UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# AN EXPLORATION INTO ADAPTIVE METHODS FOR DECREASING WEAR-LEVELING IN SCM

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

**Isaak Cherdak**

March 2019

The Thesis of Isaak Cherdak
is approved:

Prof. Darrell Long, Chair

Prof. Peter Alvaro

Prof. Heiner Litz

Lori Kletzer
Vice Provost and Dean of Graduate Studies

# Table of Contents

# List of Figures

# List of Tables

**Abstract**

An exploration into adaptive methods for decreasing wear-leveling in SCM

by

Isaak Cherdak

Storage Class Memories (SCM) have recently emerged as promising technologies for use as system memory because of their advantages such as non-volatility, byte addressability and low idle power usage. Nevertheless, lower write endurance, higher asymmetric read/write latencies, and stronger consistency requirements pose new challenges for using SCM rather than DRAM as the next generation of memory. In this report, we focus on endurance challenges in SCM. More specifically, we challenge traditional simplified wear-leveling methods like Flip-N-Write [10] by exploring the merits of adapting to data sets dynamically. In the process, we develop a novel method for improving wear leveling on SCM: Adaptive Bit Flip Pattern Learning (ABFPL). We show that our method works best in software rather than as a hardware implementation since it allows for more adaptability to changing workloads. We provide and demonstrate a preliminary configuration which can improve wear in a larger set of datasets than previous approaches. We evaluate our method and show it to have up to a improvement of 57% over Flip-N-Write [10].

I dedicate this to my mother, who has shown me that people can grow when you least expect them to. To that end, I also dedicate this to growing into a stronger and better version of myself.

# Acknowledgments

I would primarily like to acknowledge Professor Darrell Long for giving me the opportunity to improve both my outlook as a life-long-learner and my ability to thrive in challenging environments. I would also like to thank my entire committee, Darrell Long, Peter Alvaro, and Heiner Litz, for their involvement in the development of this work. I would like to thank Daniel Bittman for being a fantastic role model and guide for my graduate career, as well as for setting the groundwork in bit-flips as a metric which this work further develops. I would also like to thank Saeed Kargar for helping me begin and develop the topic for this work. I would like to thank James Byron for his wisdom and constant support throughout this work. Finally, I would like to thank the rest of my lab-mates at the Storage Systems Research Center (SSRC) for both their emotional and technical support throughout my graduate career. Last but not least, I would like to thank Garvit Rajendra Mantri and Faina Cherdak for providing helpful feedback from an educational perspective outside of the topic for this work.

# Chapter 1

# Introduction

## 1.1 Existing approach to utilizing memory

Much of computing today is designed around the assumption that memory is volatile and has much lower latency than that of persistent storage. DRAM is usable on a byte-addressable bus, enabling its use as the store for all data that is actively used in various applications that require in place updates. In addition, DRAM provides flexibility for applications utilizing data structures which don't maintain failure consistency. Maintaining consistency after failures has been less challenging in systems that utilize only DRAM since such errors do not persist in memory across reboots. At the same time, this means that memory is looked at in a more general way—as a storage write-back cache. However, failure consistency models for in-memory data structures are being reconsidered with the advent of Storage Class Memory (SCM) due to its persistence across reboots.

## 1.2 Storage Class Memory is coming

Storage Class Memory [6] is a class of technologies that provides a middle ground between memory and storage. Thus, like memory, it provides lower latencies, byte-addressability, and usability on the memory bus. Additionally, it provides data persistence across power cycles like persistent storage does. While SCM enables new applications, such as instant reboot [42] after a safe shutdown, and quick recovery [1] after unexpected crashes, it introduces new problems as well. Among these issues exist concerns for crash consistency [43] and wear-leveling [38]. In particular, wear-leveling is important in SCM because unlike current persistent devices, such as SSDs and HDDs which can automatically distribute writes evenly, SCMs are byte addressable and thus permit use cases that cause certain sectors to be written more frequently. Nevertheless, SCM technologies are promising, albeit with much work still to be done before they can be deemed usable for consumers.

## 1.3 The challenges of adopting SCM

SCM permits new use methods while posing new challenges to usability. Studies have shown that the closest SCM technology to being released for commercial use on the memory bus is Phase Change Memory (PCM) [28], which operates by heating Chalcogenide cells to one of two states. The state of a cell is interpreted as a one or zero and can be read much more quickly than can be overwritten. A resulting difference is for PCM power usage, which scales primarily based on writes that require a voltage spike to reheat the Chalcogenide, whereas reads have negligible power utilization for reading a cell's state. On the other hand, DRAM has constant power scaling [46, 44] since it must be refreshed at a constant rate, regardless of the frequency of reads and writes. This is done in order to avoid

losing data. Another difference is that PCM has write latencies that are up to two orders of magnitude higher than DRAM. More differences include the contrast between a PCM [38] endurance of $10^8$ writes, and a DRAM [28, 2, 27] endurance of $10^{16}$ writes. These differences boil down to one major question: is it worthwhile to consider algorithms that trade writes for additional reads [7]? In addition, simply decreasing the total number of writes is not necessarily the best approach to decrease wear. As Bittman *et al.* demonstrate [5], we should focus on reducing the Hamming distance, or the sum of bit-wise differences, when a write is performed. Since SCM controllers only need to write the bits that are being flipped, Hamming distance is a good metric for accurately measuring SCM wear-leveling.

## 1.4 Prior SCM wear-leveling approaches

Some existing approaches to wear-leveling focus on simple ways to decrease the Hamming distance when a write occurs. These methods trade off additional space and writes for fewer bits flipped. More specifically, they write an offset which uniquely identifies the operation required to recover the original data. Generally, these methods are very simple and as such, are proposed for implementation at the hardware level.

Flip-N-Write [10] is a good example of existing SCM wear-leveling approaches. In Flip-N-Write, a two bit offset is used to encode a changeset. If the Hamming distance of a given half is more than 25% of the total Hamming distance, the half gets inverted before being written. In addition, the corresponding offset bit is set if the write is inverted. Despite the benefits of Flip-N-Write, this method is too simple to adapt to any dataset dynamically. For example, a bit string of alternating ones and zeroes is a worst case scenario for the Flip-N-Write approach since both halves have a Hamming distance of 25% the string length. This continues

to be the case, regardless of how many times the bit string occurs. Therefore, instead of this simplistic approach, we decided to use a technique that can decrease bit-flips while also adapting to a larger variety of datasets.

## 1.5 A New SCM wear-leveling approach

Existing systems don't provide mechanisms for adapting to datasets. This leaves them in a position where the same worst case data can be provided repeatedly with no improvement over time as shown with Flip-N-Write above. We define adaptive wear-leveling methods as those that save information about writes over time to dynamically decrease their effect on endurance. Adaptive systems also could have a number of parameters such as the frequency at which information on writes is saved, and the maximum amount of information that can be saved at a time. These parameters must be chosen differently depending on the system for best results. Existing methods are simple enough to implement in the hardware and would not gain any advantage from a software-level implementation. However, while adaptive systems are likely to be more difficult to implement in hardware due to their added complexity, they will also benefit from a software-level implementation to maximize their flexibility. To the best of our knowledge, no adaptive SCM wear-leveling methods have been proposed up until now.

## 1.6 Introducing ABFPL

To counter existing traditional methods, we propose a new approach called Adaptive Bit-Flip Pattern Learning (ABFPL), which is designed to explore the territory of adaptive wear-leveling on SCM. ABFPL works by tracking bit-flips at per-bit granularity for each write requested, in order to dynamically construct

an array of *patterns*—or bitmasks—which reflect the bits that flip the most. To compare, Flip-N-Write can be seen as a method with four static patterns, denoting the inversion of left half, right half, all, or none of the bits. In contrast, ABFPL can choose from a number of dynamically constructed patterns designed to optimally consider the bits being flipped by incoming writes. ABFPL also has a number of ways in which it can be pre-configured to optimize for datasets with certain known characteristics. For example, if it is known that datasets generally only vary the lower 16 bits of a write, ABFPL can be pre-configured to construct patterns that only mask 16 bits. Next we will discuss some of the motivations for creating ABFPL, discuss the design which includes the main features and major pre-configurable parameters, and finally show the potential of using adaptive approaches to improve wear-leveling on SCM by comparing ABFPL to Flip-N-Write.

## 1.7 Contributions

1. We design and implement ABFPL, an adaptive approach to decreasing wear leveling on SCM, which trades additional writes and latency for fewer bits flipped.

2. We provide the results of an evaluation comparing the performance of ABFPL against that of Flip-N-Write in bits flipped, latency, and write amplification.

# Chapter 2

# Motivation

## 2.1 The application knows best

There is a common trend among a large portion of the literature on SCM wear-leveling: ideas are focused on very simple algorithms, often primarily to allow for the design to be implemented at the hardware level. However, Bittman *et al.* demonstrate [5] that a large source of bit-flips originate from writes requested at the application layer. This indicates that the application layer has much more potential for being optimized for specific patterns. In addition, the application has the most context regarding the data that it operates on. As an example, Persistent Memory Logs [30] primarily contain ASCII characters, and an adaptive approach can take advantage of this by optimizing for the datasets that will come from this application.

