UNIVERSITY OF CALIFORNIA

Santa Barbara

Improving Non-Volatile Memory Lifetime through Temporal Wear-Limiting

A Thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Electrical and Computer Engineering

by

Brian Kenneth Neely

Committee in charge:

Professor Frederic Chong, Chair

Professor Dmitri Strukov

Dr. Diana Franklin, Lecturer

June 2014

The thesis of Brian Kenneth Neely is approved.

Diana Franklin

Dmitri Strukov

Frederic Chong, Committee Chair

June 2014

ABSTRACT

Improving Non-Volatile Memory Lifetime through Temporal Wear-Limiting

by

Brian Kenneth Neely

Non-volatile memory technologies provide a low-power, high-density alternative to traditional DRAM main memories, yet all suffer from some degree of limited write endurance. The non-uniformity of write traffic exacerbates this limited endurance, causing write-induced wear to concentrate on a few specific lines. *Wear-leveling* attempts to mitigate this issue by distributing write-induced wear uniformly across the memory. Orthogonally, *wear-limiting* attempts to increase memory lifetime by directly reducing wear. In this paper, we present the concept of *temporal wear-limiting*, in which we exploit the trade-off between write latency and memory lifetime. Using a history of the slack between per-bank write operations, we predict future write latency, allowing for up to a 1.5x memory lifetime improvement. We present two extensions for improving the effectiveness of this history-based mechanism: a method for dynamically determining the optimum history size, and a method for increasing lifetime improvement through address prediction.

Contents

Chapter 1

Introduction

The proliferation of chip-multiprocessors places an increasing burden on traditional memory systems. Increasing the number of concurrently executing processes raises the number of working sets that must be maintained in memory, thus increasing memory pressure and causing performance degradation [1]. All of these factors accelerate the demand for faster, denser main memories. The rise of energy-sensitive devices such as smartphones compounds the problem, requiring memories that are not only fast and dense, but also low-energy. These factors–speed, density, and energy–are accelerating the search for new memory technologies that exceed the capabilities of traditional DRAM.

Several non-volatile memory technologies exist as promising candidates for a main memory replacement. Two of these, Phase Change Memory (PCM) and Redox Memory (Memristors) are poised to meet or exceed the capabilities of DRAM within the next 12 years. Table 1.1 shows selected data from the 2013

International Technology Roadmap for Semiconductors (ITRS). The 2026 projections for PCM show a write energy that is half that of DRAM and a cell area that is roughly 80% that of DRAM. The best projections for Redox, an emerging technology, show a write energy that is 20x less than that of DRAM and a cell area that is one third that of DRAM. While PCM suffers from being 5x slower than DRAM, Redox memories are anticipated to be 10x faster.

ITRS Parameter	DRAM (2026)	PCM (2026)	Redox (Best Projected)
Cell Area (nm^2)	324	256	100
Write Time (ns)	<10	<50	<1
Write Cycles	>1e16	1e9	>1e11
Write/Read Voltage (V)	1.5 / 1.5	<3 / <1	<0.5 / <0.2
Write Energy (J/bit)	2e-15	1e-15	1e-16

Table 1.1: Select ITRS data for DRAM, PCM, and Redox technologies [2]

While non-volatile technologies are projected to meet or exceed the speed, density, and low-energy capabilities of DRAM, all fall short in one category: write endurance. PCM has several orders of magnitude less endurance than DRAM (see Table 1.1), as does Redox memory. Under ideal conditions, such limited write endurance is insignificant. For our baseline system in Section 3, an 8GB PCM memory with 1e9 write endurance per cell has an expected lifetime of roughly 60 years, assuming 4GBps write traffic and uniform writes to each cell. However most systems exhibit *non*-uniform write behavior, which significantly reduces memory lifetime by concentrating writes to specific regions of the memory. Once a cell reaches its endurance limit, it may fail to change state. This potentially results

in data errors, making write endurance a serious challenge for architecting non-volatile memory systems.

