], each claiming to be superior to the previous ones. However, these papers use different workloads when comparing against each other, and none of them prove that they achieve theoretically optimal write-amplification and maximum endurance for every workload. In fact, whether such an optimal algorithm exists is still an open question. Recently, [7] proposed Balloon-FTL, which uses a genetic algorithm to do workload-aware wear leveling for dual-mode TLC/SLC SSDs. However, their focus is on load-balancing between TLC and SLC-mode blocks, and they do not compare to the theoretical optimal.

LazyFTL claims that it is the best known page-level algorithm in terms of write amplification, while Rejuvenator claims that it is superior to popular block-level schemes such as Dual-pool[8] and [9] in terms of write amplification and SSD lifetime. Thus our FTL baseline is built from a combination of their techniques. Similar to Rejuvenator[4], we maintain the invariant that the max erase count and min erase count among data blocks is within a certain threshold D, and D is adaptively decreasing throughout the lifetime of the SSD. Similar to LazyFTL[6], we always maintain two special blocks within our free pool: CDB, which has the min erase count among free blocks, to accept hot pages; and CCB, which has the max erase count among free blocks, to accept cold pages. The idea is to maintain more frequently accessed pages in less worn blocks and less frequently accessed pages in more worn blocks in order to control the variance in erase counts of the blocks. Moreover, our FTL baseline also uses ideas from papers such as [10] to identify hot and cold pages using a fixed threshold. Table 1 provides a summary on how our FTL combines techniques from these papers.

3 APPROACH

Overview

As mentioned in the introduction, we will start with introducing the baseline FTL. We first list the metric we will be using in assessing the performance of an FTL algorithm. Secondly, we will talk about the *oracle* mentioned at the beginning in more detail, and discuss the methods to establish the superiority of the FTL baseline with access to *oracle* over that of the baseline itself.

	Special blocks for hot/cold data	Erase count Distance
Dual-pool		
Rejuvenator		✓
LazyFTL	✓	
FTL Baseline	✓	✓

Table 1: FTL baseline vs various existing approaches.

Note that in our project proposal we also mentioned the possibility of directly predicting the best policy to switch to, instead of the values of the parameters of a fixed baseline. Since different policies sometimes maintain drastically different data structures and do page-to-page or block-to-block mapping in distinct manners, a considerable amount of overhead will be introduced if we switch from one specific policy to another, and we believe such overhead is hard to resolve both within the scope of the course project and in the long term. Thus we choose to use a single baseline FTL algorithm, but add multiple tunable parameters so that it is representative of a wide class of FTL algorithms.

Metrics

Several common metrics used to evaluate the effectiveness of a FTL policy include:

(1) **Write amplification(WA):** the ratio of number of physical page writes performed internally by the SSD to the number of write requests sent to the FTL. WA is high if the FTL performs a lot of internal page migration. Ideally this should be as close to 1 as possible.
(2) **Endurance:** the maximum number of write requests successfully completed by the SSD before it declares failure. This is essentially the lifetime of the SSD and should be as large as possible.
(3) **Memory usage:** There is usually very little memory available to the SSD, so FTLs must have a small memory footprint.
(4) **Throughput:** The throughput of FTL should not become the bottleneck of SSD I/O operations.

Our primary metric to determine the performance of a FTL model is **Endurance**. We will also track **WA**, but **WA** is already inversely correlated to **Endurance**: if **WA** increases, **Endurance** decreases since blocks are erased more often. Moreover, **WA** is not reflective of wear leveling: to keep **WA** low, we might as well do no wear leveling at all. Thus we will focus on optimizing for **Endurance**. We will also measure **WA**, **Memory usage**, and **throughput** of our SSD to make sure they are reasonably small.

FTL Baseline

Our baseline FTL implementation is modified based on Sheng's scoreboard rank 1 FTL implementation from the course 15746. It combines ideas from Rejuvenator and LazyFTL to perform garbage collection and wear leveling.

Our FTL uses page-to-page mapping, which is more flexible than block-to-block mapping and better for Endurance. It allows us to group pages from different logical blocks within the same physical block. For all pages, we track number of queries since the last access to a page, and use a threshold to identify hot and cold pages. There are two kinds of physical blocks: data blocks and free blocks. We use a specific free block, CDB (current data block), to accept hot pages both from write requests and page migration when cleaning blocks, and another free block, CCB (current cold block), to accept cold pages. When CDB fills up and becomes a data block, we pick the free block with the lowest erase count to be the next CDB; and for CCB, we pick the free block with the highest erase count, in the hope that we won't erase it any time soon since CCB will be filled with cold pages. Whenever the number of free blocks drop below LT, we erase blocks until we have HT free blocks. We will stop erasing if no available block has more than half a block of invalid pages. We choose the block to clean next according to the following policy:

Let *dist* be the distance between max and min erase count among data blocks

- Case 1: If $dist < D$: we pick the block with max invalid pages among all data blocks
- Case 2: If $dist >= D$: we pick the block with max invalid pages among data blocks with min erase count

D is a large number in the beginning, and is adaptively decreased to 1 as the max erase count increases. The intuition is that we want to perform Case 1 more often than Case 2 since Case 1 chooses the optimal block to clean among all blocks, and initially a large D allows us to choose Case 1 most of the time. As we near the end of an SSD's lifetime, we want D to be smaller, because we don't want some blocks to wear out much faster than the others. Finally, in the case where the system is quite full (many LBAs in the SSD) and there are repeated sequential writes to a few hot blocks, we enter a special mode, SWAP MODE, where we ignore D and always pick the data block with max invalid pages to clean, and we swap hot free blocks with cold data blocks after cleaning to HT free blocks.