## 2.2 From prediction to adaption

ABFPL was conceived by thinking about how to mask the bits that are flipped most frequently. While most traditional approaches are simple, some methods like

Captopril [23] take a more predictive approach where they consider patterns that represent some preconceived notion of common bit-flip patterns. However, this isn't sufficient because there are too many different variations of datasets. For example, if a system establishes that 16 particular bits are most likely to be flipped in every dataset, it will not help against a dataset in which a different 16 bits are flipped the most. In contrast to the predictive approach, we decided to calculate how many times each bit-flipped in a *batch*—or a group of writes buffered to DRAM—and determine a pattern that would consider the bits which flipped the most. However, in order to most optimally match patterns to writes, an ordering must be established. Writes initially must be buffered to DRAM while counting the number of times each bit has flipped. Once the batch is completed, a pattern masking the most frequently flipped bits is added to the *pattern table*—or array of patterns. Finally, before each write from a batch is written to SCM, the optimal pattern, often being the one constructed from this batch, is selected. In this way, ABFPL adapts to datasets rather than simply predicting based off a preconceived notion of a common bit-flip pattern.

# Chapter 3

# Background

## 3.1 SCM Challenges

Low write endurance [38] means that it takes SCM fewer writes than DRAM before cells become corrupted and do not consistently work as required.

High write latency and asymmetric read/write latency mean that SCM needs to be utilized differently than DRAM. Normally read and write operations would take the same time on a memory bus, however this would no longer be the case.

There is also a need for stricter consistency requirements since corruptions will persist across reboots.

## 3.2 SCM Special Applications

### 3.2.1 Instant Restart

Since SCM doesn't lose data across power cycles, normal reboots can set the contents of memory to contain a valid system state. For example, memory could be pre-loaded with the contents it would have by the time it normally reaches a

login screen, and it would persist after a shutdown because SCM is non-volatile. In addition, most of the time booting is normally spent copying the memory from storage. With these two important concepts combined, instant reboot [42] becomes possible.

### 3.2.2 Persistent Memory Logging

Persistent Memory Logging (PML) is a system of logging which is performed on SCM rather than storage. This is because PML is often used to log critical system data that would be lost in higher volumes if waiting for IOs to complete. By logging directly to main memory, it is easier to guarantee that data loss is minimized. Finally, PML guarantees that logs are up to date and consistent, which enables fast recovery [1].

## 3.3 SCM wear-leveling

There is some work that focuses on reducing the number of extra writes through a RBW (Read Before Write) mechanism. Flip-N-Write [10] and frequent pattern compression [17] are among those approaches. In Flip-N-Write, they use one overhead bit for each segment to show whether the recorded data is being inverted or not. This design ensures that the number of bit-flips will not exceed $\frac{N}{2}$, where N is the total number of bits in a word (if we do not consider the overhead bits). Frequent pattern compression [17] tries to find some common patterns and then compress data to reduce the number of bit-flips in SCM. Although this approach has succeeded in reducing the number of writes compared to Flip-N-Write, it might lead to premature failure of some specific high-entropy cells [40].

In Captopril [23], the authors show that because there are some specific pat-

terns in written bits, they can reduce the total number of writes through considering some initial patterns to mask hot locations in writes. They determine these hot locations by considering the bit-flip patterns of many different datasets. Their results show some improvement in the number of bits written in NVM. However, as it is clear, this method cannot guarantee that it works for any kind of application and data set. In other words, Captopril's practicality is limited to specific data sets.

Flip-Mirror-Rotate [40] is another system that tries to reduce the number of bit-flips per write. This method takes advantage of two existing methods, Flip-N-Write [10] and Frequent Pattern Compression (FPC) [17], to reduce the number of flipped bits. Again, this method uses only predefined patterns to mask some bits, which cannot guarantee being application agnostic.

MinShift [34] proposes a method to reduce the total number of update bits to SCMs. Essentially, if the Hamming distance falls between two specific bounds, data is rotated until the Hamming distance no longer falls in this range. Although this method is simple, it suffers from high overhead.

In minFS [18], the authors use a combination of MinShift and Flip-N-Write to decrease the number of written bits. They compute the minimum amount of some possible states to choose a pattern to encode the data. This method has advantages and disadvantages of both methods.

The primary way our approach improves over previous ones is through being adaptive. Previous methods would primarily determine what to write either entirely based off the write request or by utilizing some predetermined prediction mechanism. In contrast, ABFPL dynamically saves information about requested writes to adaptively determine the bits that would benefit most from being inverted.

# Chapter 4

# Design

## 4.1 ABFPL Overview

ABFPL adapts to datasets by creating patterns based off the most commonly flipped bits which are tracked in each batch. In addition, ABFPL can be pre-configured with a number of variables that can affect performance, and are described in 4.2.

Write requests go through a batch buffering procedure in ABFPL. First ABFPL initializes a pattern table as described in section 4.3. As writes are requested, they are put into a batch until a pre-configured number of them are gathered as mentioned in section 4.4. At this time, a pattern is created using the *bit-flip array*—an array of bit-flip counters—and persisted to SCM through the method described in section 4.5. Finally, the persisting of writes from a batch to SCM is performed as described in section 4.6. Later patterns can also be deleted as mentioned in section 4.7. This order must be maintained to ensure that patterns are completed before writes are persisted to SCM. This is so that writes can be matched with the patterns constructed from their respective batches.

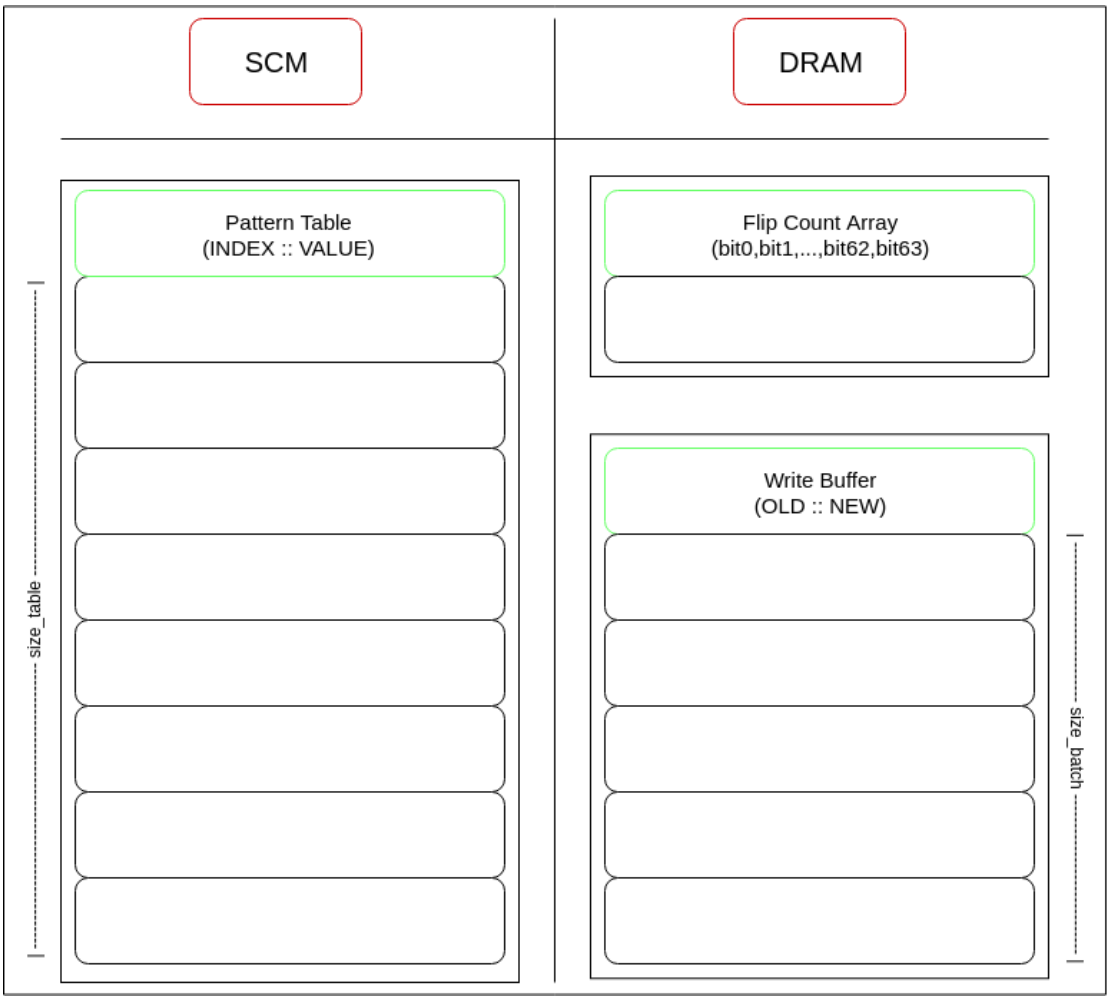


**Figure 4.1:** An overview of the main components of ABFPL and where they reside. In addition, an example with values from our simulation is shown in A.1

ABFPL also has specific requirements for what data must be stored in SCM and what must be done to prevent unnecessary wear. There is exactly one pattern table, batch buffer, and bit-flip array per instance of ABFPL. The pattern table is the only component of ABFPL that needs to reside on SCM. This is because patterns in ABFPL are like keys in cryptography: if they are lost, any associated data cannot be recovered. Figure 4.1 provides a high level illustration of the main components of ABFPL and where they reside in the system.

## 4.2 ABFPL Variables

Firstly, `table_size` refers to the maximum number of patterns in a pattern table. This must be a power of two to be able to index all locations with $\log_2$(`table_size`) bits. The `table_size` also determines the bit-width of the offset that needs to be associated with each write ($\log_2$(`table_size`)).

Another variable, `batch_size` refers to the number of writes that will be buffered to create each pattern. The implication of choosing a particular `batch_size` is a trade-off between the creation of accurate patterns and filling the pattern table too quickly. Smaller values make patterns more representative of individual writes in a batch, whereas larger values will take longer to fill the table.

