The Devil Is in the Details: Implementing Flash Page Reuse with WOM Codes

Fabio Margaglia, ***Johannes Gutenberg—Universität Mainz;*** **Gala Yadgar and Eitan Yaakobi,** ***Technion—Israel Institute of Technology;*** **Yue Li,** ***California Institute of Technology;*** **Assaf Schuster,** ***Technion—Israel Institute of Technology;*** **André Brinkmann,** ***Johannes Gutenberg—Universität Mainz***

https://www.usenix.org/conference/fast16/technical-sessions/presentation/margaglia

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

The Devil is in the Details: Implementing Flash Page Reuse with WOM Codes

Fabio Margaglia[†], Gala Yadgar[*], Eitan Yaakobi[*], Yue Li[§], Assaf Schuster[*] and André Brinkmann[†]
[†]*Johannes Gutenberg-Universität Mainz*
[*]*Computer Science Department, Technion*
[§]*California Institute of Technology*

Abstract

Flash memory is prevalent in modern servers and devices. Coupled with the scaling down of flash technology, the popularity of flash memory motivates the search for methods to increase flash reliability and lifetime. Erasures are the dominant cause of flash cell wear, but reducing them is challenging because flash is a *write-once* medium— memory cells must be erased prior to writing.

An approach that has recently received considerable attention relies on *write-once memory (WOM)* codes, designed to accommodate additional writes on write-once media. However, the techniques proposed for reusing flash pages with WOM codes are limited in their scope. Many focus on the coding theory alone, while others suggest FTL designs that are application specific, or not applicable due to their complexity, overheads, or specific constraints of MLC flash.

This work is the first that addresses all aspects of page reuse within an end-to-end implementation of a general-purpose FTL on MLC flash. We use our hardware implementation to directly measure the short and long-term effects of page reuse on SSD durability, I/O performance and energy consumption, and show that FTL design must explicitly take them into account.

1 Introduction

Flash memories have special characteristics that make them especially useful for solid-state drives (SSD). Their short read and write latencies and increasing throughput provide a great performance improvement compared to traditional hard disk based drives. However, once a flash cell is written upon, changing its value from 1 to 0, it must be erased before it can be rewritten. In addition to the latency they incur, these erasures wear the cells, degrading their reliability. Thus, flash cells have a limited lifetime, measured as the number of erasures a block can endure.

Multi-level flash cells (MLC), which support four voltage levels, increase available capacity but have especially short lifetimes, as low as several thousands of erasures. Many methods for reducing block erasures have been suggested for incorporation in the flash translation layer (FTL), the SSD management firmware. These include minimizing user and internal write traffic [14, 19, 20, 28, 37, 38, 42, 46, 55] and distributing erasure costs evenly across the drive's blocks [7, 22, 25, 27].

A promising technique for reducing block erasures is to use write-once memory (WOM) codes. WOM codes alter the logical data before it is physically written, thus allowing the reuse of cells for multiple writes. They ensure that, on every consecutive write, ones may be overwritten with zeros, but not vice versa. Reusing flash cells with this technique might make it possible to increase the amount of data written to the block before it must be erased.

Flash page reuse is appealing because it is orthogonal to other FTL optimizations. Indeed, the design of WOM codes and systems that use them has received much attention in recent years. While the coding theory community focuses on optimizing these codes to reduce their redundancy and complexity [9, 10, 13, 17, 44, 49], the storage community focuses on SSD designs that can offset these overheads and be applied to real systems [24, 36, 53].

However, the application of WOM codes to state-of-the-art flash chips is not straightforward. MLC chips impose additional constraints on modifying their voltage levels. Previous studies that examined page reuse on real hardware identified some limitations on reprogramming MLC flash, and thus resort to page reuse only on SLC flash [24], outside an SSD framework [18], or within a limited special-purpose FTL [31].

Thus, previous SSD designs that utilize WOM codes have not been implemented on real platforms, and their benefits were analyzed by simulation alone, raising the concern that they could not be achieved in real world storage systems. In particular, hardware aspects such as possible increase in cell wear and energy consumption due to the additional writes and higher resulting voltage levels have not been examined before, but may have dramatic implications on the applicability of this approach.

In this study, we present the first end-to-end evaluation and analysis of flash page reuse with WOM codes. The first part of our analysis consists of a low-level evaluation of four state-of-the-art MLC flash chips. We examine the possibility of several reprogramming schemes for MLC flash and their short and long-term effects on the chip's

durability, as well as the difference in energy consumption compared to that of traditional use.

The second part of our analysis consists of a system-level FTL evaluation on the OpenSSD board [4]. Our FTL design takes into account the limitations identified in the low-level analysis and could thus be implemented and evaluated on real hardware. We measure erasures and I/O response time and compare them to those observed in previous studies.

The discrepancy between our results and previous ones emphasizes why understanding low-level constraints on page reuse is crucial for high-level designs and their objectives. We present the lessons learned from our analysis in the form of guidelines to be taken into account in future designs, implementations, and optimizations.

The rest of this paper is organized as follows. Section 2 describes the basic concepts that determine to what extent it is possible to benefit from flash page reuse. We identify the limitations on page reuse in MLC flash in Section 3, with the implications on FTL design in Section 4. We describe our experimental setup and FTL implementation in Section 5, and present our evaluation in Section 6. We survey related work in Section 7, and conclude in Section 8.

2 Preliminaries

In this section, we introduce the basic concepts that determine the potential benefit from flash page reuse: WOM codes, MLC flash, and SSD design.

2.1 Write-Once Memory Codes

Write-once memory (WOM) codes were first introduced in 1982 by Rivest and Shamir, for recording information multiple times on a write-once storage medium [40]. They give a simple WOM code example, presented in Table 1. This code enables the recording of two bits of information in three cells twice, ensuring that in both writes the cells change their value only from 1 to 0. For example, if the first message to be stored is 00, then 110 is written, programming only the last cell. If the second message is 10, then 010 is written, programming the first cell as well. Note that without special encoding, 00 cannot be overwritten by 10 without prior erasure. If the first and second messages are identical, then the cells do not change their value between the first and second writes. Thus, before performing a second write, the cell values must be *read* in order to determine the correct encoding.

Data bits	1st write	2nd write
11	111	000
01	011	100
10	101	010
00	110	001

Table 1: WOM code example

WOM code instances, or *constructions*, differ in the number of achievable writes and in the manner in which each successive write is encoded. The applicability of a WOM code construction to storage depends on three characteristics: (a) the *capacity overhead* —the number of extra cells required to encode the original message, (b) the encoding and decoding *efficiency*, and (c) the *success rate*—the probability of producing an encoded output that can be used for overwriting the chosen cells. Any two of these characteristics can be optimized at the cost of compromising the third.

Consider, for example, the code depicted in Table 1, where encoding and decoding are done by a simple table lookup, and therefore have complexity $O(1)$ and a success rate of 100%. However, this code incurs a capacity overhead of 50% on each write. This means that (1) only $\frac{2}{3}$ of the overall physical capacity can be utilized for logical data, and (2) every read and write must access 50% more cells than what is required by the logical data size.

The theoretical lower bound on capacity overhead for two writes is 29% [40]. Codes that incur this minimal overhead (*capacity achieving*) are not suitable for real systems. They either have exponential and thus inapplicable complexity, or complexity of $n \log n$ (where n is the number of encoded bits) but a failure rate that approaches 1 [10, 56]. Thus, early proposals for rewriting flash pages using WOM codes that were based on capacity achieving codes were impractical. In addition, they required partially programming additional pages on each write, modifying the physical page size [8, 18, 23, 30, 36, 50], or compressing the logical data prior to encoding [24].

Two recently suggested WOM code families, Polar [9, 10] and LDPC [56], have the same complexities as the error correction codes they are derived from. For these complexities, different constructions incur different capacity overheads, and the failure rate decreases as the capacity overhead increases. Of particular interest are constructions in which the overhead of the first write is 0, i.e., one logical page is written on one physical page. The data encoded for the second write requires two full physical pages for one logical page. Such a construction is used in the design of ReusableSSD [53], where the second write is performed by programming pages containing invalid data on two different blocks in parallel.