To expand on Sheng's original implementation, we made several changes to support *oracle* and run experiments for this project:

- The original framework supports only 40960 pages. To run our workloads we modified the framework to support an arbitrary number of pages, limited only by memory size.

- The garbage cleaning process from the original implementation had runtime linear in the number of physical blocks N. We modified the data structures so that the runtime is $O(\log N)$.
- The workloads come in the format of (starting LBA, number of bytes accessed), but our original model takes in 1 LBA at a time. We now allow batch inputting of sequential LBAs from the tests.
- We added the capability for our FTL to take in parameters LT, HT, D from the *oracle*, and added functions that returns to the *oracle* information about the FTL, such as the max erase count, erase count sum, and time till next cleaning period.
- We allow the FTL to either use its own threshold to determine whether an LBA is hot or cold, or ask the *oracle* by calling a function.

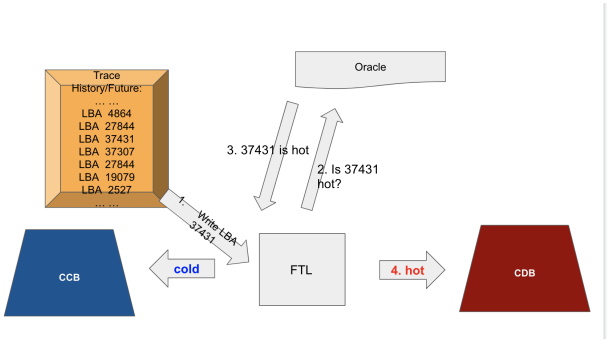

Figure 1: The workflow of FTL with *oracle*. When receiving an LBA from a write request or doing page migration, the FTL will ask the *oracle* whether the LBA is hot or cold, and place hot pages in CDB, and cold pages in CCB.

It should be noted that since SSDs typically have a small DRAM, it would be impractical to fit the full page-to-page mapping table in memory if the SSD is large and there are billions of pages. The solution to this problem has been extensively studied, such as in [6], which proposes the idea of storing the global PPN (physical page number) to LPN (logical page number) mapping in the flash memory, and only storing the reference to that mapping in the DRAM, which offers a level of indirection that greatly reduces the memory consumption of storing the page-level mapping, while preserving the flexibility it offers. Thus for the purpose of this project, we will assume that we have enough memory to support full page-to-page mapping.

Hot/Cold Page Segregation

Correctly distinguishing hot and cold pages (LBAs) is of significant importance to the performance of FTL. Hot pages are LBAs that are frequently written, meaning that once they are written to the SSD, it will not take long before they are written again and their corresponding physical pages are marked as invalid. Cold pages are LBAs that are infrequently written, meaning that they will remain valid physical pages in the SSD for a really long time. A good FTL algorithm should always cluster the hot pages together in the same blocks for two main reasons. First, when the next cleaning time arrives, the physical block being cleaned would have almost all its physical pages marked as invalid. In this scenario, we only need to migrate a very small number of valid pages in that block, which is very efficient and leads to close-to-one Write Amplification. Second, a good FTL algorithm can choose the healthiest blocks (the blocks with minimum erase counts) to store these hot pages, because those blocks would need to be cleaned again very soon. Thus, all the physical blocks in the SSD can wear off evenly. Better Write Amplification as well as evenly wearing of all the physical blocks are essential to the Endurance of SSD.

With the full knowledge of the entire workload, at any point in the the *oracle* can classify each incoming LBA as either a hot page or cold page. The general workflow is shown in Figure 1. When the FTL needs to determine whether a LBA is hot or cold, during a write request or page migration, it can ask the *oracle*, which will pass the hot/cold segregation as a *hint* to the FTL algorithm. Then arises two natural questions. Given the trace, what should be best definition for hot pages and cold pages? Should the definition be based on absolute access timestamp, or relative access frequency?

We observe that to classify every individual LBA write request as hot or cold, it is sufficient to know the timestamp for the next request for the same LBA (note that we advance the timestamp whenever the FTL accepts a new LBA request). In this manner, the hot/cold definition is tied to a LBA along with a timestamp instead of just a LBA, and so can be dynamic for the same LBA across the entire workload. For example, if the workload contains a burst of write requests for the same LBA, then the first of these write requests is definitely a hot page, because the corresponding physical page will be marked as invalid soon after that write request is done. On the other hand, the last of these write requests should be classified as a cold page, because as the burst ends, the physical page due to the last write will remain valid for a long time. Therefore, the "hotness" of the same LBA is dynamic across the entire workload trace. We explore three options for the criteria of distinguishing hot/cold LBA, based on the timestamp for the next request for the same LBA.

The first option (**OPTION 1**) we explore is based on the timestamp difference between the current and the next write request for the same LBA. For a given write request at timestamp t, for some LBA l, we scan through the workload trace to find the next write request to l after this write request. Let

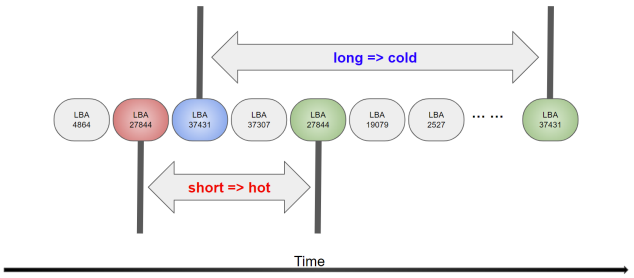

Figure 2: OPTION 1, Time period until next accessed below/above certain threshold.