In addition, `size_mem_region` refers to the portion of the SCM that is associated with a given pattern table. This means that only writes that occur within a particular pattern table's associated region will be buffered in a batch, considered for the creation of a new pattern, and written to SCM with an offset referring to a pattern from this pattern table. Smaller values of `size_mem_region` mean that the table will more likely be able to make accurate patterns, especially if the bit patterns within a region are similar. However, using smaller regions means more tables are required to cover a portion of memory that otherwise could have been associated with a single large table.

Also, `bit_prctl_cutoff` refers to the minimum percentile of bit-flips for a bit to be included in a pattern. For example, median, or a `bit_prctl_cutoff` value of 50, means that a bit has to have at least median bit-flips to be included and set to one in the new pattern.

In addition, `word_bitsize` refers to the number of bits per word. In our preliminary experiments, increasing this results in improving bit-flip performance

from ABFPL. This was kept at 64 bits for compatibility with the largest common data type (unsigned long int, or uint64_t). However, writes can be set to take up entire cachelines by setting `word_bitsize` to 512 bits, or 64 bytes.

Finally, `freq_table_clear` refers to the frequency with which the table's entries are cleared. However, regardless of the setting of `freq_table_clear`, the table can only be cleared if the associated memory has either been cleared or written back to storage.

## 4.3 Initializing ABFPL

In ABFPL, patterns are stored in a pattern table which resides in SCM. This is important because the table is needed to reconstruct the original data: losing it means the data is also lost. This table will take up `table_size` × `word_bitsize` / 8 bytes for all of its entries. The pattern table can be initialized with a number of common patterns, but this may not be as useful as letting it create patterns on its own. However, in our case, the four patterns analogous to Flip-N-Write are added to the top of our ABFPL pattern tables: invert left half (0xFFFFFFFF00000000), invert right half (0xFFFFFFFF), invert all (0xFFFFFFFFFFFFFFFF), and invert none (0x0). This is to ensure that as a baseline we could at least mask the same bit-flips as Flip-N-Write.

In addition, an array of `word_bitsize` counters are needed to track the number of bits flipped in the batch so that a pattern can be made later as described in section 4.5. This bit-flip array has `word_bitsize` entries and is initialized to all zeros. It is stored in DRAM since making it persistent would simply cause unnecessary wear.

## 4.4 Batching writes

In order to create patterns that are representative of the data we are writing, we must use that same data as a reference. Before a write can be performed, a pattern has to be created based off of it. Writes will continue to be buffered until `size_batch` writes have been requested. All writes buffered for this batch are stored in DRAM: this means that having a larger `size_batch` can result in a larger number of writes lost. While buffering writes to DRAM, the bit-flip array is updated. More specifically, if a bit would have been flipped when comparing the write to the value previously in a location, then the associated index in the bit-flip array is incremented by 1.

## 4.5 Constructing Patterns

Patterns are created at the end of a batch using the bit-flip array. First, the batch is sorted by ascending bit-flips, but in an isolated fashion which doesn't change the order that writes will be written from the batch. Then a pattern is constructed such that bits set to one have as many flips as the `bit_prctl_cutoff`$th$ percentile. Thus, this pattern would pre-invert these bits if chosen, by later writing (`pattern` $\oplus$ `new_write`) rather than the original write value. Finally, the bit-flip array is reset so that the next pattern will be created using a bit-flip array solely considering the next batch. A simple example of creating patterns for eight-bit words is provided in Figure 4.2 to illustrate this concept.

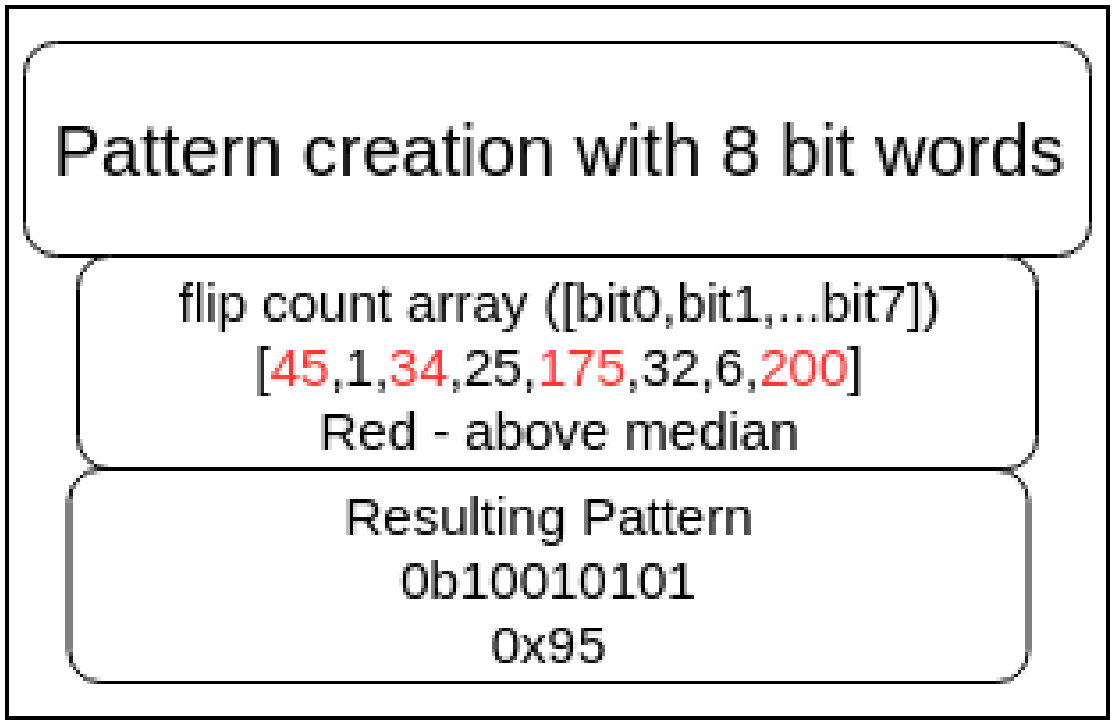


**Figure 4.2:** Example of creating eight-bit patterns from a bit-flip array using a `bit_prctl_cutoff` value of 50.

When a new pattern is requested in this case, the bits that have at least the median number of flips are set to one. All other bits are zero. Note that eight bits are used because illustrating pattern creation becomes easier than with 64 bits which our experiments are based on.

## 4.6 Persisting Writes to SCM

At this point, writes will persist to SCM. For each requested write in the batch, the ideal pattern is found. The Hamming distance of overwriting an offset is also considered. In other words, a pattern is chosen for having a minimized Hamming distance for hdist(`new_value` $\oplus$ `pattern`, `old_value`) + hdist(`new_pattern_offset, old_pattern_offset`). Once the ideal pattern is chosen, two writes are performed: first a write of `new_value`$\oplus$`pattern`,

followed by a second write containing the *pattern offset*—or the index of the pattern in the pattern table—and, if enabled, the valid bit.

## 4.7 Clearing the pattern table

An ABFPL table will eventually fill up and may not always have the best patterns for future writes. Unfortunately, patterns can only be removed if a pattern table's associated memory is also cleared. This is because writes already written to SCM require their original pattern to be recovered. Luckily, there are applications that clear entire portions of their memory periodically. A good example of such an application is Persistent Memory Logging. In persistent memory logging, data is eventually flushed to storage. During this flush, all the memory associated with a given table can be cleared. This means that the table's entries can also be completely cleared at this time.

## 4.8 ABFPL Crash Consistency

ABFPL maintains consistency for its pattern table across crashes before, during, and after each write operation. A write will also maintain consistency if its pattern offset is written with a valid bit. Assuming that the valid bit is enabled for writes, all conditions under each case will apply.

1. In all cases:

    - ABFPL will maintain consistency for all patterns and flushed writes up until the point of a crash, since they are stored in SCM.
    - All writes in the recent unfinished batch as well as the state of the bit-flip array will be lost due to the nature of DRAM losing its data

across power cycles.

2. Before a write request:

   - The write will not be performed.

3. During a write request:

   - This write will not persist. Either it will have been only written partially, if at all, to DRAM, or it may have been written partially to SCM without succeeding to set the valid bit. In either case, the data for this write will not be considered valid upon system restart.
   - This may result in a new pattern being created if this write is the last in the recent batch and the pattern is successfully written to SCM along with corresponding pattern valid bit.
   - If a pattern is constructed and persists to SCM as described in the previous point, some writes in the batch, not including this one, may persist to SCM.

4. After a write request:

   - If this is not the last write in the batch, This write will be added to the batch and be lost, because it was not yet persisted to SCM.
   - If this is the last write in the buffer, a new pattern will be constructed and persist to SCM successfully. In addition, all writes in the recent batch, including this one, will be persisted to SCM.

## 4.9 Conclusion

ABFPL adaptability has the potential to significantly improve bit-flip performance and the flexibility to optimize for datasets with known characteristics. Next we will look at how we setup our evaluation and our results.

# Chapter 5

# Evaluation

## 5.1 Flip-N-Write Implementation

The implementation of Flip-N-Write is very simple. Both a two-bit offset and a 64-bit write are considered when calculating bit-flip overhead. If the left half's Hamming distance is greater than 25% of the word size, the left half is inverted and the left offset bit is set. Similarly, this applies to the right half. A valid bit is not included with writes, since it is not part of the Flip-N-Write spec.

## 5.2 ABFPL Implementation

Pattern table initialization: The pattern table is created with a `size_table` of 256. The `size_mem_region` varies depending on experiment. The patterns are all 64 bits, since `word_bitsize` is also set to 64. The patterns of all zeros, set right half, set left half, and all ones are included. The binary values of these are 0x0, 0xFFFFFFFF, 0xFFFFFFFF00000000, and 0xFFFFFFFFFFFFFFFF. This is so that we include all the cases of Flip-N-Write as a baseline for patterns to mask.