To combat this non-uniformity issue, most systems employ some form of *wear-leveling*. Wear-leveling attempts to make non-uniform write traffic uniform by remapping heavily written lines to less frequently written lines. Various schemes exist. Table-based methods, used primarily in Flash-based Solid State Drives (SSDs), track write counts associated with each line and remap lines to achieve uniform wear-out. Shift-based methods use an algebraic relationship to map logical addresses to physical addresses. This results in a significantly smaller overhead compared to the table-based approaches. Start-Gap [3], a novel shift-based approach, is able to achieve a memory lifetime that is 95% of ideal with only 8 bytes of storage overhead. Other wear-leveling schemes exist, such as randomizing address mappings within a bank [4], shifting cache lines with a page [5], and shifting bits in a line, or lines in a segment [6]. All follow a similar theme: each takes a fixed amount of wear and distributes it uniformly across the entire memory.

An orthogonal concept to wear-leveling is *wear-limiting*. Wear-limiting attempts to reduce–rather than distribute–the amount of wear on the memory system. Well-known techniques such as DRAM buffering [5] can be used to this effect. Others, such as Flip-N-Write [7], exploit specific properties of the data being written to reduce the number of writes on a per-cell basis. Error Correcting Pointers [8] uses in-memory pointer indirection to correct failed cells, removing the extra wear induced by Error Correcting Codes (ECC). All such techniques can

be classified as *physical* techniques, in that they alter the actual contents of the memory in order to reduce wear.

Largely unexplored, however, is the use of *temporal* wear-limiting. By increasing the duration of the write pulse to a PCM cell (or other non-volatile memory technology), the write voltage can be reduced proportionally. This decrease in write voltage results in an exponential reduction in the wear caused by the write operation. Thus by lengthening or shortening the write pulse (and proportionally, the write voltage), we can directly control the amount of wear to the memory cell. As this is done temporally, there is no spatial overhead–no encoding of data, no pointer indirection, no additional buffering.

It is well known that several design trade-offs exist in the development of non-volatile technologies. The authors in [9] show that by relaxing the retention time requirement of STT-RAM, latency and energy can be reduced. Temporal wear-limiting provides yet another design knob for non-volatile memories. The ability to vary the write voltage and latency at run-time affords flexibility in deciding when lifetime matters, and when performance matters. Energy, performance, and lifetime are no longer part of a fixed relationship created at design time. Rather, the relationship is determined dynamically, on a system-by-system basis. This dynamic relationship greatly widens the design space, allowing a larger set of memory technologies to be both high performance *and* long lifetime.

In this paper, we show that using temporal wear-limiting can improve memory lifetime by a factor of 1.5x. We show that system performance is not immune to arbitrary memory write latencies, however, and that care must be taken in deciding

when, and how slow, to perform these writes. We present a novel technique for determining this, based on the slack between previous per-bank write operations. This paper is divided as follows: Section 2 describes our physical model for the write latency / lifetime trade-off, as well as our calculation for write slack. Section 3 describes our experimental methodology, including our simulation setup and figure of merit. Section 4 introduces and analyzes our temporal wear-limiting mechanism. Section 5 details a bank configuration sensitivity analysis for our mechanism, and Section 6 outlines future work in this area. Section 7 concludes.

Chapter 2

Background

2.1 Latency/Lifetime Trade-off

Non-volatile memory devices require strong nonlinearity in switching kinetics with respect to the applied write stimulus (typically a voltage). This is necessary to achieve long retention time, while allowing for small write latency [10]. For example, the retention time / write latency ratio is more than 10 orders of magnitude for NOR/NAND flash memories (e.g. 10 years of retention with microsecond/millisecond write time), while it is less than 6 orders of magnitude for DRAM (e.g. tens of milliseconds of retention with tens of nanoseconds write time) [2]. Given this strong nonlinearity, a natural way to decrease write latency in non-volatile memories is to apply a larger stimulus (voltage). However this stimulus is harmful for device lifetime (endurance) as it enhances the same mechanisms that lead to device failure.