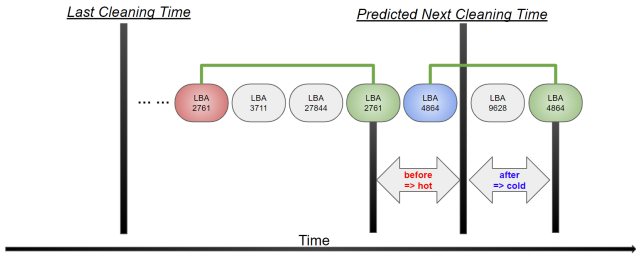

Figure 3: OPTION 2, Whether the next time accessed is before/after the next cleaning time.

t' denote the timestamp for that "next write request to l". We calculate $(t' - t)$ and compare it against a pre-determined threshold. If it is below the threshold, then it is considered as a hot page write request. Otherwise it is considered as cold. This idea is illustrated in Figure 2.

However, we find that the performance of the first option is highly dependent on the hotness threshold. For some workload with bursty write requests to a small pool of LBAs, the performance is great when the threshold is set around 10, whereas for workload with accesses spread out among a large number of LBAs, the performance is terrible with 10, and the performance is much better when the threshold is set to a larger value. Although we can tune the hotness threshold using the method discussed later in Subsection 3, this option is intrinsically inflexible and cannot give the best hot/cold definition in most workloads, which have dynamically shifting access patterns.

Thus, we want to incorporate dynamic information about the current SSD's blocks allocation information to obtain a better definition for hot/cold pages. The second option (**OPTION 2**) we explore is based on whether the LBA for the current write request will be rewritten again (and hence marked as invalid and can be efficiently cleaned without migration) before the next *cleaning period*. Note that under the context of our baseline FTL algorithm, a *cleaning period* starts when the number of free blocks drop to LT, and does not end until we have HT free blocks or until there are

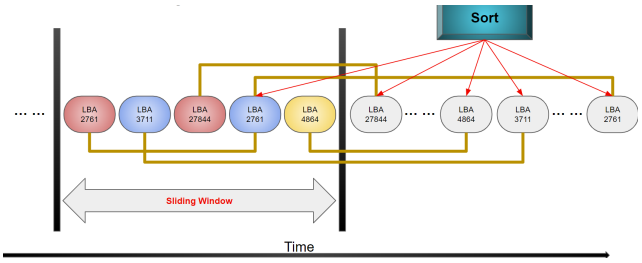

Figure 4: OPTION 3, Whether the next time accessed is before/after the median next accessed time of LBAs within a sliding window. The red LBA requests are hot. The blue are cold. The yellow is the median.

no more good candidates for cleaning. Therefore, the time till the next *cleaning period* is known exactly: for a given write request at timestamp t, the next cleaning timestamp is calculated as

$$\text{time\_till\_cleaning} = (F - LT) * P - CDB_P - CCB_P$$

, where F is the number of free blocks, P denotes the block size (number of pages in each block), and CDB_P, CCB_P denote the number of valid pages in CDB and CCB respectively. We call the period between two *cleaning periods* a *normal period*. All write requests are processed during the *normal period*, within which we know exactly the time till the next *cleaning period*, and all page migrations are processed during a *cleaning period*. So for **OPTION 2**, similar to the first option, we find t' as the timestamp for the next write to the same LBA after t. We classify the current LBA write request as hot if and only if

$$t' < t + \text{time\_till\_cleaning}$$

This idea is illustrated in Figure 3.

When we experiment with the second option, we observe that the majority of pages are classified as cold pages, due the fact that for most workloads, unless the free pool size F is really large, *normal periods* are pretty short once the SSD is written to full for the first time. As a result, only a few LBAs will be written again before the next *cleaning period* and this criteria is too restrictive for a LBA write request to be identified as hot, which results in a very skewed hot/cold page writes distribution. If we can only find a very small percentage of hot blocks, our hot-cold segregation is ineffective since the majority of pages get written to CCB and CCB gets populated faster than CDB (not ideal since CCB is actually the free block with max erase count).

To balance the ratio between hot and cold LBA writes, we come up with the third option (**OPTION 3**, we call it *median*), which keeps track of the most recent LBA write requests in a sliding window, and sorts them based on each LBA write request's next access time. If the next access time

of the current LBA is smaller than the median then it is considered as hot. Otherwise it is considered as cold. In this way, approximate half of pages will be classified as hot pages, and the segregation criteria is relative and evolving with the workload. This *median* hot-cold page segregation policy is illustrated in Figure 4.

Endurance with different hot-cold segregation policies)	
	MSR-hm0
Baseline	2.058
Baseline + OPTION 1	2.060
Baseline + OPTION 2	2.062
Baseline + OPTION 3 (median)	2.079
Baseline + Clustering	**2.518**

Table 2: Endurance with different hot-cold segregation policies based on the FTL baseline, on the MSR-hm0 workload.

We run a simple benchmark comparing these three options (and *Clustering*, which we introduce in the next section) on a sample workload trace (more details about the workload are described in Section 4). The result is shown in Table 2. Note that for all these 3 options, the *oracle* is only required to answer "hot" or "cold" when queried by the FTL with an LBA and a timestamp, so the level of *hints* provided by the *oracle* would be quite easy to imitate by a workload-aware FTL algorithm.