2.2 Multi-Level Cell (MLC) Flash

A flash chip is built from floating-gate cells whose state depends on the number of electrons they retain. Writing is done by *programming* the cell, increasing the *threshold voltage (V_{th})* required to activate it. Cells are organized in blocks, which are the unit of erasure. Blocks are further divided into pages, which are the read and program units.

Single-level cells (SLC) support two voltage levels, mapped to either 1 (in the initial state) or 0. Thus, SLC flash is a classic write-once memory, where pages can be reused by programming some of their 1's to 0's. We refer to programming without prior erasure as *reprogramming*. Multi-level cells (MLC) support four voltage levels, mapped to 11 (in the initial state), 01, 00 or 10. This mapping, in which a single bit is flipped between successive

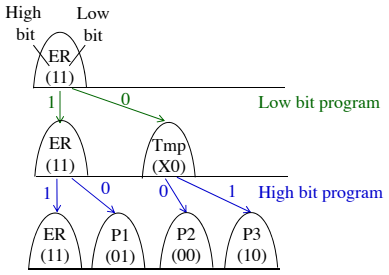


Figure 1: Normal programming order and states of MLC flash. ER is the initial (erased) state.

	A16	A27	B16	B29	C35
Feature size	16nm	27nm	16nm	29nm	35nm
Page size	16KB	8KB	16KB	4KB	8KB
Pages/block	256	256	512	256	128
Spare area (%)	10.15	7.81	11.42	5.47	3.12
Lifetime (T)	3K	5K	10K	10K	NA

Table 2: Evaluated flash chip characteristics. A, B and C represent different manufacturers. The C35 chip was examined in a previous study, and is included here for completeness.

states, minimizes bit errors if the cell's voltage level is disturbed. The least and most significant bits represented by the voltage levels of a multi-level cell are mapped to two separate pages, the *low page* and *high page*, respectively. These pages can be programmed and read independently. However, programming must be done in a certain order to ensure that all possible bit combinations can be read correctly. Triple-level cells (TLC) support eight voltage levels, and can thus store three bits. Their mapping schemes and programming constraints are similar to those of MLC flash. We focus our discussion on MLC flash, which is the most common technology in SSDs today.

Figure 1 depicts a normal programming order of the low and high bits in a multi-level cell. The cell's initial state is the erased (ER) state corresponding to 11. The low bit is programmed first: programming 1 leaves the cell in the erased state, while programming 0 raises its level and moves it to a temporary state. Programming the high bit changes the cell's state according to the state it was in after the low bit was programmed, as shown in the bottom part of the figure.[1] We discuss the implications of this mapping scheme on page reuse in the following section.

Bit errors occur when the state of the cell changes unintentionally, causing a bit value to flip. The reliability of a flash block is measured by its *bit error rate (BER)*—the average number of bit errors per page. The high voltage applied to flash cells during repeated program and erase operations gradually degrades their ability to retain the applied voltage level. This causes the BER to increase as the block approaches the end of its lifetime, which is measured in program/erase (P/E) cycles.

Bit errors in MLC flash are due mainly to *retention errors* and *program disturbance* [11]. Retention errors occur when the cell's voltage level gradually decreases below the boundaries of the state it was programmed to. Program disturbance occurs when a cell's state is altered during programming of cells in a neighboring page. In the following section, we discuss how program disturbance limits MLC page reuse, and evaluate the effects of reusing a block's pages on its BER.

Error correction codes (ECC) are used to correct some of the errors described above. The redundant bits of the ECC are stored in each page's *spare area*. The number of bit errors an ECC can correct increases with the number of redundant bits, chosen according to the expected BER at the end of a block's lifetime [56].

Write requests cannot update the data in the same place it is stored, because the pages must first be erased. Thus, writes are performed *out-of-place*: the previous data location is marked as invalid, and the data is written again on a clean page. The *flash translation layer (FTL)* is the SSD firmware component responsible for mapping logical addresses to physical pages. We discuss relevant components of the FTL further in Section 4.

3 Flash Reliability

Flash chips do not support reprogramming via their standard interfaces. Thus, the implications of reprogramming on the cells' state transitions and durability cannot be derived from standard documentation, and require experimentation with specialized hardware. We performed a series of experiments with several state-of-the-art flash chips to evaluate the limitations on reprogramming MLC flash pages and the implications of reprogramming on the chip's lifetime, reliability, and energy consumption.

3.1 Flash Evaluation Setup

We used four NAND flash chips from two manufacturers and various feature sizes, detailed in Table 2. We also include in our discussion the observations from a previous study on a chip from a third manufacturer [31]. Thus, our analysis covers three out of four existing flash vendors.

Chip datasheets include the expected lifetime of the chip, which is usually the maximal number of P/E cycles that can be performed before the average BER reaches 10^{-3}. However, cycling the chips in a lab setup usually wears the cells faster than normal operation because they program and erase the same block continuously. Thus, the threshold BER is reached after fewer P/E cycles than expected. In our evaluation, we consider the lifetime (T) of the chips as the minimum of the expected number of cycles, and the number required to reach a BER of 10^{-3}.

Our experiments were conducted using the SigNASII commercial NAND flash tester [6]. The tester allows software control of the physically programmed flash blocks and pages within them. By disabling the ECC hardware we were able to examine the state of each cell, and to count the bit errors in each page.

[1] Partially programming the high bit in the temporary state is designed to reduce program disturbance.

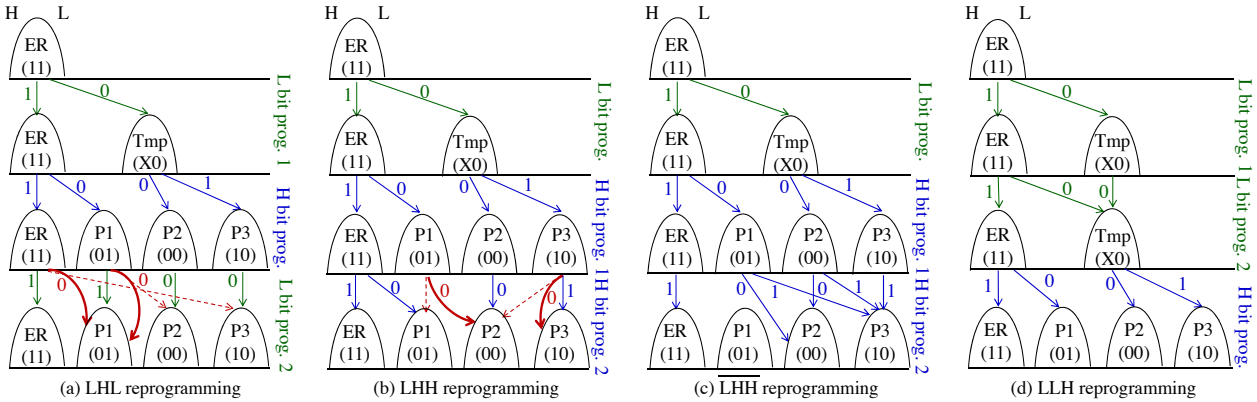


Figure 2: State transitions in the three reprogramming schemes. A thin arrow represents an attempted transition. A dashed arrow represents a failed transition, with a bold arrow representing the erroneous transition that takes place instead. Only LLH reprogramming achieves all the required transitions for page reuse without program disturbance.

Some manufacturers employ *scrambling* within their chip, where a random vector is added to the logical data before it is programmed. Scrambling achieves uniform distribution of the flash cell levels, thus reducing various disturbance effects. In order to control the exact data that is programmed on each page, we bypass the scrambling mechanism on the chips that employ it.

Our evaluation excludes retention errors, which occur when considerable time passes between programming and reading a page. Reprogramming might increase the probability of retention errors because it increases the cell's V_{th}. However, since it is intended primarily for hot data, we believe it will not cause additional retention errors.

3.2 Limitations on reprogramming

Flash cell reprogramming is strictly limited by the constraint that V_{th} can only increase, unless the block is erased. At the same time, WOM encoding ensures that reprogramming only attempts to change the value of each bit from 1 to 0. However, additional limitations are imposed by the scheme used for mapping voltage levels to bit values, and by the need to avoid additional program disturbance. Thus, page reuse must follow a *reprogramming scheme* which ensures that all reprogrammed cells reach their desired state. We use our evaluation setup to examine which state transitions are possible in practice. We first consider three reprogramming schemes in which a block has been fully programmed, and show why they are impractical. We then validate the applicability of reprogramming when only the low pages of the block have been programmed before.