Buffering writes: The write is saved in a vector and all bit-flips are added to the associated indices in the bit-flip array.

Creating new patterns: A pattern add is attempted every time `size_batch` = 100 writes have been requested. The `bit_prctl_cutoff` is set to 50, so every bit that has at least median flips is set in a new pattern. If the pattern doesn't exist and there is still space, it is added. Specifically, it is added to the next available spot in the table that would also result in the lowest Hamming distance. The table may be cleared depending on the experiment, by choosing a value for `freq_table_clear`, so that the table is either never cleared, or cleared soon after being full.

Clearing the table: A pattern table clear simply sets the valid bit of every pattern in the pattern table to zero, thus invalidating each one. In our case, the initial Flip-N-Write patterns are never invalidated since we wanted to always consider those as a baseline. Furthermore, when a pattern is later added, it is always added to the index that would result in the smallest Hamming distance, based off the left over pattern values.

Writing batch to SCM: For every write, the best pattern to decrease bit-flips is determined. That is, a pattern and associated pattern offset such that
hdist(`new_value` $\oplus$ `pattern`, `old_value`) +
hdist(`new_pattern_offset`, `old_pattern_offset`)
is minimized. This also requires reading the old value and old pattern offset in order to determine which pattern is best. A valid bit is not included with writes since Flip-N-Write doesn't include one either. There still are valid bits for pattern table entries because they are part of the design requirements for ABFPL.

## 5.3 Bit-Flips per Write

The calculation for average bits flipped in Flip-N-Write (`fnw_fs`) and ABFPL (`abfpl_fs`) are in equations 5.1a and 5.1c respectively. The Table overhead for ABFPL (`tbl_over`) is defined below based off the table size (`tbl_sz`), the number of tables (`num_tbl`), initial pattern bit-flips (`init_ptrn_fs`), and a running count of bits flipped for all patterns in the table(`all_ptrn_fs`). Where $N$ is the number of actual writes requested, $\texttt{wrt\_fs}_i$ is the number of bit-flips for write $i$, and $\texttt{off\_fs}_i$ is the number of bit-flips for the offset of write $i$, the average bits flipped are:

$$\texttt{fnw\_fs} = \frac{\sum_{i=0}^{N-1}(\texttt{wrt\_fs}_i + \texttt{off\_fs}_i)}{N} \tag{5.1a}$$

$$\texttt{tbl\_over} = \texttt{tbl\_sz} \times \texttt{num\_tbl} + \texttt{init\_ptrn\_fs} + \texttt{all\_ptrn\_fs} \tag{5.1b}$$

$$\texttt{abfpl\_fs} = \frac{\texttt{tbl\_over} + \sum_{i=0}^{N-1}(\texttt{wrt\_fs}_i + \texttt{off\_fs}_i)}{N} \tag{5.1c}$$

Both the ABFPL structure and pointer are counted toward bit-flips. An additional overhead is incurred when creating and deleting patterns. Namely, every time a pattern is created, the number of bits flipped is added to a running pattern bit-flip count (`all_ptrn_fs`). When a clear is performed, every dynamically created entry is invalidated, not including those baseline Flip-N-Write patterns that are initially added. This also adds an additional bit to the running pattern bit-flip count for each invalidation.

## 5.4 Gathering Memory Traces

Intel Pin, a programming instrumentation framework, was used to generate memory traces of the runs for four different programs. These programs from MiBench [22] are Basicmath, Typeset, Patricia, and Stringsearch. They are chosen because they are also used for evaluation in the Flip-N-Write [10] paper. In our experimental setup, the generated traces contained the values that are read/written and their associated memory addresses. As a dataset, this simulates the kinds of writes that traditional memory would experience during the run of an application. Furthermore, we simulated performing these writes while keeping track of bit-flips, latency, and write amplification.

## 5.5 Firmware Update experiment

Similarly to the Flip-N-Write paper's [10] Firmware Update experiment, we simulated a firmware update by overwriting a C executable compiled with -O1 by code compiled with -O3. We did this by writing the former sequentially onto a vector, and then overwriting that vector sequentially with the latter. As a dataset, this is interesting for scenarios where data is being written to an uncached region of SCM. This experiment keeps track of bit-flips, latency, and write amplification. Once again, to compare with Flip-N-Write, these experiments are based off Basicmath, Typeset, Patricia, and Stringsearch from the MiBench [22].

## 5.6 Benchmarking System

The benchmarking system evaluated the number of bits flipped per write request, simulation latency in microseconds per write request, and write amplifica-

tion per write request using each of the methods. For ABFPL, multiple runs are performed with different values for `freq_table_clear` and `size_mem_region`. Our simulation generates CSVs for all metrics using each method. Finally, all result CSVs are analyzed using the Pilot analysis tool [32]. Essentially, this tool provides the mean, variance, confidence intervals, and a number of other useful statistics on a CSV dataset. At this point we just described how the evaluation is setup and next we will describe our results.

# Chapter 6

# Results

## 6.1 Overview

The results comprise the bit-flips, latency, and writes of the naive writing method, Flip-N-Write, and ABFPL. In addition, four parameter variations of ABFPL are used. Refer to table 6.1 for a description of all methods. The bit-flips and writes are tracked throughout the simulation. Latency is determined by timing the functions that perform the associated method's computations, reads, and writes. Note that these latencies are gathered on a machine not equipped with SCM.

| Method | Description |
|---|---|
| Naive | Write data directly to SCM without any wear-leveling method |
| Flip-N-Write | Flip-N-Write algorithm |
| ABFPL_M0_C0 | ABFPL with one table and clear disabled |
| ABFPL_M0_C1 | ABFPL with one table which is cleared soon after the table fills up |
| ABFPL_M1_C0 | ABFPL with a table every 100 kB, but clear is disabled |
| ABFPL_M1_C1 | ABFPL with a table every 100 kB, and clear occurs soon after the table fills up. |

**Table 6.1:** This table gives an overview of what each method does.

In Table 6.1, ABFPL is shown to have two options for the associated memory region: an associated memory region so large that only one table will ever be needed, or a region small enough to result in multiple tables (a `size_mem_region` value of 100 kB). ABFPL also has two options for clear: either clear is disabled or it happens every time the batch associated with the last pattern is done being flushed (a `freq_table_clr` value of `table_size` × `batch_size`).

There are eight datasets in total. Four are the memory traces from the four different micro benchmarks. The other four are the simulation of a "firmware update", which is a dataset that the Flip-N-Write method made use of. Appendix B also provides more information and discussion about these datasets. Finally, ABFPL is configured with a `table_size` of 256, a `batch_size` of 100, a `bit_prctl_cutoff` of 50, and a `word_bitsize` of 64 throughout all experiments. These are chosen as good values over the course of our experiments and are

kept constant in order to emphasize the impact of varying `size_mem_region` and `freq_table_clear` instead.

## 6.2 Bit-flip Results

Bit-flip results are based on experiments using the Basicmath, Typeset, Patricia, and Stringsearch microbenchmarks. Figure 6.1 shows the Trace experiments with these four microbenchmarks, whereas Figure 6.2 shows the Firmware Update experiment using these four microbenchmarks.

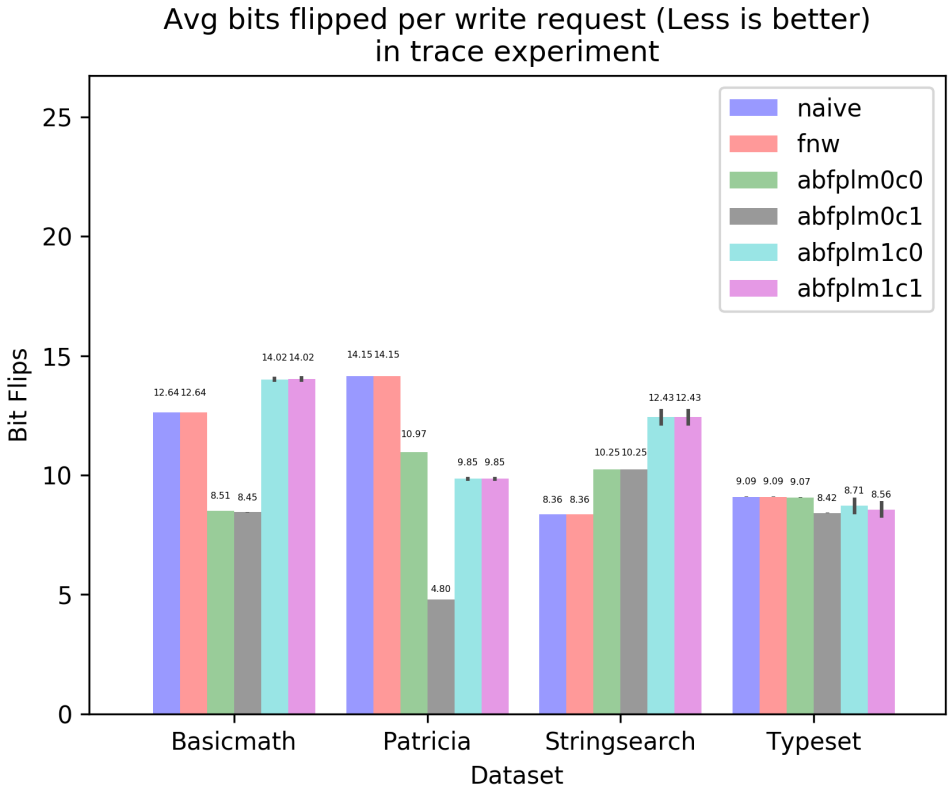


**Figure 6.1:** Shown is the number of bits flipped per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Trace experiment. The values for this graph are listed in Table C.1.