We argue that this write latency / lifetime trade-off holds for all non-volatile technologies, both emerging and established, because a similar failure mechanism can be found in all. For flash memory, [11] shows that increasing write (program/erase) voltage exponentially increases the probability of creating and/or filling existing deep traps–a primary source for limited lifetime in these memories. In phase change memories, switching speed is exponentially dependent upon the applied electric field and temperature [12]. This is similarly true for magnetoresistive, ferroelectric, and electrochemical/thermochemical (i.e. memristor) memories, due to the thermally-activated nature of their switching operation. Studies into the failure mechanisms of memristive devices have thus far been limited, however preliminary evidence shows that it is related to excessive stressing of the device upon each write operation [10]. High temperatures combined with a high electric field cause melting of the electrode material, causing permanent failure. It is reasonable to assume that this thermally-activated failure mechanism will be found in all technologies with thermally-activated switching mechanisms. Thus it follows that device lifetime will decrease exponentially with increased write stimulus.

As there are no published results describing the write latency / lifetime trade-off (to the best of our knowledge), we assume a model based loosely on the model presented in [13]. This model is derived for metal oxide memristive devices, but should be applicable to different non-volatile memory technologies because of the similar physics of their failure mechanisms. For example, it predicts the quantitatively experimentally observed trade-off in floating gate memories reported in

[14]. Assuming a thermally activated mechanism with activation energy U_S, temperature T, and constant electric field, switching speed can be approximated as:

$$t_S \approx \frac{d}{v_s} \approx \frac{2d}{fa} e^{\frac{U_S}{kT}} e^{-\frac{Vq}{2kT}\frac{a}{d}}$$

where v_s is the speed of ions, k is the Boltzmann constant, q is an elementary charge, f is the frequency of escape attempts, and a is an average hopping distance [13]. Temperature T is linearly proportional to the dissipated power. The failure mechanism is assumed to be caused by the thermally activated motion of ions across the same distance d but with higher activation energy U_F, such that the average time to failure of the device is:

$$t_F \approx \frac{d}{v_F} \approx \frac{2d}{fa} e^{\frac{U_F}{kT}} e^{-\frac{Vq}{2kT}\frac{a}{d}}$$

Using the above equations, lifetime L (the number of times a device can be switched before failure) is proportional to the ratio of failure time to switching time:

$$L \approx \frac{t_F}{t_S}$$

Calculating this write latency / lifetime trade-off for various temperature-voltage dependencies shows that the trade-off is mostly dependent upon the magnitude of the $\frac{U_F}{U_S}$ ratio (i.e. it is insensitive to any particular law of temperature-voltage dependence). For $\frac{U_F}{U_S} = 2$, latency decrease is linearly proportional to lifetime

decrease. The relationship is approximately quadratic and cubic for $\frac{U_F}{U_S} = 3$ and $\frac{U_F}{U_S} = 4$, respectively. Therefore, for a relatively small electric field ($Vqa << 2dU_S$), lifetime is approximated as:

$$L \approx (\frac{t_S}{t_0})^{(\frac{U_F}{U_S}-1)}, \text{ where } t_0 = \frac{2d}{fa}$$

In this study, we chose to model a quadratic write latency / lifetime trade-off. For consistency with JEDEC timing symbols, we define lifetime in terms of the write pulse width, tWP (note that this is equivalent to the switching time t_S). Thus our lifetime model becomes:

$$L \approx (\frac{tWP}{tWP_0})^2$$

2.2 Performance of Reads and Writes

The central concept of temporal wear-limiting is maximizing the latency of writes without impacting performance. Doing so requires understanding how write latency impacts CPU performance, and therefore requires an understanding of the memory system organization itself. Figure 2.1 gives an overview of a basic memory system, containing one read buffer, one write buffer, one controller, two ranks, and two banks per rank. Memory requests from the CPU are placed in the read or write buffer, depending on the memory operation. The controller selects one command from one of the queues per cycle. This selection is dependent upon the

memory controller scheduling algorithm. In this study we use a First-Ready, First-Come, First-Serve algorithm, in which the controller prioritizes commands that can issue immediately. The memory controller dequeues the selected command and sends the appropriate bank the series of commands required to complete the memory access. This series is dependent upon the active row in the bank. Once a bank is issued a series of commands, it cannot accept any new commands until the series completes.