Clustering: Locally Optimal Solution for Hot/Cold Page Segregation

Given the entire workload trace, which enables us to foresee upcoming writes in the future, we wonder, what would be the optimal solution for choosing the best physical page for each individual LBA write request? Let's start with a simple thought experiment: given an empty SSD, if we just sequentially write into the SSD, when it first becomes full, it will end up in the scenario depicted in Figure 5. The write amplification will be high because every block has both valid and invalid pages in it when cleaned.

Experiments with **OPTION 2** inspired us to follow the heuristic that we should always place write requests that will become invalid together in the same blocks, and we come up with the following **clustering** policy. Recall that when processing write requests, we are in a *normal period* and both the *oracle* and FTL knows exactly the number of timesteps T until the next *cleaning period*. Therefore, at the start of a *normal period*, the *oracle* can sort the next T LBA write requests based on their next access time, so that these T LBA accesses each have a rank between 1 and T (smaller rank means the block will be accessed sooner and is hotter). When the FTL queries the *oracle* to ask for a *hint* on how hot/cold an LBA is, the *oracle* does not respond with a simple "hot" or "cold" answer; it responds with the rank of the LBA (between 1 and T). Note that at the start of the *normal period*, the FTL has enough free blocks (note that we reserve LT free blocks, which are not involved in this process) to accommodate these T writes. The FTL can sort these F free blocks according to their current erase count, and place the smallest rank LBAs (the hot pages) within the free block with minimum erase count, and the largest rank LBAs (the cold pages) within the free block with maximum erase count. More precisely, if each physical block has B pages, then the rank T LBA will be written to the free block with $\frac{T}{B}$-th lowest erase count.

In this way, any LBA written within a *normal period* will be written to the same block as the other LBAs in the same *normal period* whose next access time is closest to it. We have effectively grouped LBAs together by their next access time, and this will lead to these LBAs becoming invalid at approximately the same time region in the future, which means the block they're in will be quite empty and ideal for cleaning. This **clustering** policy is better than **OPTION 3**, because the FTL is no longer limited to classifying a page as "hot" or "cold" in a binary fashion; it can classify the hotness of pages in a much more fine-grained manner and group

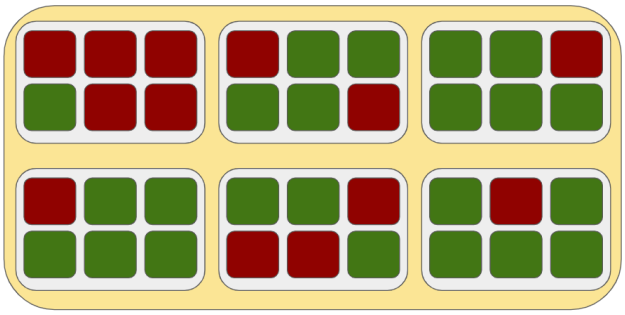

Figure 5: Naive Sequential Writes to SSD. Green squares denote valid pages. Red squares denote invalid pages.

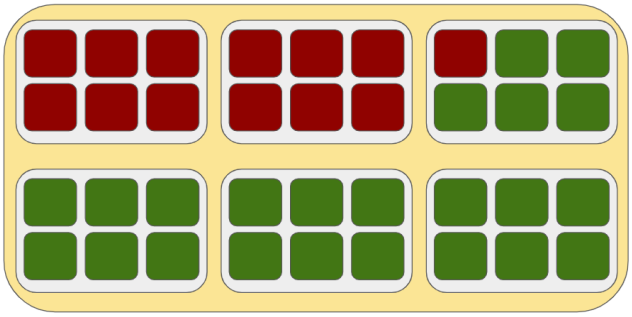

Figure 6: Ideal Scenario when SSD is full. Green squares denote valid pages. Red squares denote invalid pages.

LBAs that may be far apart in the trace but still within the same *normal period* together in the same block.

Now, since **clustering** can only deal with write requests during the *normal period*, we need a separate policy to segregate hot and cold blocks during the *cleaning period*. For this, we use a simple thresholding policy similar to **OPTION 1**. When entering a *cleaning period*, we estimate the length of the next *normal period*, and calculate the median of next access times among LBAs in that *normal period*. Then within the current *cleaning period*, when migrating a page, we compare its next access time to the threshold to determine whether it is hot or cold.

This **clustering** policy also has a nice property, that at any point of time in the future, at most one of these F blocks would have both valid and invalid pages, and all other blocks among these F blocks are either full of valid pages, or full of invalid pages. Clearly, blocks full of invalid pages are most ideal for cleaning. In the best case, our SSD can result in the optimal scenario depicted in Figure 6.

It should be noted, thought, that this **clustering** policy is only locally optimal and not globally optimal. We are only grouping LBAs in the same *normal period*, and not across multiple *normal periods*. Nevertheless, results in Section 4 shows that the performance is already significantly improved by more than 20% using this **clustering** algorithm.

It should also be noted that this **clustering** policy requires a higher level of *hints* from the *oracle* to the FTL, since it is passing the rank of LBAs instead of just a binary “hot" or “cold" answer. If we are to implement a workload-aware algorithm using machine learning, to achieve the performance of this **clustering** policy, the machine learning model must be able to approximate the rank of LBAs, which is a harder task than approximating binary “hot" or “cold".

Greedy Parameter Approximation

FTL operates in an online fashion in that it makes deterministic decisions based on the system state and sometimes the traces it has seen so far at each timestep along its operation. Realistic traces are the embodiment of realistic workloads, and there should exist certain patterns within the traces of those workloads. For example, large sequential writes usually occur during file I/Os and are signified by clustered large writes within the trace. Therefore, with perfect knowledge of an entire trace, FTL algorithms has much more information and could very likely make better decisions (see Figure 7).