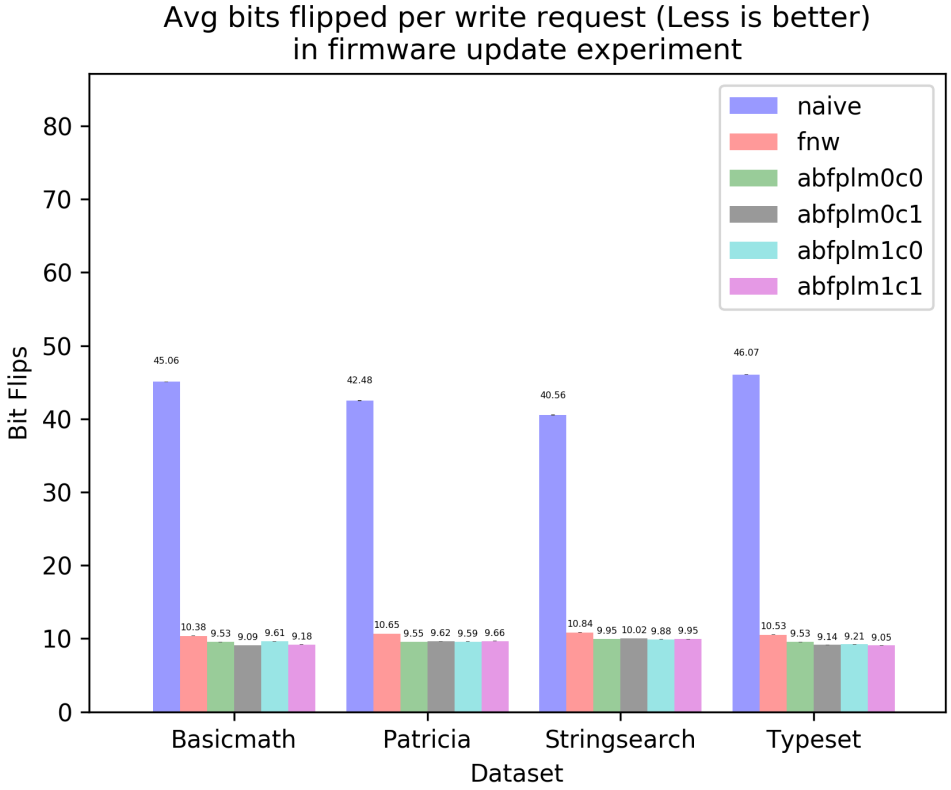


**Figure 6.2:** Shown is the number of bits flipped per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Firmware Update experiment. The values for this graph are listed in Table C.2.

ABFPL outperforms both the naive and Flip-N-Write methods on most datasets in terms of total bits flipped per write request. Since naive just writes directly to memory, it ends up resulting in the most bit-flips. Flip-N-Write is able to optimize by ensuring that no more than 25% of the total bits flip in each half of a word. However, ABFPL outperforms Flip-N-Write in most cases. In particular, the Trace experiment shown in Figure 6.1, illustrates a case where Flip-N-Write barely can make a difference. This is because on average the bit-flips are kept under 25% of the 64-bit word size, meaning neither half can be optimized by Flip-N-Write.

In regards to the overhead associated with creating multiple tables and updating them individually, the performance suffered where there are more tables than necessary. In the Trace experiment show in Figure 6.1, certain addresses are disproportionally written more than others, while some are sparsely written if at all. This means that those areas which are written more frequently could benefit from having more tables, whereas those that are sparsely written could perform better by sharing one big table. On the other hand, in the Firmware Update experiment shown in Figure 6.2, writes are sequential, meaning each address is accessed in order and once, so the usage of memory addresses is uniform.

In contrast to the effects of having multiple tables, the pattern table clear operation generally improved performance. This worked even in the Trace experiment shown in Figure 6.1 because table clear didn't occur for those tables that are sparsely used, and therefore overhead of clearing didn't significantly impact performance in this aspect. On the other hand, when a table clear occurs, new patterns are created for a decreased cost in bit flips, since similar patterns are already written to the table. In addition, the new patterns are more representative of future batches which results in an overall improvement in bit-flips for most cases. One particularly interesting case is the Patricia Trace experiment where we found that the median number of flips was approximately 2× that of the others on average. This means that the Patricia Trace has significantly more writes than the other experiments which could benefit from inverting the bits with more than median flips.

## 6.3 Simulation Latency

Latency results are based on experiments using the Basicmath, Typeset, Patricia, and Stringsearch microbenchmarks. Figure 6.3 shows the Trace experiments

with these four microbenchmarks, whereas Figure 6.4 shows the Firmware Update experiment using these four microbenchmarks.

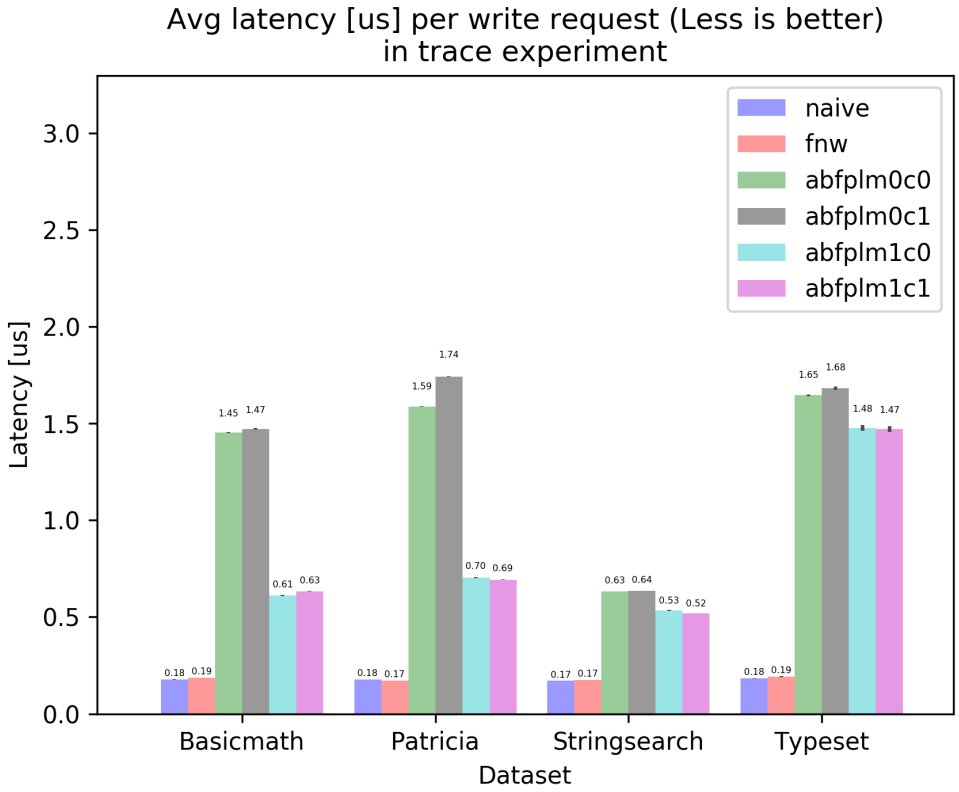


**Figure 6.3:** Shown is the simulation microsecond latency per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Trace experiment. The values for this graph are listed in Table C.3.

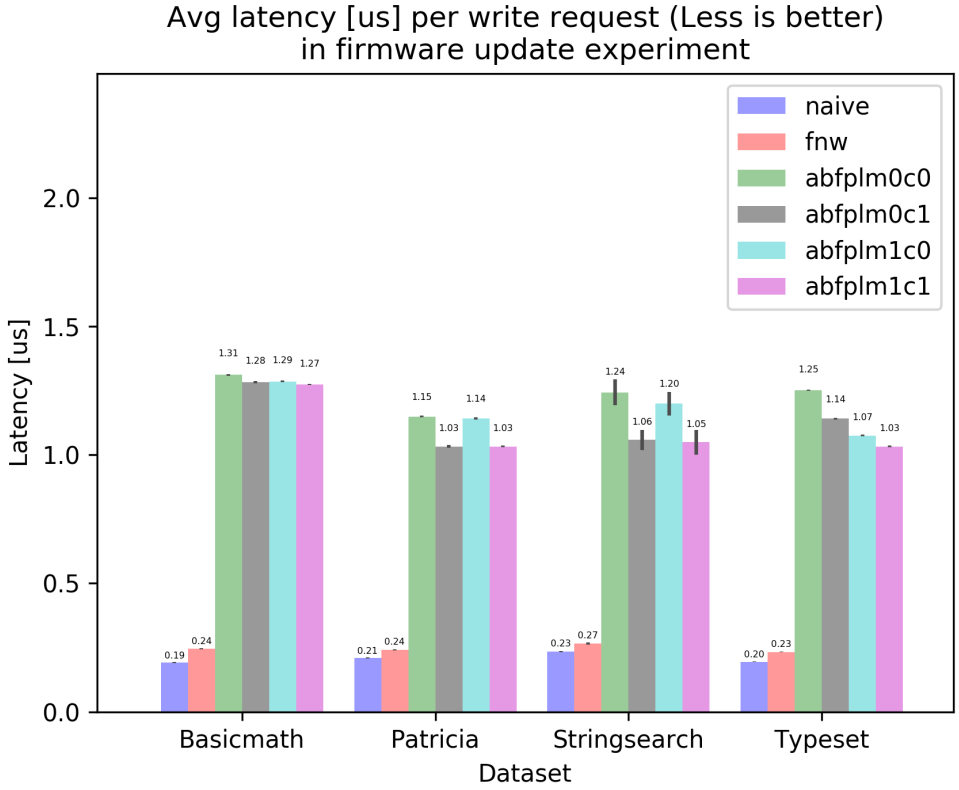


**Figure 6.4:** Shown is the simulation microsecond latency per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Firmware Update experiment. The values for this graph are listed in Table C.4.