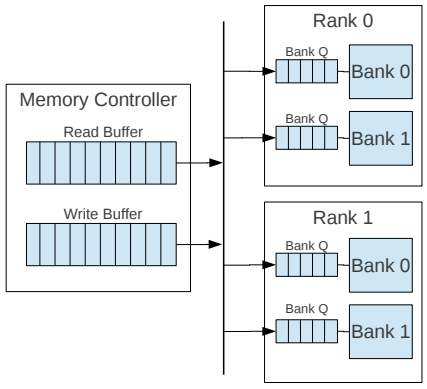


Figure 2.1: A simple memory system

Bank reads and writes are performed by first issuing an *activate* command, which reads the contents of a bank row into set of sense amplifiers, also known as a *row buffer*. Reads simply read the contents of this buffer, while writes overwrite its contents and (in a write-through scheme) write the data to the memory array. To complete a read/write command a *precharge* command is issued, which clears the sense amplifiers and readies them to read the another row. This is known as *closing* the row. Reads and writes to different rows must follow this Activate → Read/Write → Precharge sequence. However, reads and writes to the same

row can operate repeatedly on the row buffer, omitting the Activate and Precharge commands. This optimization for row buffer hits is known as the *open page policy*, as the memory controller does not close a row until it is certain it must activate another.

Reads and writes have different effects on CPU performance. A CPU cannot retire a read instruction until the data is actually read into a register (from cache or from memory). Any time elapsed from when the CPU requests the data and when it receives it is a performance penalty. Unlike reads, writes are not on the critical path for retiring a CPU instruction. Once a write is received by the memory (or cache), the instruction is retired. At this point, the write is fulfilled from the perspective of the CPU. Writes can only stall the CPU if the memory cannot accept the command, i.e. the write buffer is full. Thus, a temporal wear-limiting scheme must not cause additional write buffer stalls, as this impacts write performance.

More subtle, however, is the impact of writes on read performance. As mentioned above, one memory command is issued to and processed by each bank at a time. A write can stall a read if a write occupies the bank at the time when a read is requested on that bank. Writes also stall writes in this manner, and both are known as *bank conflicts*. Bank conflicts are a precursor to buffer stalls. Reads or writes that cannot be issued cannot be removed from the queue, causing the queue to fill. Thus, any temporal scheme that attempts to minimize write buffer stalls must first minimize bank conflicts.

2.3 Exploiting Write Slack

The measure of a good temporal wear-limiting scheme is its ability to maximize write latency without causing bank conflicts. This implies that it must quantify the amount of time available between a write operation on a bank and the next operation (write *or* read) scheduled for that bank: the *slack* between requests. Slack is simple to calculate:

$$Slack = T_{next} - (T_{current} + T_{latency})$$

where $T_{current}$ is the bank issue cycle of the current write command, T_{next} is the issue cycle of the next command, and $T_{latency}$ is the latency of the write command.

$T_{latency}$ is equally simple to calculate. It depends on both the current bank state and the next command issued. If the write command is a row buffer hit, $T_{latency}$ is simply T_{write}: the latency of the write command. However if the command is a row buffer miss, $T_{latency} = (T_{precharge} + T_{activate} + T_{write})$: the time to precharge the previous row, activate the new one, and write the data. The latency of T_{write} is dependent upon the next command issued to the bank. Table 2.1 details the timing values used in our simulations. These values were obtained from [15].

The difficulty in calculating write slack lies in determining T_{next}. It must be calculated at time $T_{current}$ but is only known for certain at time T_{next}. For a temporal wear-limiting scheme to be perfect, it must know T_{next} at $T_{current}$, or predict it with perfect accuracy.