Let us assume that the entire block's pages have been programmed before they are reused. Thus, the states of the cells are as depicted in the bottom row of Figure 1. In the *low-high-low (LHL)* reprogramming scheme, depicted in Figure 2(a), we attempt to program the low bit from this state. The thin arrows depict possible desired transitions in this scheme. Two such transitions are impossible, resulting in an undesired state (depicted by the bold arrow). In the *low-high-high (LHH)* reprogramming scheme, depicted in Figure 2(b), the high page is reprogrammed in a fully used block. Here, too, two state transitions fail.

A possible reason for the failed transitions in the LHL scheme is that the voltage applied by the command to program the low bit is not high enough to raise V_{th} from $P1$ to $P2$ and from ER to $P3$.[2] The transition from $P3$ to $P2$ in the LHH scheme is impossible, because it entails decreasing V_{th}. Another problem in the LHH scheme occurs in state $P1$ when we attempt to leave the already programmed high bit untouched. Due to an unknown disturbance, the cell transitions unintentionally to $P2$, corrupting the data on the corresponding low page.

Three of these problematic transitions can probably be made possible with proper manufacturer support—the transition from $P3$ to $P2$ in the LHH scheme would be possible with a different mapping of voltage levels to states, and the two transitions in the LHL scheme could succeed if a higher voltage was applied during reprogramming. While recent technology trends, such as one-shot programming and 3D V-NAND [21], eliminate some constraints on page programming, applying such architectural changes to existing MLC flash might amplify program disturbance and increase the BER. Thus, they require careful investigation and optimization.

An alternative to modifying the state mapping is modifying the WOM encoding, so that the requirement that 1's are only overwritten by 0's is replaced by the requirement that 0's are only overwritten by 1's. Figure 2(c) shows the resulting *$\overline{\text{low-high-high}}$ ($\overline{\text{LHH}}$)* reprogramming scheme. Its first drawback is that it corrupts the low pages, so a high page can be reused only if the data on the low page is either invalid, or copied elsewhere prior to reprogramming. Such reprogramming also corrupted the high pages adjacent to the reprogrammed one. Thus, this scheme allows safe reprogramming of only one out of two high pages.

[2] The transition from ER to $P3$ actually succeeded in the older, C35 chip [31]. All other problematic transitions discussed in this section failed in all the chips in Table 2.

The benefits from such a scheme are marginal, as these pages must also store the redundancy of the encoded data.

Interestingly, reprogramming the high bits in chips from manufacturer A returned an error code and did not change their state, regardless of the attempted transition. A possible explanation is that this manufacturer might block reprogramming of the high bit by some internal mechanism to prevent the corruption described above.

The problems with the LHL and LHH schemes motivated the introduction of the *low-low-high (LLH)* reprogramming scheme by Margaglia et al. [31]. Blocks in this scheme are programmed in two rounds. In the first round only the low pages are programmed. The second round takes place after most of the low pages have been invalidated. All the pages in the block are programmed in order, i.e., a low page is reprogrammed and then the corresponding high page is programmed for the first time, before moving on to the next pair of pages.

We validated the applicability of the LLH scheme on the chips of manufacturers A and B. Figure 2(d) depicts the corresponding state transitions of the cells. Since both programming and reprogramming of the low bit leave the cell in either the erased or temporary state, there are no limitations on the programming of the high page in the bottom row. This scheme works well in all the chips we examined. However, it has the obvious drawback of leaving half of the block's capacity unused in the first round. This leads to the first lesson from our low-level evaluation.

Lesson 1: *Page reuse in MLC flash is possible, but can utilize only half of the pages and only if some of its capacity has been reserved in advance. FTL designs must consider the implications of this reservation.*

3.3 Average V_{th} and BER

In analyzing the effects of reprogramming on a chip's durability, we distinguish between *short-term* effects on the BER due to modifications in the current P/E cycle, and *long-term* wear on the cell, which might increase the probability of errors in future cycles. With this distinction, we wish to identify a *safe* portion of the chip's lifetime, during which the resulting BER as well as the long term wear are kept at an acceptable level.

Reprogramming increases the probability that a cell's value is 0. Thus, the average V_{th} of reused pages is higher than that of pages that have only been programmed once. A higher V_{th} increases the probability of a bit error. The short-term effects of increased V_{th} include increased program disturbance and retention errors, which are a direct result of the current V_{th} of the cell and its neighboring cells. The long-term wear is due to the higher voltage applied during programming and erasure.

Our first set of experiments evaluated the short-term effects of increased V_{th} on a block's BER. In each chip, we performed T regular P/E cycles writing random data on one block, where T is the lifetime of the chip as detailed in Table 2. We repeated this processwith different distributions of 1 and 0. $P_{0.5}$, in which the probability of a bit to be 0 is 0.5, is our baseline. With P_{LLH} the probability of 0 was 0.75 and 0.5 in the low and high page, respectively. This corresponds to the expected probabilities after LLH reprogramming. We read the block's content and recorded the BER after every P/E cycle. We repeated each experiment on six blocks, and calculated the average.

Num. of P_{LLH} cycles	A16	A27	B16	B29
T (= entire lifetime)	32%	29%	20%	30.5%
$0.6 \times T$	8%	9%	8%	9%
$0.4 \times T$	6%	6.5%	6%	6.5%
$0.2 \times T$	2%	3%	3%	3.5%

Table 3: Expected reduction in lifetime due to increased V_{th}.

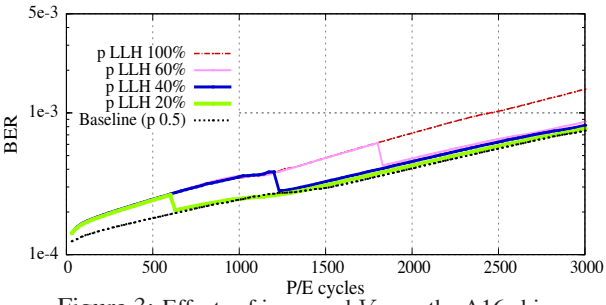


Figure 3: Effects of increased V_{th} on the A16 chip.

The implication of an increase in BER depends on whether it remains within the error correction capabilities of the ECC. A small increase in BER at the end of a block's lifetime might deem it unusable, while a large increase in a 'young' block has little practical effect. For a chip with lifetime T, let T' be the number of cycles required to reach a BER of 10^{-3} in this experiment. Then $T - T'$ is the *lifetime reduction* caused by increasing V_{th}. Our results, summarized in Table 3, were consistent in all the chips we examined.[3] Programming with P_{LLH}, which corresponds to a higher average V_{th}, shortened the chips' lifetime considerably, by 20–32%.

In the next set of experiments, we evaluated the long-term effects of V_{th}. Each experiment had two parts: we programmed the block with P_{LLH} in the first part, for a portion of its lifetime, and with $P_{0.5}$ in the second part, which consists of the remaining cycles. Thus, the BER in the second part represents the long-term effect of the biased programming in the first part. We varied the length of the first part between 20%, 40% and 60% of the block's lifetime. Figure 3 shows the BER of blocks in the A16 chip (the graphs for the different chips were similar), with the lifetime reduction of the rest of the chips in Table 3.

Our results show that the long-term effect of increasing V_{th} is modest, though nonnegligible—increasing V_{th} early in the block's lifetime shortened it by as much as 3.5%, 6.5% and 9%, with increased V_{th} during 20%, 40% and 60% of the block's lifetime, respectively.

[3]The complete set of graphs for all the experiments described in this section is available in our technical report [54].

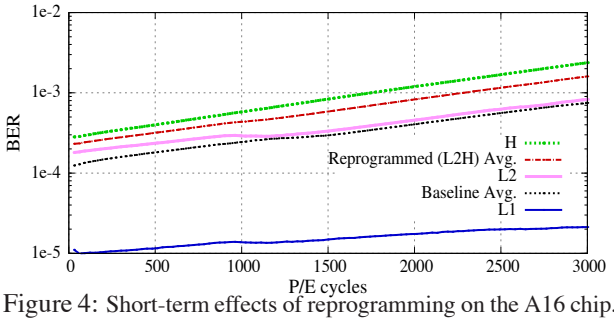


Figure 4: Short-term effects of reprogramming on the A16 chip.