ABFPL latencies reach up to 10 times that of the naive or Flip-N-Write methods in our simulations. The Naive method is the fastest because it simply writes the original value without any additional offsets or optimizations. Flip-N-Write is similar since it can compute a value to write entirely based off the requested write. On the other hand, ABFPL has to look through the pattern table to determine which entry to use before actually persisting a write.

ABFPL can be configured to significantly decrease latency. One method to achieve this is to decrease the table_size, which will decrease the maximum number of patterns that will be considered per write. Another method is hinted by

the results. In particular, the Trace experiment shown in Figure 6.3 shows that having more tables can significantly decrease latency. This is because writes are less likely to be associated with a full pattern table, and therefore will on average spend less time considering patterns. In contrast, when there is only one table, it fills up early in an experiment, making every write afterwards require the traversal of the entire table to choose an optimal pattern. In the Firmware Update experiment shown in Figure 6.4 the affects of having multiple tables still results in better performance, but since writes are uniformly distributed to each memory location, it's not as helpful an optimization as for the Trace experiment which is disproportionately written to a sparse set of locations.

Overall, we can conclude that decreasing latency in ABFPL primarily boils down to decreasing the time spent choosing patterns. Finally, ABFPL latency will improve but not likely outperform Flip-N-Write when using SCM because write latencies are significantly longer than read latencies, and ABFPL's largest latency overhead is reading the pattern table.

## 6.4 Write Amplification

Write amplification results are based on experiments using the Basicmath, Typeset, Patricia, and Stringsearch microbenchmarks. Figure 6.5 shows the Trace experiments with these four microbenchmarks, whereas Figure 6.6 shows the firmware update experiment using these four microbenchmarks.

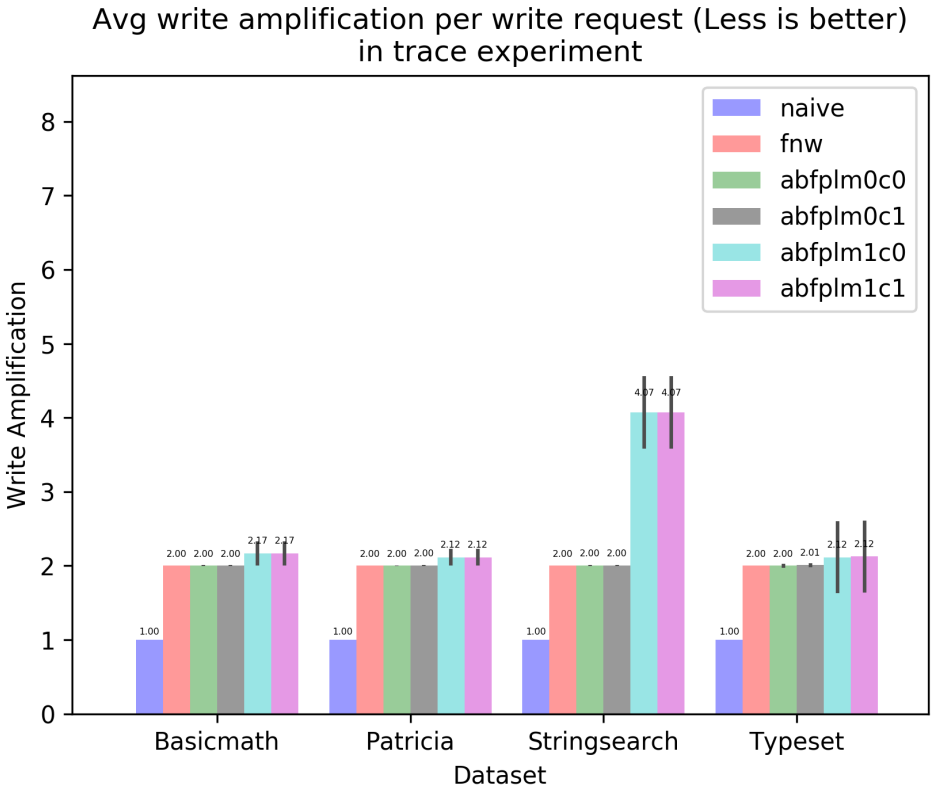


**Figure 6.5:** Shown is the write amplification per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Trace experiment. The values for this graph are listed in Table C.5.

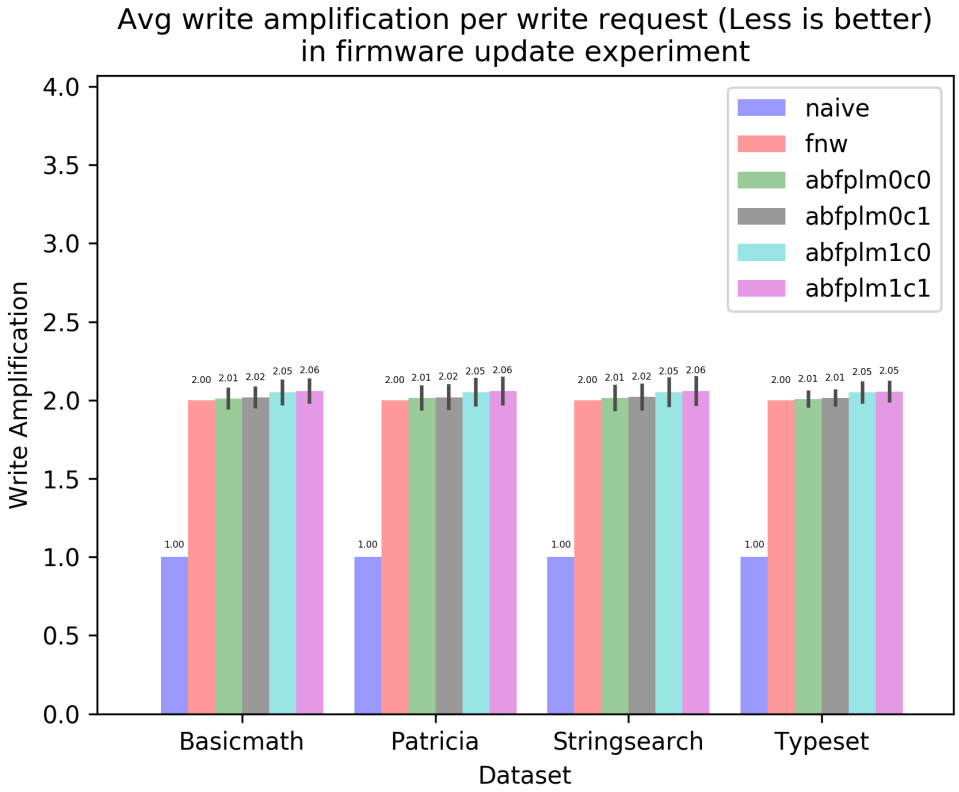


**Figure 6.6:** Shown is the write amplification per write request (including offset) for the naive, Flip-N-Write, and ABFPL benchmarks (less is better) from the Firmware Update experiment. The values for this graph are listed in Table C.6.

ABFPL and Flip-N-Write have about the same write amplification ratio of about 2× that of the naive method in most cases. The naive method performs exactly one write per write request and thus is the baseline with a 1× factor. While Flip-N-Write only does simple computations to determine what to actually write, it still needs a second write to store the offset. On the other hand, despite the writes required to fill the table, the maximum 256 pattern entries are a small fraction of the about 1-30 million line traces and approximately 1-2 MB Firmware Update executables. Appendix B provides additional context, especially for the write amplification results.

One interesting observation is that the additional ABFPL overhead, when

both "clear" and "multiple tables" are enabled, is roughly the sum of the additional overhead from their individual results. For example, the Typeset Trace has 2.010 and 2.115 write amplification respectively for ABFPL_M0_C1 and ABFPL_M1_C0 while ABFPL_M1_C1 is 2.124. This makes sense since $0.010 + 0.115 = 0.125 \approx 0.124$.

Another interesting observation is the variation in error bars. In the Trace experiment shown in Figure 6.5 which accesses a roughly 400 MB range of addresses, approximately 4500 tables are spawned due to the 100 kB associated memory regions. Initializing and filling in these tables results in large write spikes. On the other hand, in the firmware experiments shown in Figure 6.6, only 9 – 21 tables are created, which was small enough that it did not have a significant effect on write amplification in ABFPL.

Finally, another interesting finding is the especially high amplification on the Stringsearch Trace experiment. Despite having the same maximum memory range and the same number of tables as the other Trace experiments, it has roughly $\frac{1}{30}th$ the number of memory traces. This means that the writes incurred for initializing tables has a much larger affect on write amplification compared to the other experiments.

## 6.5 Conclusion

While ABFPL incurred some additional write amplification and significant latency overhead, it is able to decrease bits flips in most of our experiments. In addition, some interesting relevant metrics are also established.

Firstly, a higher number of bit-flips at the `bit_prctl_cutoff`$th$ percentile is correlated with significant performance improvements when paired with frequent table clears.

Next, we observed that in our memory traces, accesses are disproportionate, thus resulting in some locations being written frequently while most are sparsely touched, if at all. Thus, one future optimization is to have tables with different values for `size_mem_region` based on the frequency of writes to a region. In other words, large tables can cover many regions that are sparsely used, and small tables can cover regions of memory that are written frequently.