Parameter	Next Cmd	Timing Value
$T_{precharge}$	N/A	T_{RP}
$T_{activate}$	N/A	$T_{RCD} - T_{AL}$
T_{write}	Read	$T_{CWD} + T_{BURST} + T_{WTR} + T_{WP}$
T_{write}	Write	$MAX(T_{BURST}, T_{CCD}) + T_{WP}$
T_{write}	Precharge	$T_{WR} + T_{AL} + T_{CWD} + T_{BURST} + T_{WP}$

Table 2.1: Memory simulation timing parameters

Chapter 3

Experimental Methodology

3.1 Configuration

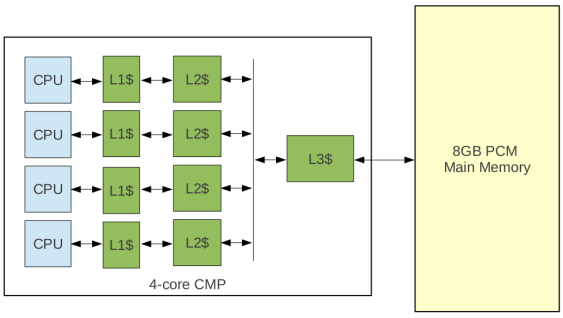


Figure 3.1: Baseline system

Figure 3.1 gives an overview of our baseline system. We model a four-core Intel Core i7 system, the details of which are shown in Table 3.1. An aggressive 8-issue out-of-order CPU model is used, as we wish to eliminate CPU bandwidth

from affecting our study. Each core contains a private L1 and L2 cache, and all share an 8MB L3 cache. The memory system is an 8GB PCM with one channel, four ranks per channel, and four banks per rank. A single channel (single memory controller) is used in order to maximize memory pressure. The controller uses a First-Ready First-Come First-Serve scheduling policy with a 32-entry write buffer, and an open-page row-buffer management policy. Read bank hits are optimized by servicing them immediately from the bank queue, if possible. For simulation we use the gem5 simulator [16] in conjunction with NVMain [15], a timing-accurate simulator for non-volatile memory technologies.

Target System	Intel Core i7
CPU	4-core OoO CMP, 2 GHz, 8-issue, Alpha ISA
L1 Cache (I/D)	(32kB / 4-way) / (32kB / 8-way), private
L2 Cache	256kB / 8-way, private
L3 Cache	8MB / 16-way, shared
Main Memory	8GB PCM Channels: 1 Ranks: 4 per channel Banks: 4 per rank Controller: FRFCFS Scheduler Write Buffer: 32-entry, 64B entries

Table 3.1: Baseline configuration

3.2 Workloads

Table 3.2 outlines the set of benchmarks used in this paper, all of which are memory-intensive applications. The set of benchmarks from the SPEC CPU2006 benchmark suite (astar, GemsFDTD, lbm, milc) are single-threaded applications

which we execute on a single-core simulation with a 2MB L3 cache. The Vips benchmark from PARSEC is a 16-thread image processing pipeline. GUPS (Giga-updates per second) executes updates to random 64-byte words across a 4GB memory region. The SPEC benchmarks and GUPS are simulated for 8 billion CPU cycles (4 seconds) with a 2-second cache warm-up period. Vips is simulated for 8 billion cycles, starting from the benchmark region of interest (ROI), as outlined in [17].

Name	Package	Category
astar	SPEC CPU2006	2D path-finding
GemsFDTD	SPEC CPU2006	Comp. Electromagnetics
lbm	SPEC CPU2006	Comp. Fluid Dynamics
milc	SPEC CPU2006	Quantum Chromodynamics
vips	PARSEC	Image processing pipeline
GUPS	GUPS [18]	N/A

Table 3.2: Workloads

3.3 Figure of Merit

A figure of merit for any wear-limiting scheme is memory lifetime. Temporal wear-limiting requires a second metric, however, as it can impact system performance through its ability to increase or decrease write latency. We therefore use two metrics in this paper: memory lifetime and Instructions per Cycle (IPC).

To obtain IPC, we use the values reported by our simulator. For simulations involving multiple CPUs, we use combined IPC (total number of instructions for

all cores, total number of cycles for all cores).