No particular policy is suitable for all possible workloads and different optimizations could be made for different access patterns. For our FTL baseline, there exists optimal parameter settings for each workload. However, given that each workload might contain different access patterns, we believe that we can strive for a more optimal policy by making certain parameters of the FTL baseline adaptive, in that it

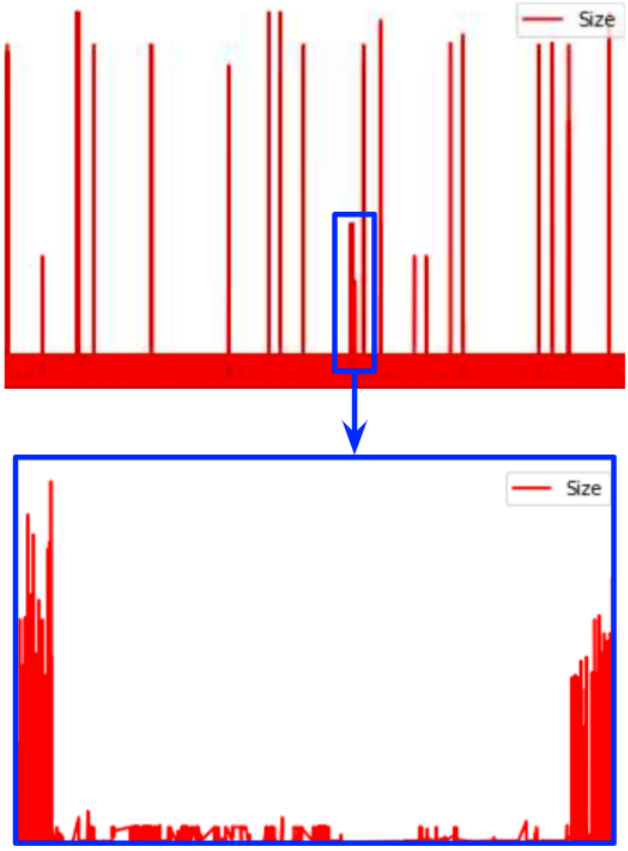

Figure 7: A closer look into MSR-hm0 shows that distinctive patterns are prevalent within a workload.

adjusts its parameter settings upon seeing different patterns within a particular trace. However, it is extremely hard to find the optimal way of making adjustments, especially considering the optimal way may involve shifting parameters throughout the whole trace. As mentioned in the previous section, given perfect knowledge about the trace, there are numerous policies to optimize the segregation of hot and cold pages based on the next access time of LBAs, but the relationship between the other adjustable parameters and the workload is more vague, and doing exhaustive searches to identify the best way of adjusting parameters is too expensive. Therefore, we decide to optimize these other adjustable parameters in a more coarse-grained manner. We assume that patterns exist at fix-sized segments, even though they might last arbitrarily long in realistic cases. More specifically, we break the workload into segments, and find the optimal parameter settings for each segment. Our hope is to pick an appropriate segment size such that more access patterns can be captured by the segmentation.

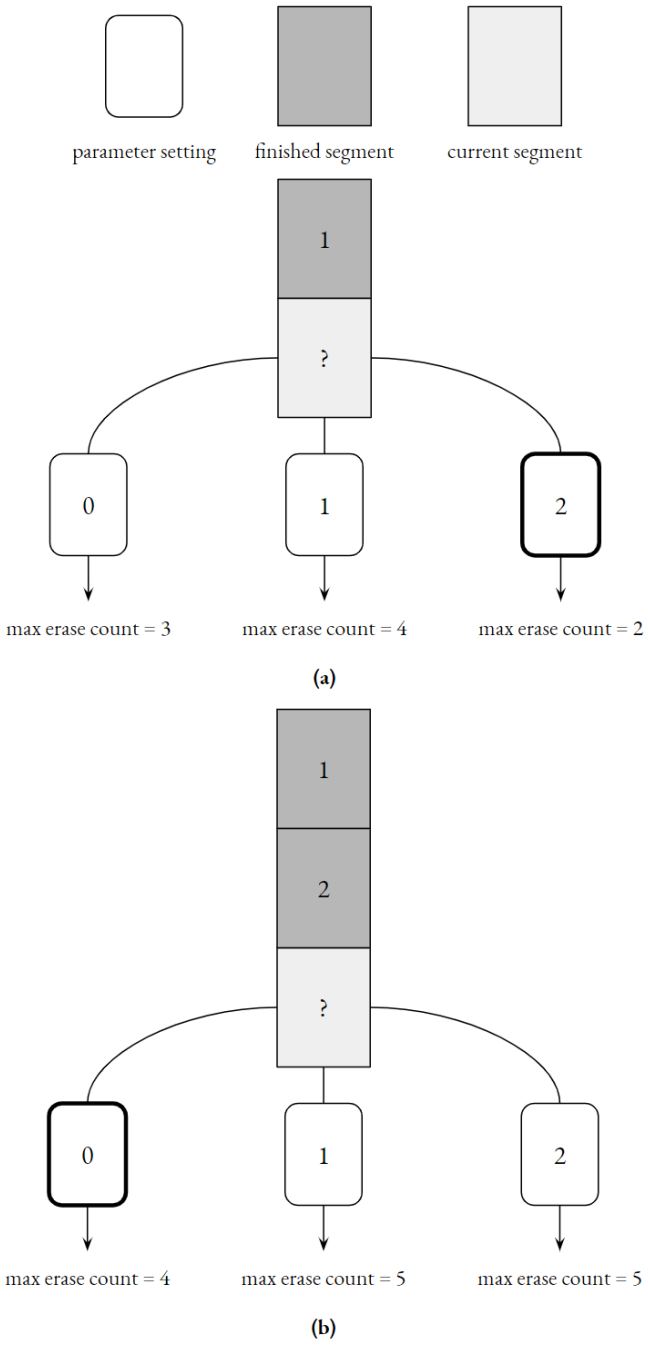

Figure 8: The greedy parameter approximation process. In (a), parameter setting 2 achieves the best performance, so the next iteration will be conditioned on having chosen 2 at the previous segment, as shown in (b).