Finally, the performance could have also improved by varying the `bit_prctl_cutoff` for each group of datasets. This can be seen in the consistency of the results for the naive method in bit-flips. In the Trace experiment shown in Figure 6.1 the number of bits flipped by the naive method was always around 10, whereas in the Firmware Update experiment shown in Figure 6.2 the number of bits flipped by the naive method was always around 43. This means that we may have benefited from setting a `bit_prctl_cutoff` value of ($100 \times 10$ / `word_bitsize`) for the former, and a value of ($100 \times 43$ / `word_bitsize`) for the latter.

In the end, ABFPL is an exploration into the implications of adaptive approaches to wear-leveling and has potential, but plenty of room for improvement.

# Chapter 7

# Conclusion

We provide an implementation of a novel adaptive approach to SCM wear-leveling called Adaptive Bit-Flip Pattern Learning (ABFPL) and benchmark it against Flip-N-Write. Our experiments show that ABFPL is a viable solution for decreasing wear on SCM. ABFPL trades off higher write amplification and latency for up to a 57% improvement over Flip-N-Write in bit-flips.

In addition, future research in adaptive SCM wear-leveling can further identify ways to tune ABFPL and optimize its performance for different datasets. One particularly interesting thing we learned throughout this work is that there is much more depth to dataset adaption than making patterns. For example, the configuration variables we have described in our design came about as a result of observing ABFPL's response to certain workloads and understanding what parts of the design may vary to optimize for those cases. However, even though we have not been able to find definitive ways to configure these variables, we are now able to establish new metrics that have an effect on them such as high bit-flips for a chosen certain percentile or uniformity of writes on all memory locations.

In conclusion, while hardware approaches to SCM wear-leveling may seem more appropriate due to the simplicity of manipulating writes directly to and from SCM, software approaches are worth pursuing due to their potential for adaptability and flexibility.

# Appendix A

# ABFPL components during experiments

Figure A.1 shows the components of ABFPL with real values from our experiments. The numbers used are gathered during execution of the results for the Basicmath Trace experiment with a single table and clear enabled. `table_size` is illustrated as the maximum number of patterns allowed and `batch_size` is shown to be all of the writes in the buffer before they are persisted to SCM. The bit-flip array is shown with the values it has before dynamically creating the first pattern, which can be found as the fourth index in the pattern table.

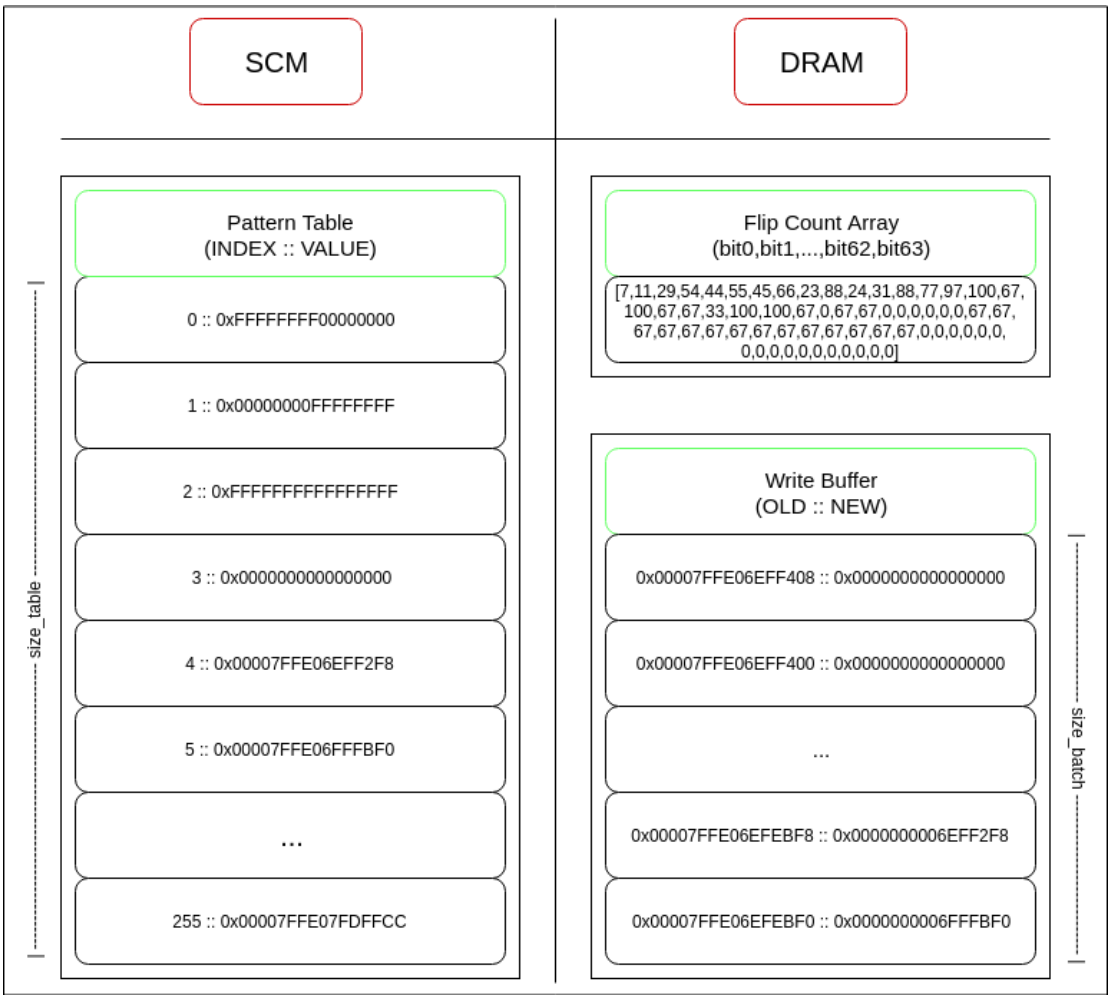


**Figure A.1:** An overview of the main components of ABFPL and where they reside with values from experimental runs

# Appendix B

# Dataset in-depth information

The datasets, their memory access ranges, and sizes are displayed in Table B.1. The memory access range is defined by the largest address accessed from a dataset, whereas the size refers to the number of entries in the dataset.

| Dataset | Memory Access Range | Size |
|---|---|---|
| Trace Basicmath | 454641221 B | 17840335 |
| Trace Patricia | 454641221 B | 26242035 |
| Trace Stringsearch | 454641221 B | 1119875 |
| Trace Typeset | 454641221 B | 28400867 |
| FW Basicmath | 1326928 B | 1326928 |
| FW Patricia | 963064 B | 963064 |
| FW Stringsearch | 875432 B | 875432 |
| FW Typeset | 2055176 B | 2055176 |

**Table B.1:** This table lists each dataset, its memory access range, and its size.

As shown in Table B.1, Traces always ended up accessing the same maximum memory address despite having different numbers of entries, whereas Firmware Updates are sequential, using as many addresses as there are entries in each case. This information is needed to understand some of the performance numbers. Access ranges are used to determine the number of tables that will be created. This can be calculated with $\lceil$(`access_range` / `size_mem_region`)$\rceil$.

We know that in our results, which are introduced in section 6.1, we use a `size_mem_region` value of 100 kB to achieve multiple tables. Thus all Trace ABFPL_M1 experiments have 454641221 B / 100 kB = 4547 tables. On the other hand, the Firmware Update ABFPL_M1 experiments have 14, 10, 9, and 21 tables respectively for Basicmath, Patricia, Stringsearch, and Typeset.

# Appendix C

# Experimental Data

These tables are meant to be used in conjunction with their respective referenced graphs and associated explanations in Chapter 6. Table C.1 shows the complete Trace bit-flip results. Table C.2 shows the complete Firmware Update bit-flip results. Table C.3 shows the complete Trace microsecond latency results. Table C.4 shows the complete Firmware Update microsecond latency results. Table C.5 shows the complete Trace microsecond latency results. Finally, Table C.6 shows the complete Firmware Update microsecond latency results.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 12.637 | 0.006 | 14.150 | 0.005 | 8.364 | 0.047 | 9.092 | 0.007 |
| Flip-N-Write | 12.636 | 0.006 | 14.150 | 0.005 | 8.364 | 0.047 | 9.092 | 0.007 |
| ABFPL_M0_C0 | 8.511 | 0.005 | 10.972 | 0.002 | 10.246 | 0.016 | 9.072 | 0.004 |
| ABFPL_M0_C1 | 8.446 | 0.005 | 4.799 | 0.004 | 10.246 | 0.016 | 8.418 | 0.005 |
| ABFPL_M1_C0 | 14.017 | 0.467 | 9.846 | 0.327 | 12.429 | 5.876 | 8.713 | 0.325 |
| ABFPL_M1_C1 | 14.019 | 0.467 | 9.846 | 0.327 | 12.429 | 5.876 | 8.564 | 0.325 |

**Table C.1:** This table contains the complete bit-flips and associated confidence intervals for the Trace experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.1.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 45.056 | 0.255 | 42.475 | 0.312 | 40.564 | 0.340 | 46.073 | 0.192 |
| Flip-N-Write | 10.380 | 0.092 | 10.654 | 0.112 | 10.842 | 0.120 | 10.533 | 0.073 |
| ABFPL_M0_C0 | 9.531 | 0.093 | 9.553 | 0.104 | 9.945 | 0.112 | 9.532 | 0.740 |
| ABFPL_M0_C1 | 9.086 | 0.084 | 9.620 | 0.107 | 10.024 | 0.115 | 9.136 | 0.068 |
| ABFPL_M1_C0 | 9.609 | 0.155 | 9.587 | 0.160 | 9.879 | 0.163 | 9.213 | 0.139 |
| ABFPL_M1_C1 | 9.183 | 0.150 | 9.657 | 0.162 | 9.946 | 0.165 | 9.052 | 0.138 |