In this study we define memory lifetime as the number of iterations of a single benchmark if the benchmark were to execute repeatedly until memory failure. Memory lifetime is therefore proportional to $\frac{1}{wear_T}$, where $wear_T$ is the total amount of wear induced on the memory system by write operations during the benchmark's execution. From the lifetime equation given in Section 2.1, it follows that the wear induced by an individual write, $wear_i$, at a constant write pulse tWP_i is equal to $\frac{1}{L}$, or equivalently

$$wear_i = \frac{1}{(\frac{tWP_i}{tWP_0})^2}$$

To broaden the applicability our study we choose not to define wear and lifetime in terms of a specific technology. Instead we define it generically as *unit wear*, in which a write performed with the minimum write pulse width tWP_{min} induces 1 unit of wear. To obtain *unit wear* for a given write pulse width tWP_i we normalize tWP_i to tWP_{min}, resulting in the wear equation:

$$wear_i = \frac{1}{tWP_{Ni}}^2$$

where $tWP_{Ni} = \frac{tWP_i}{tWP_{min}}$, the normalized write pulse width for width tWP_i. The total unit wear induced for a benchmark is the sum of the unit wear induced by individual write operations:

$$wear_T = \sum_{i=1}^{tWP_{Nmax}} \frac{1}{tWP_{Ni}}^2 Q_i$$

where Q_i is the quantity of writes performed at a specified normalized write pulse tWP_{Ni}, and tWP_{Nmax} is the maximum normalized write pulse. Therefore, the lifetime metric for a given benchmark can be described as:

$$Lifetime \propto \frac{1}{\sum_{i=1}^{tWP_{Nmax}} (\frac{1}{tWP_{Ni}}^2 Q_i)}$$

To characterize the performance and lifetime of our various implementations, we compare them against an oracle. Our oracle captures the amount of *slack* available to write operations in each benchmark. We define slack as the time between memory requests at the bank-level. Because the oracle only records timing information, it cannot not affect the timing–and scheduling–of memory operations. It therefore is an indicator of the best memory lifetime possible without affecting performance. In this paper we list our results as fractional lifetime and fractional performance, compared against the oracle.

The slack values our oracle reports are often large to be used as write pulse widths (e.g. 10^5x tWP_{min}). Therefore, we cap the maximum write pulse width, tWP_{max}, at 100x tWP_{min}. $Slack_{max}$ is therefore capped at $tWP_{max} - tWP_{min}$. For our simulation configuration $tWP_{min} = 60$ cycles and therefore $tWP_{max} = 6000$ cycles.

For equal comparison of different implementations, our lifetime metric requires the number of write operations performed in each iteration of a benchmark

to be the same as that of the oracle. Because we modify the latency of write operations, it is possible for a given simulation to execute less write operations in the allotted simulation time. To account for this, we assign the minimum write pulse–and maximum wear–to any un-issued writes, and then include this value in the lifetime calculation.

Chapter 4

Temporal Wear-Limiting

4.1 Perfect Wear-Limiting with an Oracle

Our oracle allows us to simulate a perfect wear-limiting scheme. It does so by calculating write slack in hindsight: it observes $T_{current}$ at time T_{next} and deduces the slack that *could* have been exploited by the write at $T_{current}$. It is important to note that this observes the slack in the system as-is and performs no optimizations to increase slack. Therefore, it serves as a baseline for any wear-limiting scheme, not as an indication of the absolute maximum lifetime.

Table 4.1 shows the lifetime improvement the oracle achieves for the six benchmarks we simulate. The range of improvements is wide: from 1.02x to 7.37x. These values are a direct indication of the amount of write slack available in each benchmark, and therefore the amount of lifetime improvement available to any wear-limiting scheme we develop. It is important to understand not only

the amount of lifetime improvement available with the oracle, but also how this lifetime is attained. Figure 4.1 shows the cumulative percentage of the oracle's lifetime contributed by each write pulse width. These distributions highlight the write pulse widths that have the greatest impact on the oracle's lifetime. Again, we see that the results vary. For benchmarks like GemsFDTD, milc, and vips, the contributions are spread across the entire pulse width range. For others, a small set contributes a majority of the lifetime. In astar, for example, the maximum write pulse width contributes roughly 40% of the lifetime. This data gives insight into the requirements of any wear-limiting scheme that approximates the oracle. It shows which oracle write pulse widths a scheme must accurately predict in order to achieve the largest fraction of the oracle's lifetime. For benchmarks such as astar, it must predict the maximum width accurately. For benchmarks like GemsFDTD, however, it must predict the entire range accurately.