Therefore, we designed a greedy process for the *oracle* to approximate the optimal parameter settings for the FTL baseline. We also selected a subset of adjustable parameters within the algorithm that we believe are crucial to the overall endurance of the FTL baseline. Specifically, we selected the following parameters:

- LT: low-threshold for cleaning. The garbage collection process will be triggered when the number of free blocks falls below LT.
- HT: high-threshold for cleaning. The garbage collection process will stop when the number of free blocks reaches HT.
- D_{init}: the maximum distance between block erase counts we allow among the blocks.
- D_{int}: the interval at which we decrease D_{init} by 1.

Given the parameters specified above, our greedy oracle emulation process is as follows: we pragmatically select a fixed segment length S. Specifically, we tried different reasonable values for S, and select the segment size that result in a better endurance of the *oracle*. Intuitively, the segment size should be large enough to span several *cleaning periods*, and not too large (in the extreme case that the segment size is equal to the trace size, segmentation is useless). For each non-overlapping segment of length S within the entire trace, we will try certain number of parameter settings and will choose the one that offers the best performance within that specific segment. The parameter setting for the i^{th} segment depends on the current state of the system, and is thus conditioned on the selections made in the preceding segments. The optimal parameter settings serve as *hints* from the *oracle* to the FTL.

Endurance is measured by the maximum number of writes that the algorithm could survive until its blocks reach the erase limit specified by the SSD. Given that we are having segments with smaller sizes, it's unreasonable to use the number of writes that a policy could possibly survive as the measurement of performance at segment-level. At the beginning of the trace, nearly all policies are able to survive through the first several segments since there aren't that many write requests to handle. Different parameter settings in turn get the same endurance score and we will not be able to tell which one is superior. We also decided not to solely use the erase count sum of all blocks as measurement, because the erase count sum does not take into account wear leveling. Therefore, we use as measurement the maximum erase count among all the blocks in the system when the policy under a particular parameter setting finishes its current segment(illustrated in Figure 8). If two parameter settings tie on max erase count, we will pick the one with smaller erase count sum. Our overall goal is to show that in the end, the number of writes the emulated oracle could survive surpasses that of the FTL baseline on its own.

Note that greedy parameter tuning works seamlessly with hotcold segregation, since they affect separate parts of the

FTL algorithm. Greedy parameter tuning changes the hyper-parameters such as LT, HT, and D, while hotcold segregation is only invoked when the FTL accepts write requests or performs page migration.

4 EVALUATION

System and Simulator

We ran our experiments on a 6-core ubuntu server on AWS. For experimental efficiency purposes, we ran the workloads with an SSD of 1600 physical blocks and 409600 physical pages (each block has 256 pages). All LBA accesses in the workloads were taken modulo 409600. After experimenting with a number of segment sizes, we chose 50000 as a good segment size. This means we greedily tune the parameters of the FTL after every 50000 write requests. 50000 is large enough to span several cleaning periods, yet still small enough so that different access patterns can likely fall in different cleaning periods.

We built our FTL and *oracle* implementation on the FlashSim[11] SSD simulator. Even though FlashSim does not model new features of modern SSDs [12] such as multi-queues, it is sufficient for our purpose of measuring write-amplification and wear-leveling.

Memory and Throughput

The memory usage of our FTL is constant across all experiments. We use 16 bytes per physical page to keep track of page-to-page mapping, and another 2 bytes per page and 80 bytes per block for FTL internal use. This averages to around 19 bytes per physical page, which is reasonable memory consumption, given that physical pages are typically 512 bytes in size.

The throughput of our FTL is mostly consistent across all policies. Both the baseline FTL and FTL with *oracle* are able to process around 60000 write requests per second, which matches the industry standard. The only exception is when we use some hot cold policies such as *median*, that requires scanning multiple requests in a window. These policies can bring the throughput down to around 40000 write requests per second.

Workloads

In our experiments we use real world block I/O traces from UMass Trace Repository and from MSR Cambridge. We run two OLTP financial application I/O traces and three popular search engine I/O traces from UMass Trace Repository. We also run two 1-week block I/O traces of enterprise servers at Microsoft from MSR Cambridge. All of these traces are a sequence of operations, where each operation includes a starting LBA, number of bytes accessed, and read/write. We ignore all the read requests from these traces and only feed their write requests to our FTL, because read requests do not influence garbage collection, wear leveling, or the lifetime of the SSD.

We conducted our experiments mainly on the two 1-week block I/O traces of enterprise servers from MSR, called MSR-hm0 and MSR-hm1. Since our concern is the warmed-up case, we also conducted experiments on the 5 workloads from UMass and measured their write amplification and endurance on various policies with a warmup phase. We show the average of their endurance and write amplification in Table 4 and Table 6.

We also wrote 5 of our own tests that use mixes of sequential and random accesses to test the correctness and performance of our FTL. We wrote one test with purely random workload, one with purely sequential workload, one with random workload combined with TRIMs, one with mix of sequential and random workload under low SSD usage, and one with mix of sequential and random workload under high SSD usage.