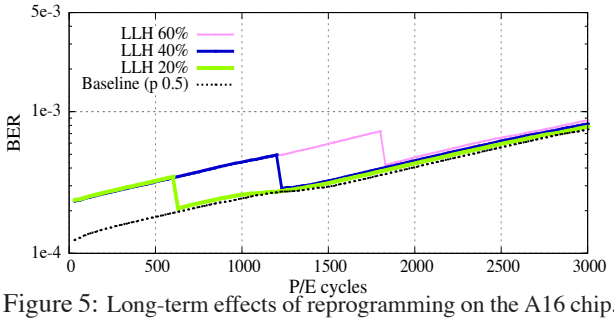


Figure 5: Long-term effects of reprogramming on the A16 chip.

Num. of LLH cycles	A16	A27	B16	B29
T (= entire lifetime)	38%	59.5%	99%	31%
$0.6 \times T$	8.5%	8%	7%	8.5%
$0.4 \times T$	5.2%	6%	5%	5.5%
$0.2 \times T$	1%	2.5%	3%	3%

Table 4: Expected reduction in lifetime due to reprogramming.

Operation	Baseline (μJ)	LLH (μJ)
Erase	192.79	186.49
Read (L)	50.37	50.37
Read (H)	51.25	51.25
Program (L_1)	68.18	68.55
Reprogram (L_2)	NA	63.04
Program (H)	195.65	180.85
Average logical read	50.81	60.79
Average logical write	132.64	145.71

Table 5: Energy consumed by flash operations on chip A16.

3.4 Reprogramming and BER

In the third set of experiments, we measured the effects of reprogramming by performing T LLH reprogramming cycles on blocks in each chip. Figure 4 shows the BER results for the A16 chip, and Table 4 summarizes the expected lifetime reduction for the remaining chips.

In all the chips, the BER in the first round of programming the low pages was extremely low, thanks to the lack of interference from the high pages. In the second round, however, the BER of all pages was higher than the baseline, and resulted in a reduction of lifetime greater than that caused by increasing V_{th}. We believe that a major cause of this difference are optimizations tailored for the regular LH programming order [39]. These optimizations are more common in recent chips, such as the B16 chip.

In the last set of experiments, we evaluated the long-term effects of reprogramming. Here, too, each experiment was composed of two parts: we programmed the block with LLH reprogramming in the first part, and with $P_{0.5}$ and regular programming in the second part. We varied the length of the first part between 20%, 40% and 60% of the block's lifetime. Figure 5 shows the BER results for the A16 chip, and Table 4 summarizes the expected lifetime reduction for the remaining chips.

We observe that the long-term effects of reprogramming are modest, and comparable to the long-term effects of increasing V_{th}. This supports our assumption that the additional short-term increase in BER observed in the previous set of experiments is not a result of the actual reprogramming process, but rather of the mismatch between the programming order the chips are optimized for and the LLH reprogramming scheme. This is especially evident in the B16 chip, in which the BER during the first part was high above the limit of 10^{-3}, but substantially smaller in the second part of the experiment.

Thus, schemes that reuse flash pages only at the beginning of the block's lifetime can increase its utilization without degrading its long-term reliability. Moreover, in all but the B16 chips, LLH reprogramming in the first 40% of the block's lifetime resulted in BER that was well within the error correction capabilities of the ECC. We rely on this observation in our FTL design in Section 4.

We note, however, that the variance between the chips we examined is high, and that short and long-term effects do not depend only on the feature size. For example, the A16 chip is "better" than the A27 chip, but the B16 chip is "worse" than the B29 chip. This leads to the second lesson from our low-level evaluation.

Lesson 2: *The portion of the block's lifetime in which its pages can be reused safely depends on the characteristics of its chip. The FTL must take into account the long-term implications of reuse on the chips it is designed for.*

3.5 Energy consumption

Flash read, write and erase operations consume different amounts of energy, which also depend on whether the operation is performed on the high page or on the low one, and on its data pattern. We examined the effect of reprogramming on energy consumption by connecting an oscilloscope to the SigNAS tester. We calculated the energy consumed by each of the following operations on the A16 chip: an erasure of a block programmed with P_{LLH} and p=0.5, reading and writing a high and a low page, reprogramming a low page, and programming a high page on a partially-used block.

To account for the transfer overhead of WOM encoded data, our measurements of read, program and reprogram operations included the I/O transfer to/from the registers. Our results, averaged over three independent measurements, are summarized in Table 5. We also present the average energy consumption per read or write operation with baseline and with LLH reprogramming, taking into account the size of the programmed data, the reading of

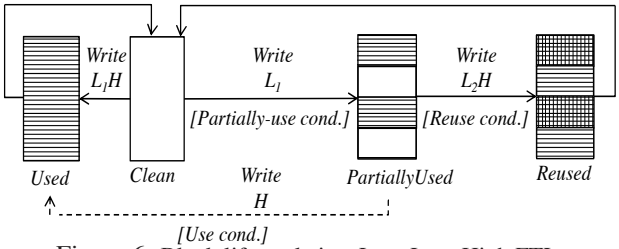


Figure 6: Block life cycle in a Low-Low-High FTL.

used pages for supplying the invalid data as input to the WOM encoder, and the number of pages that can be written before each erasure.

These results show that page reuse consumes more overall energy than the baseline. This is in contrast to previous studies showing possible energy savings. These studies assumed that the energy is proportional to the number of programmed *cells*, which is equivalent in a first and in a second write [18, 53]. However, our hardware evaluation shows that the number of reprogrammed *pages* is the dominant factor in energy consumption. While reprogramming a lower page consumes less energy than the average logical write in the baseline, the use of WOM encoding entails an extra read and page reprogram for each logical write. The low energy consumption of the saved erasures does not offset the additional energy consumed by those operations. We note, however, that when page reuse reduces the internal writes by the FTL, some energy savings may result. We examine that possibility further in Section 6, but can already draw the following lesson.

Lesson 3: *With WOM encoded data, the energy consumed by the additional flash operations is larger than that required by the saved erase operations. Energy savings are possible only if they reduce the number of write operations performed on the flash chip.*

4 FTL Design

Following our lessons from Section 3, we describe the general design principles for a *Low-Low-High FTL*—an FTL that reuses flash pages using the LLH reprogramming scheme. We assume such an FTL would run on the SSD controller, and utilize the physical page and block operations supported by the flash controller. Thus, it shares the following basic concepts with the standard FTL and SSD.

To accommodate out-of-place writes, the physical storage capacity of the drive is larger than its exported logical capacity. The drive's *overprovisioning* is defined as $\frac{T-U}{U}$, where T and U represent the number of physical and logical blocks, respectively [15]. Typical values of overprovisioning are 7% and 28% for consumer and enterprise class SSDs, respectively [45].

Whenever the number of clean blocks drops below a certain threshold, *garbage collection* is invoked. Garbage collection is typically performed *greedily*, picking the block with the minimum *valid count* (the lowest number of valid pages) as the victim for *cleaning*. The valid pages are *moved*—read and copied to another available block, and then the block is erased. These additional internal writes, referred to as *write amplification*, delay the cleaning process, and require, eventually, additional erasures. Write amplification does not accurately represent the utilization of drives that reuse pages for WOM encoded data, since some redundancy must always be added to the logical data to enable second writes [51, 52]. Thus, instead of deriving the number of erasures performed by the FTL from its write amplification, we measure them directly.

Low-Low-High (LLH) programming. Blocks in a Low-Low-High FTL cycle between four states, as depicted in Figure 6. In the initial, *clean* state all the cells are in the erased state, *ER*. If all the pages are programmed ($write\ L_1H$), the block reaches the *used* state. Alternatively, if only the low pages are used ($write\ L_1$), the block reaches the *partially-used* state. A partially-used block can be reused, in which case the FTL will reprogram all or some of the low pages and all the high pages ($write\ L_2H$), transitioning the block to the *reused* state. Alternatively, the FTL can program the high pages and leave the low pages untouched ($write\ H$), thus transitioning the block to the used state. Used and reused blocks return to the clean state when they are erased.