Benchmark	Improvement
astar	2.36x
GemsFDTD	1.02x
lbm	1.60x
milc	1.15x
vips	1.20x
GUPS	7.37x

Table 4.1: Oracle lifetime improvement over baseline

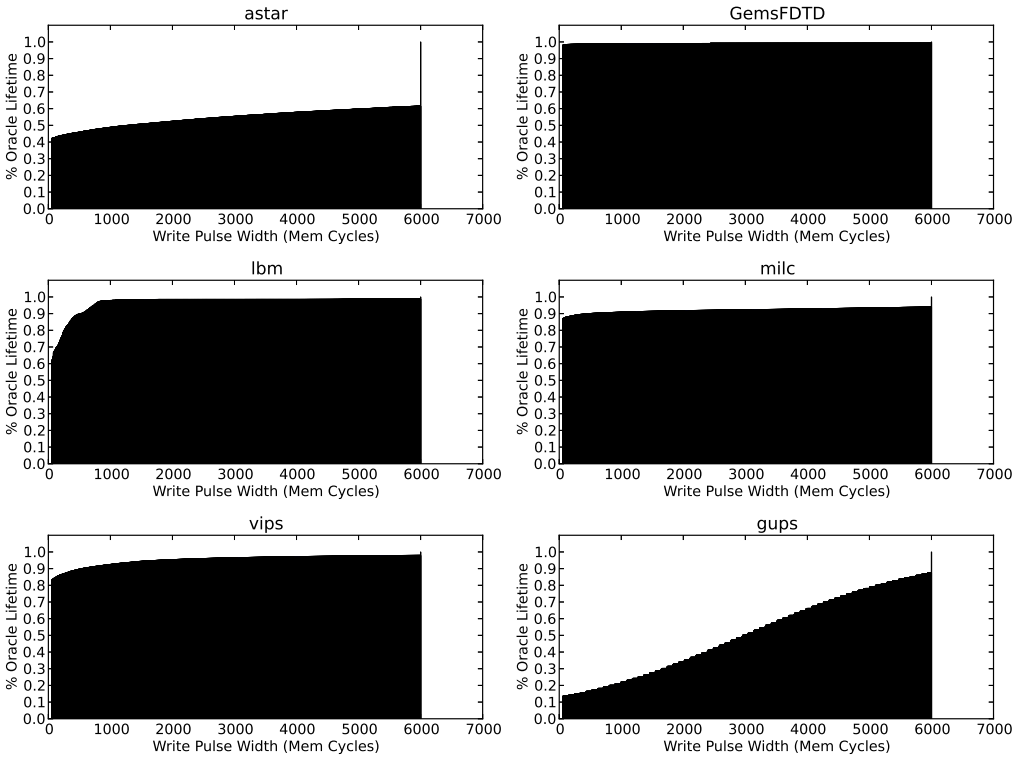


Figure 4.1: Cumulative distribution of oracle lifetime, per write pulse width

4.2 Approximating the Oracle

It is impossible to exactly model the oracle by knowing the value of T_{next} ahead of time. The best any wear-limiting scheme can do is predict T_{next} using information available at time $T_{current}$. Thus, our task becomes developing a predictor for T_{next} that accurately models the oracle. A first-order approximation of T_{next} can be derived from historical values of T_{next}, resulting in a steady-state, history-based predictor. Using the last n values of T_{next}, we predict the current value. In doing so we assume that history is indicative of the present, i.e. T_{next} has reached a steady-state. For implementations in which $n > 1$, a heuristic must be used to interpret the set of history values. Many solutions are possible: using the mean, median, or mode of the histories, extrapolating the next value using a least-squares regression, a combination of these methods, etc.