TRIMs

Unfortunately none of the workloads from UMass Trace Repository or MSR Cambridge contain TRIM requests, which are hints from the operating system that a page can be safely deleted from the SSD. To simulate real world scenarios, we implemented a TRIM manager that optionally adds a TRIM request after a write request, if the LBA written is never read before its next update.

Note that a workload-aware algorithm without *oracle* is not allowed to add TRIM requests, because it does not know if a LBA will be read soon in the future, and an incorrect TRIM request will cause future reads to be unserviceable. Therefore, adding TRIM requests is not a feature of our *oracle*. As we will discuss below, we add TRIM requests in some experiments to mimic real world scenarios.

Warmup

SSD performance evaluation standards explicitly clarify that the SSD performance should be reported in the steady state, where the SSD is already filled with application data and warmed up. Papers such as MQSim[12] have discussed numerous reasons why it is important to measure SSD performance in the steady state: for example, garbage collection activities are invoked only after the warmup phase. Therefore, we do experiments both with a warmup phase and without a warmup phase. We examine warmup phases with different ratios: we can warmup by filling as much as 80% of pages on the SSD, or as low as 20%.

We discovered, that with a warmup phase, the SSD will be quite full, and TRIMs will be more effective. Therefore, we measure the effect of TRIMs with warmup phases of different ratios, on the FTL baseline and FTL with *oracle*. As seen in

Table 5, where we run the MSR-hm0 workload under different warmup ratios, adding TRIMs via our TRIM manager significantly increases endurance, especially under the 80% warmup ratio case where Endurance increased by 100%. We conjecture that when the SSD is close to full, the FTL has little maneuver space and must incur a large number of page migrations and high WA upon cleaning, so freeing pages via TRIM is particular useful in this case. We see that with TRIMs added, the FTL with *oracle* (using greedy parameter tuning and clustering hot-cold page segregation) still outperforms the baseline in all the cases by 20% to 25%.

Experiments without a warmup phase

We first perform experiments to measure the Endurance for our various *oracle* policies on the MSR-hm0 and MSR-hm1 workloads without a warmup phase. Our results are shown in Table 3.

As seen from the results, greedy parameter tuning only provides marginal improvement in Endurance. This is probably because our greedy tuning is coarse and has segment-granularity. It could also be due to the nature of FTL baseline, that tuning parameters such as LT, HT, and D is not influential. Combining greedy parameter tuning with the *median* policy of hot-cold page segregation provides a 1% improvement in Endurance, whereas the *clustering* policy of hot-cold page segregation provides more than 20% of improvement. This is probably because even though the *median* policy also knows the hotness of pages, it is limited to telling the FTL to write pages either into CDB or CCB, meaning that it can only classify a page as hot or cold. On the other hand, the *clustering* policy can tell the FTL to write pages to any of the free blocks, which means it can put the hotter pages in free blocks with smaller erase counts and colder pages in free blocks with larger erase counts, and classify the hotness rating of a block in a much more fine-grained manner. Our results show that correctly distinguishing the hotness/coldness rating of page can greatly improve the page-allocation and page-migration policy of FTLs and avoid many unnecessary page migrations.

Experiments with a warmup phase

We then perform the same set of experiments with a warmup phase that fills up 80% of pages on the SSD. We additionally run the 5 workloads from UMass and take the average of their Endurance to display in Table 4. As expected, the Endurance is lower with warmup, since the FTL has much less maneuver space and would suffer from high write amplification when cleaning.

The results for *oracle* are mostly consistent with the no-warmup case. Greedy parameter tuning provides marginal improvement, *median* policy provides around 1% improvement, and *clustering* policy provides around 25% improvement.

Write Amplification

As a sanity check, we also measure the Write Amplification of the various policies with a warmup phase that fills up 80% of the SSD for different workloads. Table 6 shows our results. We can see that the results are mostly consistent with those from Table 4: Endurance is inversely correlated with Write Amplification, as expected. We conjecture that a good FTL policy will incur less page migrations upon cleaning, which would lead to both small Write Amplification and high Endurance.

Endurance without a warmup phase (in millions)			
	MSR-hm0	MSR-hm1	UMass Avg
Baseline	2.06	3.28	2.76
Baseline + Greedy	2.06	3.29	2.77
Baseline + Greedy + Median	2.08	3.28	2.77
Baseline + Clustering	2.52	4.01	2.91
Baseline + Greedy + Clustering	**2.53**	**4.03**	**2.91**

Table 3: Endurance of algorithms using an empty SSD.

Endurance with a warmup phase (in millions)			
	MSR-hm0	MSR-hm1	UMass avg
Baseline	1.56	1.26	1.60
Baseline + Greedy	1.55	1.26	1.61
Baseline + Greedy + Median	1.56	1.26	1.64
Baseline + Clustering	1.87	1.54	1.92
Baseline + Greedy + Clustering	**1.88**	**1.55**	**1.94**

Table 4: Endurance of algorithms with warmup phase that fills up 80% of SSD.

Endurance with a warmup phase and TRIMs (in millions)			
	with TRIMs	No TRIMs	Baseline with TRIMs
warmup 0% of pages	4.41	2.53	3.61
warmup 20% of pages	4.37	2.46	3.57
warmup 40% of pages	4.25	2.33	3.50
warmup 60% of pages	4.03	2.16	3.33
warmup 80% of pages	3.65	1.88	3.04

Table 5: Endurance of the Baseline + Greedy + Clustering policy, and baseline policy on the workload MSR-hm0, with TRIMs added, under different warmup ratios.