The choice of state transition is determined by the conditions depicted in Figure 6. The conditions that determine when to *partially use*, *use* or *reuse* a block, as well as the encoding scheme used for reprogrammed pages, are in turn determined by the specific FTL design. We next describe *LLH-FTL*—the FTL used for our evaluation.

WOM encoding. When WOM codes are employed for reusing flash pages, the FTL is responsible for determining whether a logical page is written in a first or a second write, and for recording the required metadata. The choice of WOM code determines the data written on the low pages of partially-used blocks, and the data written on them when they are reprogrammed. The encoding scheme in LLH-FTL is similar to that of ReusableSSD [53]. Data in the low pages of partially-used blocks is written as is, without storage or encoding overheads. Data written as a second write on low pages of reused blocks is encoded with a Polar WOM code that requires two physical pages to store the encoded data of one logical page [9, 10]. This WOM implementation has a 0.25% encoding failure rate.

We note that the mathematical properties of WOM codes ensure they can be applied to any data pattern, including data that was previously scrambled or compressed. In fact, WOM encoding also ensures an even distribution of zeroes throughout the page, and can thus replace data scrambling on second writes.

While manufacturers have increased the flash page size (see Table 2), the most common size used by file systems remains 4KB. Our LLH-FTL design distinguishes between the logical page used by the host and some larger

physical page size. Thus, the FTL maps several logical pages onto each physical page. This allows LLH-FTL to program the encoded data for a second write on one physical page. In the rest of this section we assume that the physical page size is exactly twice the logical page size. We note that the changes required in the design if the physical pages are even larger are straightforward.

If the physical and logical page sizes are equal, a Low-Low-High FTL can utilize the multi-plane command that allows programming two physical pages in parallel on two different blocks, as in the ReusableSSD design. In both approaches, the latency required for reading or writing an encoded logical page on a second write is equal to the latency of one flash page write.

As in the design of ReusableSSD [53], LLH-FTL addresses the 0.25% probability of encoding failure by writing the respective logical page as a first write on a clean block, and prefetches the content of physical pages that are about to be rewritten to avoid the latency of an additional read. Pages are reprogrammed only in the safe portion of their block's lifetime (the first 40% in all but one of the chips we examined), thus limiting the long-term effect of reprogramming to an acceptable level.

Hot and cold data separation. Workloads typically exhibit a certain amount of skew, combining frequently updated *hot* data with infrequently written *cold* data. Separating hot and cold pages has been demonstrated as beneficial in several studies [16, 22, 25, 47]. Previous studies also showed that second writes are most beneficial for hot pages, minimizing the time in which the capacity of reused blocks is not fully utilized [31, 36, 52, 53]. In LLH-FTL, we write hot data on partially-used and reused blocks, and cold data on used blocks. Hot data on partially-used blocks is invalidated quickly, maximizing the benefit from reusing the low pages they are written on. Reused blocks store pages in first as well as in second writes. Nevertheless, we use them only for hot data, in order to maintain the separation of hot pages from cold ones. The classification of hot and cold pages is orthogonal to the design of LLH-FTL, and can be done using a variety of approaches [12, 22, 33, 47]. We describe the classification schemes used in our experiments in Section 5.

Partially-use, use and reuse conditions. The number of partially-used blocks greatly affects the performance of a Low-Low-High FTL. Too few mean that the blocks will be reused too soon, while they still contain too many valid low pages, thus limiting the benefit from reprogramming. Too many mean that too many high pages will remain unused, reducing the available overprovisioned space, which might increase internal page moves. The three conditions in Figure 6 control the number of partially-used blocks: if the partially-use condition does not hold, a clean block is used with regular LH programming. In addition, the FTL may define a use condition, which specifies the circumstances in which a partially-used block is reclaimed, and its high pages will be written without rewriting the low pages. Finally, the reuse condition ensures efficient reuse of the low pages. The FTL allows partially-used blocks to accumulate until the reuse condition is met.

Our LLH-FTL allows accumulation of at most $threshold_{pu}$ partially-used blocks. This threshold is updated in each garbage collection invocation. An increase in the valid count of the victim block compared to previous garbage collections indicates that the effective overprovisioned space is too low. In this case the threshold is *decreased*. Similarly, a decrease in the valid count indicates that page reuse is effective in reducing garbage collections, in which case the threshold is *increased* to allow more reuse. Thus, the partially-use and reuse conditions simply compare the number of partially-used blocks to the threshold. To maintain the separation between hot and cold pages, LLH-FTL does not utilize the use condition.

Expected benefit. The reduction in erasures in LLH-FTL depends on the amount of hot data in the workload, and on the number of valid pages that remain on partially-used blocks when they are reused. We assume, for the sake of this analysis, that the low pages on a reused block, as well as all the pages on an erased block, have all been invalidated. Without reprogramming, this means that there is no write amplification, and the expected number of erasures is $E{=}\frac{M}{N}$, where M is the number of logical page write requests, and N is the number of pages in each block. With LLH programming, every two low pages are reused to write an extra logical page, so $N{+}\frac{N}{4}$ logical pages are written on each block before it is erased. Let X be the portion of hot data in the workload, $0 \leq X < 1$, and recall that only blocks containing hot pages are reused. Then the expected number of erasures is $E'{=}(1{-}X)\frac{M}{N}{+}X\frac{M}{N+\frac{N}{4}}{=}E(\frac{5-X}{5})$. The maximal reduction in erasures is expected in traces where almost all the write requests access hot pages ($X \rightarrow 1$), where $E' = 0.8E$, a reduction of 20%.

For a rough estimate of the resulting lifetime extension, let us assume that all the blocks are reused in the first 40% of their lifetime, i.e., during $0.4T$ cycles. In each of these cycles, $\frac{5N}{4}$ logical pages are written on these blocks, a total of $0.5TN$. Assuming we can use the remaining $0.6T$ cycles, we write an additional $0.6TN$ pages. The total amount of data written is $1.1TN$, an increase of 10% compared to regular programming. However, we must also consider the reduction in lifetime observed in the experiments in Section 3.3. A 5%–6% reduction means that the reduction in erasures translates to a modest 4%–5% increase in lifetime.

Comparing our analysis to that of previous designs is not straightforward. Most studies, including of designs that reuse flash pages with WOM codes, did not consider the overall amount of *logical data* that could be written

on the device. The only comparable analysis is that of ReusableSSD [53], which resulted in an estimated reduction of up to 33% of erasures, assuming that all the blocks (storing both hot and cold data) could be reused, and that both the low and high pages could be reprogrammed. This analysis also excluded the lifetime reduction due to reprogramming. This discrepancy leads to our next lesson.

Lesson 4: *A reduction in erasures does not necessarily translate to a substantial lifetime increase, due to the low utilization of pages that store WOM encoded data, and to the long-term effects of reprogramming. The increase in lifetime strongly depends on chip characteristics.*

5 SSD Evaluation Setup

In our FTL evaluation, we wish to quantify the possible benefit from reusing flash pages with WOM codes, when all the limitations of physical MLC flash and practical codes are considered. Thus, we measure the savings in erasures and the lifetime extension they entail, as well as the effects of LLH reprogramming on I/O performance.

5.1 OpenSSD evaluation board

We use the OpenSSD Jasmineboard [4] for our FTL evaluation. The board includes an ARM-based Indilinx™ Barefoot controller, 64MB of DRAM for storing the flash translation mapping and SATA buffers, and eight slots for custom flash modules, each of which can host four 64Gb 35nm MLC flash chips. The chips have two planes and 8KB physical pages. The device uses large 32KB virtual pages for improved parallelism. Thus, erase blocks are 4MB and consist of 128 contiguous virtual pages [4].

On the OpenSSD board, an FTL that uses 8KB pages rather than 32KB virtual pages incurs unacceptable latencies [43]. Thus, we use a mapping granularity of 4KB logical pages and a merge buffer that ensures that data is written at virtual-page granularity [31, 43]. The downside of this optimization is an exceptionally large block size (1024 logical pages) that increases the valid count of used and partially-used blocks. As a result, garbage collection entails more page moves, and reprogramming is possible on fewer pages.