**Table C.2:** This table contains the complete bit-flips and associated confidence intervals for the Firmware Update experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.2.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 0.178 | 0.004 | 0.177 | 0.001 | 0.170 | 0.002 | 0.182 | 0.001 |
| Flip-N-Write | 0.186 | 0.001 | 0.172 | 0.001 | 0.175 | 0.002 | 0.191 | 0.003 |
| ABFPL_M0_C0 | 1.453 | 0.009 | 1.587 | 0.004 | 0.633 | 0.004 | 1.645 | 0.004 |
| ABFPL_M0_C1 | 1.472 | 0.013 | 1.742 | 0.009 | 0.635 | 0.004 | 1.682 | 0.007 |
| ABFPL_M1_C0 | 0.612 | 0.012 | 0.703 | 0.013 | 0.533 | 0.052 | 1.477 | 0.014 |
| ABFPL_M1_C1 | 0.633 | 0.006 | 0.692 | 0.005 | 0.519 | 0.019 | 1.470 | 0.013 |

**Table C.3:** This table contains the complete microsecond latencies and associated confidence intervals for the Trace experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.3.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 0.192 | 0.007 | 0.209 | 0.008 | 0.235 | 0.008 | 0.195 | 0.005 |
| Flip-N-Write | 0.245 | 0.010 | 0.242 | 0.009 | 0.266 | 0.015 | 0.233 | 0.007 |
| ABFPL_M0_C0 | 1.311 | 0.013 | 1.149 | 0.014 | 1.243 | 0.198 | 1.251 | 0.011 |
| ABFPL_M0_C1 | 1.283 | 0.020 | 1.033 | 0.020 | 1.058 | 0.154 | 1.141 | 0.012 |
| ABFPL_M1_C0 | 1.286 | 0.013 | 1.142 | 0.015 | 1.199 | 0.181 | 1.075 | 0.013 |
| ABFPL_M1_C1 | 1.274 | 0.013 | 1.033 | 0.015 | 1.049 | 0.188 | 1.033 | 0.012 |

**Table C.4:** This table contains the complete microsecond latencies and associated confidence intervals for the Firmware Update experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.4.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 1.000 | 0.000 | 1.000 | 0.000 | 1.000 | 0.000 | 1.000 | 0.000 |
| Flip-N-Write | 2.000 | 0.000 | 2.000 | 0.000 | 2.000 | 0.000 | 2.000 | 0.000 |
| ABFPL_M0_C0 | 2.000 | 0.029 | 2.000 | 0.024 | 2.001 | 0.103 | 2.000 | 0.024 |
| ABFPL_M0_C1 | 2.000 | 0.029 | 2.002 | 0.024 | 2.001 | 0.103 | 2.010 | 0.024 |
| ABFPL_M1_C0 | 2.165 | 0.646 | 2.116 | 0.454 | 4.073 | 8.127 | 2.115 | 0.451 |
| ABFPL_M1_C1 | 0.165 | 0.646 | 2.116 | 0.454 | 4.073 | 8.127 | 2.124 | 0.451 |

**Table C.5:** This table contains the complete write amplifications and associated confidence intervals for the Trace experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.5.

| Method | Bm-v | Bm-c | P-v | P-c | Ss-v | Ss-c | Ts-v | Ts-c |
|---|---|---|---|---|---|---|---|---|
| Naive | 1.000 | 0.000 | 1.000 | 0.000 | 1.000 | 0.000 | 1.000 | 0.000 |
| Flip-N-Write | 2.000 | 0.000 | 2.000 | 0.000 | 2.000 | 0.000 | 2.000 | 0.000 |
| ABFPL_M0_C0 | 2.009 | 0.272 | 2.013 | 0.320 | 2.014 | 0.335 | 2.006 | 0.219 |
| ABFPL_M0_C1 | 2.017 | 0.273 | 2.019 | 0.320 | 2.020 | 0.336 | 2.014 | 0.219 |
| ABFPL_M1_C0 | 2.050 | 0.322 | 2.052 | 0.361 | 4.052 | 0.375 | 2.049 | 0.275 |
| ABFPL_M1_C1 | 0.058 | 0.322 | 2.057 | 0.362 | 4.058 | 0.375 | 2.054 | 0.276 |

**Table C.6:** This table contains the complete write amplifications and associated confidence intervals for the Firmware Update experiment. Bm is Basicmath, P is Patricia, Ss is Stringsearch, and Ts is Typeset. Furthermore, -v is the value for the experiment and -c is the confidence interval. This data is illustrated in Figure 6.6.

# Bibliography


[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[2] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting (IEDM), 2004. IEEE International*, pages 587–590. IEEE, 2004.

[3] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, June 1981.

[4] Daniel Bittman, Matthew Bryson, Yuanjiang Ni, Arjun Govindjee, Isaak Cherdak, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: An operating system for next-generation memory hierarchies. Technical Report UCSC-SSRC-17-01, University of California, Santa Cruz, December 2017.

[5] Daniel Bittman, Mathew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. Designing data structures to minimize bit flips on nvm. In *The 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, August 2018.

[6] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, July 2008.

[7] Erin Carson, James Demmel, Laura Grigori, Nick Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. *Write-Avoiding Algorithms*. Berkeley, CA, 2015.

[8] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *5th Biennial Conference on Innovative Data Systems Research (CIDR 2011), Conference Proceedings*, pages 21–31, 04 2011.

[9] Ping Chi, Wang-Chien Lee, and Yuan Xie. Adapting b+tree for emerging nonvolatile memory-based main memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1461–1474, 2016.

[10] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–357. ACM, 2009.

[11] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[12] Chanwoo Chung, Jinhyung Koo, Arvind, and Sungjin Lee. Lightweight kv-based distributed store for datacenters. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, (SPAA '18), pages 271–282, New York, NY, USA, 2018. ACM.

[14] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review*, 49(2):18–26, 2016.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, (SOSP '07), pages 205–220, New York, NY, USA, 2007. ACM.

[16] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[17] David B Dgien, Poovaiah M Palangappa, Nathan A Hunter, Jiayin Li, and Kartik Mohanram. Compression architecture for bit-write reduction in non-volatile memory technologies. In *Proceedings of the 2014 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 51–56. ACM, 2014.

[18] Wei Dong, Xin Li, Yanbin Li, Meikang Qiu, Lei Dou, Lei Ju, and Zhiping Jia. Minimizing update bits of nvm-based main memory using bit flipping and cyclic shifting. In *2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, pages 290–295. IEEE, 2015.

[19] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 371–384, 2013.

[20] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[21] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, December 2001.

[23] Majid Jalili and Hamid Sarbazi-Azad. Captopril: Reducing the pressure of bit flips on hot locations in non-volatile main memories. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1116–1119. IEEE, 2016.

[24] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. Enabling nvme WRR support in linux block layer. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[25] Saeed Kargar and Leyli Mohammad-Khanli. Fractal: An advanced multidimensional range query lookup protocol on nested rings for distributed systems. *Journal of Network and Computer Applications*, 87:147–168, 2017.

[26] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing array of ssds when the storage device is no longer the performance bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[27] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, April 2013.

[28] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 257–270, Santa Clara, CA, 2017. USENIX Association.

[30] Mengjie Li, Matheus Ogleari, and Jishen Zhao. Logging in persistent memory: To cache, or not to cache? In *Proceedings of the International Symposium on Memory Systems*, (MEMSYS '17), pages 177–179, New York, NY, USA, 2017. ACM.

[31] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 476–488. ACM, 2015.

[32] Yan Li, Yash Gupta, Ethan L. Miller, and Darrell D. E. Long. Pilot: A framework that understands how to do performance benchmarks the right way. In *Proceedings of the 24th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2016)*, September 2016.

[33] H. Liu, L. Huang, Y. Zhu, and Y. Shen. Librekv: A persistent in-memory key-value store. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2017.

[34] Xianlu Luo, D. Liu, Kan Zhong, Dan Zhang, Yi Lin, Jie Dai, and Weichen Liu. Enhancing lifetime of nvm-based main memory with bit shifting and flipping. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–7, Aug 2014.

[35] R. Maddah, S. M. Seyedzadeh, and R. Melhem. Cafo: Cost aware flip optimization for asymmetric memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 320–330, Feb 2015.

[36] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[37] Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[38] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[39] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[40] Poovaiah M Palangappa and Kartik Mohanram. Flip-mirror-rotate: An architecture for bit-write reduction and wear-leveling in non-volatile memories. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 221–224. ACM, 2015.

[41] Dusan Ramljak and Krishna Kant. Belief-based storage systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[42] D. Schwalb, M. Faust, M. Dreseler, P. Flemming, and H. Plattner. Leveraging non-volatile memory for instant restarts of in-memory database systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1386–1389, May 2016.

[43] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, December 1999.

[44] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[45] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. *Proceedings of the Sixteenth International*

*Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 91–104, 2011.

[46] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers*, 28(1):44–51, 2011.

[47] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[48] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, (SOSP '17), pages 478–496, New York, NY, USA, 2017. ACM.

[49] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3):209–220, 2014.

[50] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[51] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH computer architecture news*, volume 37, pages 14–23. ACM, 2009.

[52] P. Zuo and Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel & Distributed Systems*, 29(5):985–998, May 2018.