Write Amplification with a warmup phase			
	MSR-hm0	MSR-hm1	UMass avg
Baseline	2.89	1.21	2.93
Baseline+Greedy	2.89	1.22	2.93
Baseline + Greedy + Median	2.88	1.20	2.91
Baseline + Clustering	2.69	1.12	2.77
Baseline + Greedy + Clustering	**2.60**	**1.08**	**2.73**

Table 6: Write amplification of algorithms with a warmup phase that fills up 80% of SSD.

5 FUTURE WORK

There are three promising future directions to extend this project: an even better *oracle*, emulation of oracle using machine learning techniques, and FTL-oriented OS optimization.

Even better oracle

We explored many policies for the *oracle*, but we conjecture that we are still far from finding the optimal page allocation and placement policy for every workload and there is still much room for improvement. For example, our identification parameter tuning based on segments is quite coarse, and our best hot-cold page segregation policy, clustering, is locally optimal but not globally optimal - we place the pages with similar next access times within the same *normal period* in the same block, but we do not look across multiple *normal periods*.

Emulation of oracle using machine learning techniques

Below we will provide some rough ideas on how to actually devise a superior workload-aware algorithm that uses machine learning instead of the *oracle* to provide *hints* for the FTL. We have not actually implemented the ML algorithm, but we offer some preliminary ideas inspired by the way our *oracle* is implemented.

The oracle will be turned into a function for ML models to approximate. In practice, the training can be done offline in a data center that receives a lot of LBA accesses. Assuming that past LBA accesses have similar access patterns to future access patterns in the same data center, we can deploy the *oracle* algorithm on past accesses to obtain our dataset for training ML algorithms.

How the dataset looks like depends on how the ML model is structured. Assuming that we always split LBA accesses into segments, one method is to make the ML model read in the first K accesses of a segment, and output the parameter adjustment as similar to the decision that the oracle would have made on that segment as possible. In this case, the input will be the first K accesses within every segment of the traces the datacenter has already obtained, and the corresponding label would be the best parameter settings that an oracle would have chosen for that segment. Alternatively, the ML model could finish reading an entire segment and output the parameter setting for the upcoming segment. In this case, the inputs to the model would be one whole segment and the output would be the *oracle* parameter setting for the segment immediately following it.

Since the ML model reads in sequential data, sequence-to-sequence models capable of capturing temporal dependency might be good candidates for modeling this approximation process.

Then, while the training can happen in the background, the ML will perform inference on new accesses. The inference overhead can be mitigated with software optimizations such as batching and hardware optimizations such as the incorporation of GPUs with NAND flash memory in the future.

FTL-oriented OS optimization

In many cases, the Operating System (OS) has some partial knowledge about the workload it is passing on to the FTL. For example, the OS might know that a specific application invokes a certain workload pattern, or that a series of file system operations will repeatedly access the same region of LBAs. In these scenarios, the OS is essentially a weaker version of the *oracle*, and it will be interesting to explore ways in which the OS can use its knowledge to provide useful *hints* to the FTL to improve FTL endurance.

6 CONCLUSION

We implemented an *oracle* with complete knowledge of the workload to provide *hints* to the FTL. We compared the performance of our FTL baseline (which is based upon state-of-the-art FTL algorithms) to FTL with *oracle*, showing that FTL with *oracle* has 20% to 25% increase in Endurance and reasonable memory and throughput. This improvement in SSD lifetime serves as a proof-of-concept that a workload-aware FTL algorithm superior to current state-of-the-art algorithms is feasible, possibly using machine techniques to generate *hints* based on past LBA accesses instead of using an *oracle*.

REFERENCES

[1] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[2] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[3] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer

using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18–es, July 2007.

[4] M. Murugan and D. H. C. Du. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2011.

[5] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. pages 1126–1130, 01 2007.

[6] Dongzhe Ma, Jianhua Feng, and Guoliang Li. Lazyftl: A page-level flash translation layer optimized for nand flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1–12, New York, NY, USA, 2011. Association for Computing Machinery.

[7] L. Yao, D. Liu, K. Zhong, L. Long, and Z. Shao. Tlc-ftl: Workload-aware flash translation layer for tlc/slc dual-mode flash memory in embedded systems. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 831–834, Aug 2015.

[8] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, page 1126–1130, New York, NY, USA, 2007. Association for Computing Machinery.

[9] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design. pages 212–217, 07 2007.

[10] Dongchul Park and David Du. Hot data identification for flash-based storage systems using multiple bloom filters. pages 1 – 11, 06 2011.

[11] Kim Youngjae, Brendan Tauras, Aayush Gupta, Dragos Mihai, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. *Advances in System Simulation, International Conference on*, 0, 04 2009.

[12] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. Mqsim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, Oakland, CA, February 2018. USENIX Association.

A APPENDIX

Our goals from the interim project report

Our goals for this project are:

- **75% goal:** Adapt our FTL baseline to work with realistic workloads. Identify key parameters that affect performance for different workloads.
- **100% goal:** Either empirically show that a workload-aware *oracle* will boost the performance of the FTL baseline, or show that such an *oracle* does not bring significant performance improvement. As proof-of-concept, the existence of such an *oracle* shows that ML policies approximating such an *oracle* can be effective.
- **125% goal:** Provide clear experimental setup for training ML models that emulate the FTL baseline with oracle. Experiment with various neural networks and document their performance. Discover a suitable neural network design that best emulate the oracle and outperforms the original FTL baseline on its own. Optionally cut down the overhead of running an ML model,