We were also unable to fully implement WOM encoded second writes on the OpenSSD board. While mature and commonly used error correction codes are implemented in hardware, the Polar WOM codes used in our design are currently only available with software encoding and decoding. These implementations are prohibitively slow, and are thus impractical for latency evaluation purposes. In addition, in OpenSSD, only the ECC hardware accelerator is allowed to access the page spare area, and cannot be disabled. Thus, reprogrammed pages will always appear as corrupted when compared with their corresponding ECC. This also prevents the FTL from utilizing the page spare area for encoding purposes [53]. We address these limitations in our FTL implementation described below.

5.2 FTL Implementation

The FTL used on the OpenSSD board is implemented in software, and can thus also be used as an *emulator* of SSD performance when executed on a standard server without being connected to the actual board. Replaying a workload on the emulator is considerably faster than on the board itself, because it does not perform the physical flash operations. We validated this emulator, ensuring that it reports the same *count* of flash operations as would be performed on the actual board. Thus, using the emulator, we were able to experiment with a broad set of setups and parameters that are impractical on the Jasmine board. In particular, we were able to evaluate an FTL that uses 8KB physical pages, rather than the 32KB physical pages mandated by the limitations of the board.We refer to the FTL versions with 32KB pages as $\langle FTL\ name \rangle$-32.

We first implemented a *baseline* FTL that performs only first writes on all the blocks. It employs greedy garbage collection within each bank and separates hot and cold pages by writing them on two different active blocks. The identification of hot pages is described in Section 5.3. We also implemented LLH-FTL described in Section 4.[4] It uses greedy garbage collection for choosing the block with the minimum number of valid logical pages among used and reused blocks. Garbage collection is triggered whenever a clean block should be allocated and no such block is available. If the number of partially-used blocks is lower than the threshold and a hot active block is required, LLH-FTL allocates the partially-used block with the minimum valid count. If the threshold is exceeded or if a cold active block is required, it allocates a new clean block.

The threshold is updated after each garbage collection, taking into account the valid count in w previous garbage collections. Due to lack of space, we present results only for $w = 5$, and two initial threshold values, which were the most dominant factor in the performance of LLH-FTL.

LLH-FTL reuses low pages on partially-used blocks only if all the logical pages on them have been invalidated. LLH-FTL-32 writes four logical pages on each reused physical pages, requiring eight consecutive invalid logical pages in order to reuse a low page. We evaluate the effect of this limitation on LLH-FTL-32 in Section 6.

Our implementation of LLH-FTL does not include actual WOM encoding and decoding for the reasons described above. Instead, it writes arbitrary data during reprogramming of low pages, and ignores the ECC when reading reprogrammed data. In a real system, the WOM encoding and decoding would be implemented in hardware, and incur the same latency as the ECC. Thus, in our evaluation setup, their overheads are simulated by the ECC computations on the OpenSSD board. Coding failures are simulated by a random "coin flip" with the appropriate probability. To account for the additional prefetch-

[4] The code for the emulator and FTLs is available online [1, 2].

Volume	Requests (M)	Drive size (GB)	Requests/sec	Write ratio	Hot write ratio	Total writes (GB)
src1_2	2	16	3.15	0.75	0.22	45
stg_0	2		3.36	0.85	0.85	16
hm_0	4	32	6.6	0.64	0.7	23
rsrch_0	1.5		2.37	0.91	0.95	11
src2_0	1.5		2.58	0.89	0.91	10
ts_0	2		2.98	0.82	0.94	12
usr_0	2.5		3.7	0.6	0.86	14
wdev_0	1		1.89	0.8	0.85	7
prxy_0	12.5	64	20.7	0.97	0.67	83
proj_0	4		6.98	0.88	0.14	145
web_0	2		3.36	0.7	0.87	17
online	5.5	16	3.14	0.74	0.31	16
webresearch	3		1.8	1	0.41	13
webusers	8		4.26	0.9	0.32	27
webmail	8	32	4.3	0.82	0.3	24
web-online	14		7.88	0.78	0.31	43
zipf(0.9,0.95,1)	12.5	16	200	1	0.5	48

Table 6: Trace characteristics of MSR (top box), FIU (middle), and synthetic (bottom) workloads.

ing of the invalid data, this data is read from the flash into the DRAM, but is never transferred to the host.

5.3 Workloads

We use real world traces from two sources. The first is the MSR Cambridge workload [5, 35], which contains week-long traces from 36 volumes on 13 servers. The second is a set of traces from FIU [3, 29], collected during three weeks on servers of the computer science department. Some of the volumes are too big to fit on the drive size supported by our FTL implementation, which is limited by the DRAM available on the OpenSSD. We used the 16 traces whose address space fit in an SSD size of 64GB or less, and that include enough write requests to invoke the garbage collector on that drive. These traces vary in a wide range of parameters, summarized in Table 6. We also used three synthetic workloads with a Zipf distribution with exponential parameter $\alpha = 0.9, 0.95$ and 1.

We used a different hot/cold classification heuristic for each set of traces. For the MSR traces, pages were classified as cold if they were written in a request of size 64KB or larger. This simple online heuristic was shown to perform well in several previous studies [12, 22, 53]. In the FIU traces, all the requests are of size 4KB, so accesses to contiguous data chunks appear as sequential accesses. We applied a similar heuristic by tracking previously accessed pages, and classifying pages as cold if they appeared in a sequence of more than two consecutive pages. In the synthetically generated Zipf traces, the frequency of access to page n is proportional to $\frac{1}{\alpha^n}$. Thus, we extracted the threshold n for each Zipf trace, such that pages with logical address smaller than n were accessed 50% of the time, and pages with logical address larger than n were classified as cold. While this classification is impossible in real world settings, these traces demonstrate the benefit from page reuse under optimal conditions.

Each workload required a different device size, and thus, a different number of blocks. In order to maintain the same degree of parallelism in all experiments, we always configured the SSD with 16 banks, with 256, 512 and 1024 4MB blocks per bank for drives of size 16GB, 32GB and 64GB, respectively. Pages were striped across banks, so that page p belonged to bank $b = p \ mod \ 16$.

6 Evaluation

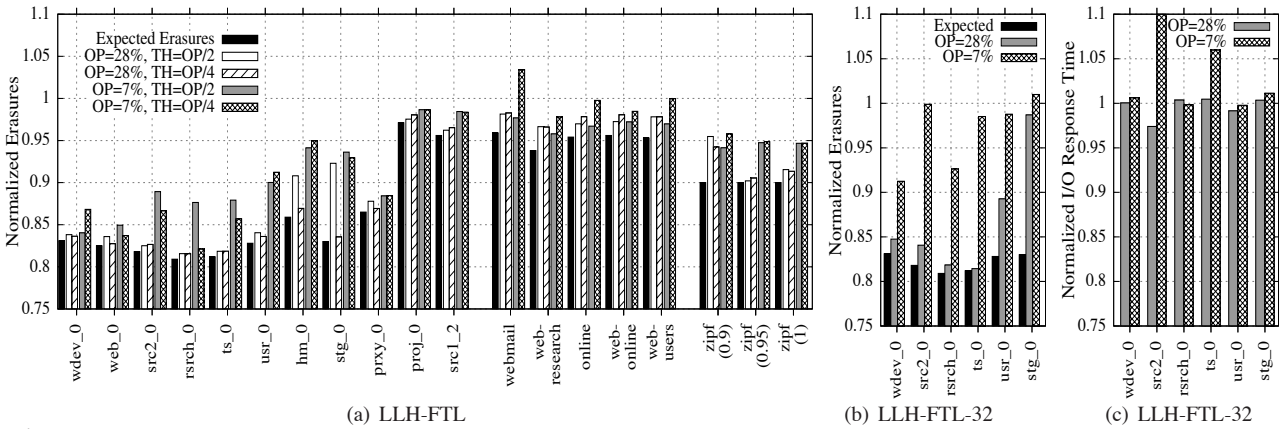
Reduction in erasures. To verify that the expected reduction in erasures from LLH reprogramming can be achieved in real workloads, we calculated the expected reduction for each workload according to the formula in Section 4. We then used the emulator to compare the number of erasures performed by the baseline and LLH-FTL. Our results are presented in Figure 7(a), where the workloads are aggregated according to their source (and corresponding hot page classification) and ordered by the amount of data written divided by the corresponding drive size. Our results show that the normalized number of erasures is between 0.8 and 1. The reduction in erasures mostly depends on the workload and the amount of hot data in it.

The amount of overprovisioning (OP) substantially affects the benefit from reprogramming. With 28% overprovisioning, the reduction in erasures is very close to the expected reduction. Low overprovisioning is known to incur excessive internal writes. Thus, with the already low 7% overprovisioning, reserving partially-used blocks for additional writes was not as efficient for reducing erasures; it might increase the number of erasures instead. The adaptive $threshold_{pu}$ avoids this situation quite well, as it is decreased whenever the valid count increases. Still, the reduction in erasures is smaller than with OP=28% because both the low overprovisioning and low threshold result in more valid logical pages on the partially-used blocks, allowing fewer pages to be reused.

The time required for the adaptive $threshold_{pu}$ to converge depends on its initial value. In setups where the reservation of partially-used blocks is useful, such as high overprovisioning, LLH-FTL with initial $threshold_{pu}$ = OP/2 achieves greater reduction than with $threshold_{pu}$ = OP/4, because a higher initial value means that the optimal value is found earlier. The difference between the two initial values is smaller for traces that write more data, allowing the threshold more time to adapt.

The quality of the hot data classification also affected the reduction in erasures. While the baseline and LLH-FTL use the same classification, misclassification interferes with page reuse in a manner similar to low overprovisioning, as it increases the number of valid logical pages during block erase and reuse. This effect is demonstrated in the lower reductions achieved on the FIU workloads, in which classification was based on a naive heuristic.

We repeated the above experiments with LLH-FTL-32,



(a) LLH-FTL (b) LLH-FTL-32 (c) LLH-FTL-32

Figure 7: (a) Normalized number of erasures (compared to baseline) of LLH-FTL (b) Normalized number of erasures (compared to baseline-32) of LLH-FTL-32 (c) Normalized I/O response time (compared to baseline-32) of LLH-FTL-32 (c).

to evaluate the effect of increasing the physical page size. Indeed, the reduction in erasures was smaller than with 8KB pages, although the differences were minor. The average difference was 1% with 28% overprovisioning, but it was 6% with 7% overprovisioning because of the higher number of leftover valid logical pages on each physical page in partially-used blocks.

I/O response time. To evaluate the effect of LLH reprogramming on I/O response time, we replayed the workloads on the OpenSSD board. We warmed up the board by filling the SSD and then replaying the workload twice, measuring the I/O response time in the last 12 hours of each workload. We accelerated the workloads by a factor of 10 in order to speed up the experiment. While maintaining the original access pattern, the accelerated request rate is more realistic for workloads that use SSDs.

We use LLH-FTL-32 with the optimal initial $threshold_{pu}$ for representative MSR traces. Figure 7(b) shows the normalized number of erasures compared to baseline-32, and Figure 7(c) shows the normalized I/O response time of LLH-FTL-32. Despite the considerable reduction in erasures, and thus, garbage collection invocations, the average I/O response time is almost unchanged. The 90th and 99th percentiles were also similar. This contradicts previous simulation results [53] that correlated the reduction in erasures with a reduction in I/O response time.

One reason for this discrepancy is that the accumulation of write requests in the merge buffers in OpenSSD causes writes to behave asynchronously—the write request returns to the host as complete once the page is written in the buffer. Flushing the merge buffer onto a physical flash page is the cause for latency in writes. The baseline flushes the buffer whenever eight logical pages are accumulated in it. However, a buffer containing WOM encoded data must be flushed after accumulating four logical pages, possibly incurring additional latency. This effect was not observed in previous studies that used a simulator that flushes all writes synchronously.

The average I/O response time does not *increase* because even though the trace is accelerated, the extra buffer flushes usually do not delay the following I/O requests. In addition, due to the allocation of partially-used and reused pages for hot data, this data is more likely to reside on low pages, which are faster to read and program [18].

The second reason for the discrepancy is the reservation of partially-used blocks for reprogramming. This reduces the available overprovisioned capacity, potentially increasing the number of valid pages that must be copied during each garbage collection. As a result, although the number of erasures decreased, the total amount of data copied by LLH-FTL-32 was similar to that copied by the baseline, and sometimes higher (by up to 50%). One exception is the src1_2 workload, where in the last 12 hours, garbage collection in LLH-FTL-32 moved less data than in baseline-32. In the other traces, the total delay caused by garbage collections was not reduced, despite the considerably lower number of erasures.

Energy consumption. We used the values from Table 5 and the operation counts from the emulator to compare the energy consumption of LLH-FTL-32 to that of baseline-32. The energy measurements were done on the A16 chip, whose page size is 16KB. We doubled the values for the read and program operations to estimate the energy for programming 32KB pages as in LLH-FTL-32. Figure 8 shows that when reprogramming reduced erasures, the energy consumed by LLH-FTL-32 increased with inverse proportion to this reduction. This is not surprising, since the reduction in erasures does not reduce the amount of internal data copying in most of the workloads. In the FIU traces with 7% OP, reprogramming increased the number of erasures due to increased internal writes, which, in turn, also increased the energy consumption.

Lesson 5: *A reduction in erasures does not necessarily translate to a reduction in I/O response time or energy consumption. These are determined by the overall amount*

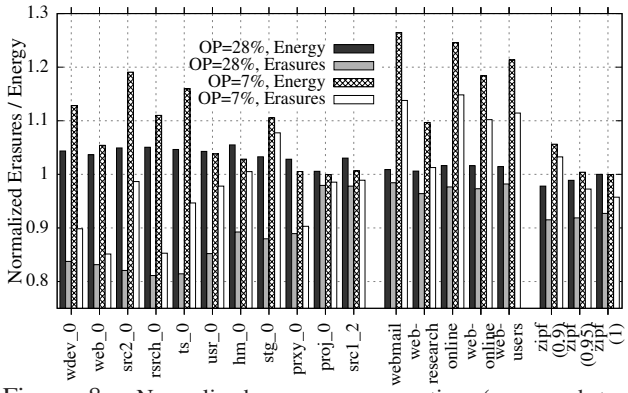


Figure 8: Normalized energy consumption (compared to baseline-32) of LLH-FTL-32 with two overprovisioning values.

of data moved during garbage collections. Designs that are aimed at reducing energy consumption or I/O response time should address these objectives explicitly.

7 Related Work

Several studies proposed FTL designs that reuse pages to extend SSD lifetime. Some are based on capacity achieving codes, and bound the resulting capacity loss by limiting second writes to several blocks [36] or by assuming the logical data has been compressed by the upper level [24]. The overheads and complexities in these designs are addressed in the design of ReusableSSD [53]. However, none of these studies addressed the limitations of reprogramming MLC flash pages. Some of these limitations were addressed in the design of an overwrite compatible B^+-tree data structure, assuming the mapping of V_{th} to bits can be modified [26]. Like the previous approaches, it has been implemented only in simulation. Extended P/E cycles [31] were implemented on real hardware, but the FTL that uses them relies on the host to supply and indicate data that is overwrite compatible. LLH-FTL is the first general-purpose FTL that addresses all practical limitations of WOM codes as well as MLC flash. Thus, we were able to demonstrate its strengths and weaknesses on real hardware and workloads.

Numerous studies explored the contributors to BER in flash, on a wide variety of chip technologies and manufacturers. They show the effects of erasures, retention, program disturbance and scaling down technology on the BER [11, 18, 32, 48]. These studies demonstrate a trend of increased BER as flash feature sizes scale down, and the need for specialized optimizations employed by manufacturers as a result. Thus, we believe that some of the interference effects observed in our experiments are a result of optimizing the chips for regular LH programming. Adjusting these optimizations to LLH reprogramming is a potential approach to increase the benefit from page reuse.

Several studies examined the possibility of reprogramming flash cells. Most used either SLC chips [24], or MLC chips as if they were SLC [17]. A thorough study on 50nm and 72nm MLC chips demonstrated that after a full use of the block (LH programming), half of the pages are "WOM-safe" [18]. However, they do not present the exact reprogramming scheme, nor the problems encountered when using other schemes. A recent study [31] mapped all possible state transitions with reprogramming on a 35nm MLC chip, and proposed the LLH reprogramming scheme. Our results in Section 3 show that smaller feature sizes impose additional restrictions on reprogramming, but that LLH reprogramming is still possible.

Previous studies examined the energy consumption of flash chips as a factor of the programmed pattern and page [34], and suggested methods for reducing the energy consumption of the flash device [41]. To the best of our knowledge, this study is the first to measure the effect of reprogramming on the energy consumption of a real flash chip and incorporate it into the evaluation of the FTL.

8 Conclusions

Our study is the first to evaluate the possible benefit from reusing flash pages with WOM codes on real flash chips and an end-to-end FTL implementation. We showed that page reuse in MLC flash is possible, but can utilize only half of the pages and only if some of its capacity has been reserved in advance. While reprogramming is safe for at least 40% of the lifetime of the chips we examined, it incurs additional *long-term* wear on their blocks. Thus, even with an impressive 20% reduction in *erasures*, the increase in *lifetime* strongly depends on chip physical characteristics, and is fairly modest.

A reduction in erasures does not necessarily translate to a reduction in I/O response time or energy consumption. These are determined by the overall amount of data moved during garbage collections, which strongly depends on the overprovisioning. The reduction in physical flash page writes is limited by the storage overhead of WOM encoded data, and is mainly constrained by the limitation of reusing only half of the block's pages.

This study exposed a considerable gap between the previously shown benefits of page reuse, which were based on theoretical analysis and simulations, and those that can be achieved on current state-of-the-art hardware. However, we believe that most of the limitations on these benefits can be addressed with manufacturer support, and that the potential benefits of page reuse justify reevaluation of current MLC programming constraints.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Andrea Arpaci-Dusseau, whose suggestions helped improve this paper. We also thank Alex Yucovich and Hila Arobas for their help with the low-level experiments. This work was supported in part by BSF grant 2010075, NSF grant CCF-1218005, ISF grant 1624/14 and EU Marie Curie Initial Training Network SCALUS grant 238808.

References

[1] https://github.com/zdvresearch/fast2016-ftl.

[2] https://github.com/zdvresearch/fast2016-openssd-emulator.

[3] I/O deduplication traces. http://sylab-srv.cs.fiu.edu/doku.php?id=projects:iodedup:start. Retrieved: 2014.

[4] Jasmine OpenSSD platform. http://www.openssd-project.org/.

[5] SNIA IOTTA. http://iotta.snia.org/traces/388. Retrieved: 2014.

[6] NAND flash memory tester (SigNASII). http://www.siglead.com/eng/innovation_signas2.html, 2014.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC)*, 2008.

[8] A. Berman and Y. Birk. Retired-page utilization in write-once memory – a coding perspective. In *IEEE International Symposium on Information Theory (ISIT)*, 2013.

[9] D. Burshtein. Coding for asymmetric side information channels with applications to polar codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[10] D. Burshtein and A. Strugatski. Polar write once memory codes. *IEEE Transactions on Information Theory*, 59(8):5088–5101, 2013.

[11] Y. Cai, O. Mutlu, E. Haratsch, and K. Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *31st IEEE International Conference onComputer Design (ICCD)*, 2013.

[12] M.-L. Chiao and D.-W. Chang. ROSE: A novel flash translation layer for NAND flash memory based on hybrid address translation. *IEEE Transactions on Computers*, 60(6):753–766, 2011.

[13] G. D. Cohen, P. Godlewski, and F. Merkx. Linear binary code for write-once memories. *IEEE Transactions on Information Theory*, 32(5):697–700, 1986.

[14] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[15] P. Desnoyers. What systems researchers need to know about NAND flash. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[16] P. Desnoyers. Analytic models of SSD write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[17] E. En Gad, H. W., Y. Li, and J. Bruck. Rewriting flash memories by message passing. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[18] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[19] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[20] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[21] J.-W. Im et al. A 128Gb 3b/cell V-NAND flash memory with 1gb/s i/o rate. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2015.

[22] S. Im and D. Shin. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.*, 56(12):641–653, Dec. 2010.

[23] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Writing cosets of a convolutional code to increase the lifetime of flash memory. In *50th Annual Allerton Conference on Communication, Control, and Computing*, 2012.

[24] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in NAND flash through multi-write coding. In *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[25] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[26] J. Kaiser, F. Margaglia, and A. Brinkmann. Extending SSD lifetime in database applications with page overwrites. In *6th International Systems and Storage Conference (SYSTOR)*, 2013.

[27] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[28] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *6th USENIX Conference on File and Storage*

Technologies (FAST), 2008.

[29] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[30] X. Luojie, B. M. Kurkoski, and E. Yaakobi. WOM codes reduce write amplification in NAND flash memory. In *IEEE Global Communications Conference (GLOBECOM)*, 2012.

[31] F. Margaglia and A. Brinkmann. Improving MLC flash performance and endurance with extended P/E cycles. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[32] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill. Bit error rate in NAND flash memories. In *Reliability Physics Symposium (IRPS). IEEE International*, 2008.

[33] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[34] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. Stan, and S. Swanson. Modeling power consumption of NAND flash memories using FlashPower. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1031–1044, July 2013.

[35] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[36] S. Odeh and Y. Cassuto. NAND flash architectures reducing write amplification through multi-write codes. In *IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[37] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[38] H. Park, J. Kim, J. Choi, D. Lee, and S. Noh. Incremental redundancy to reduce data retention errors in flash-based SSDs. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[39] K.-T. Park, M. Kang, D. Kim, S.-W. Hwang, B. Y. Choi, Y.-T. Lee, C. Kim, and K. Kim. A zeroing cell-to-cell interference page architecture with temporary LSB storing and parallel MSB program scheme for MLC NAND flash memories. *IEEE Journal of Solid-State Circuits*, 43(4):919–928, April 2008.

[40] R. L. Rivest and A. Shamir. How to Reuse a Write-Once Memory. *Inform. and Contr.*, 55(1-3):1–19, Dec. 1982.

[41] M. Salajegheh, Y. Wang, K. Fu, A. Jiang, and E. Learned-Miller. Exploiting half-wits: Smarter storage for low-power devices. In *9th USENIX Conference on File and Stroage Technologies (FAST)*, 2011.

[42] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[43] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting real: Lessons in transitioning research simulations into hardware systems. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[44] A. Shpilka. Capacity achieving multiwrite WOM codes. *IEEE Transactions on Information Theory*, 60(3):1481–1487, 2014.

[45] K. Smith. Understanding SSD over-provisioning. *EDN Network*, January 2013.

[46] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[47] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[48] E. Yaakobi, L. Grupp, P. Siegel, S. Swanson, and J. Wolf. Characterization and error-correcting codes for TLC flash memories. In *International Conference on Computing, Networking and Communications (ICNC)*, 2012.

[49] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf. Codes for write-once memories. *IEEE Transactions on Information Theory*, 58(9):5985–5999, 2012.

[50] E. Yaakobi, J. Ma, L. Grupp, P. H. Siegel, S. Swanson, and J. K. Wolf. Error characterization and coding schemes for flash memories. In *IEEE GLOBECOM Workshops (GC Wkshps)*, 2010.

[51] E. Yaakobi, A. Yucovich, G. Maor, and G. Yadgar. When do WOM codes improve the erasure factor in flash memories? In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[52] G. Yadgar, R. Shor, E. Yaakobi, and A. Schuster. It's not where your data is, it's how it got there. In *7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2015.

[53] G. Yadgar, E. Yaakobi, and A. Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[54] G. Yadgar, A. Yucovich, H. Arobas, E. Yaakobi, Y. Li, F. Margaglia, A. Brinkmann, and A. Schuster. Limitations on MLC flash page reuse and its effects on durability. Technical Report CS-2016-02, Computer Science Department, Technion, 2016.

[55] J. Yang, N. Plasson, G. Gillis, and N. Talagala. HEC: Improving endurance of high performance flash-based cache devices. In *6th International Systems and Storage Conference (SYSTOR)*, 2013.

[56] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.