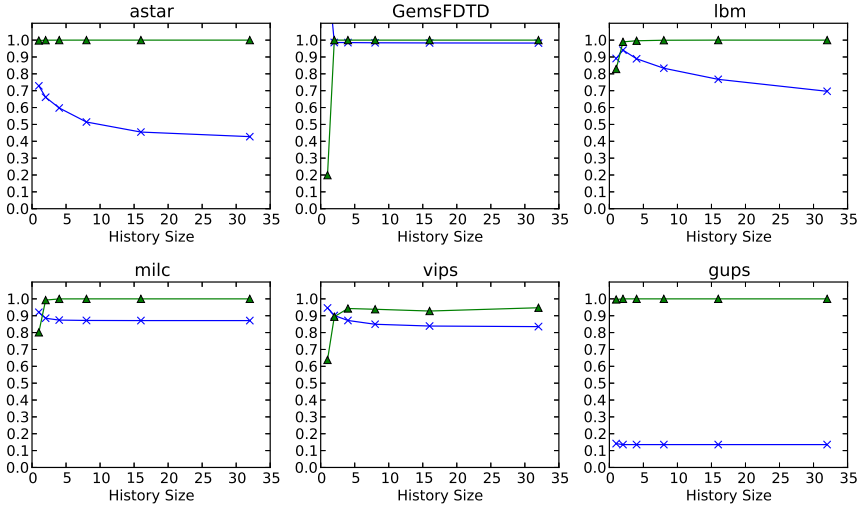
In our simulations, we calculate the value of T_{next} as the arrival time of CPU-to-memory requests. Because we intend to capture the rate at which the CPU requests a specific bank, we capture the arrival time on a per-bank basis and do so regardless of if the request is queued or blocked by the controller. For each write operation, the amount of slack available is calculated as outlined in Section 2.3. If slack is calculated to be zero or negative (i.e. T_{next} is anticipated to occur on or before the fastest write can finish), we cap the value at zero. If slack is calculated to be larger than $Slack_{max}$ (see Section 3.3), we cap it at this maximum value. Thus the write pulse width for a write is simply $tWP = tWP_{min} + Slack$.

For history length $n > 1$ we evaluate two heuristics: *min*, which selects the

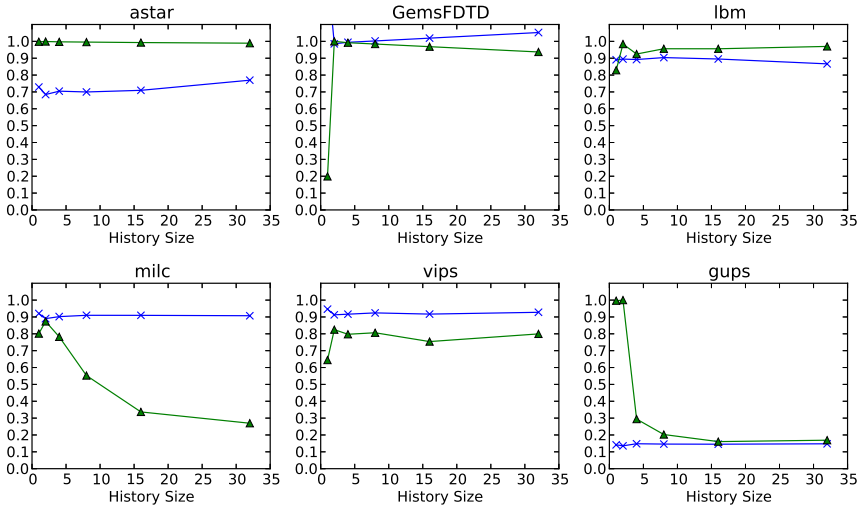
minimum of the n histories, and *slope*, which selects between the arithmetic mean and the minimum of the histories. To implement *slope* we calculate a linear best-fit line for the n histories. The slope of this line is indicative of the trend of T_{next}. A positive slope indicates a decreasing rate of requests, while a negative slope indicates an increasing rate. For an increasing rate we select a conservative estimate of T_{next}: the minimum of the histories. For a constant or decreasing rate we select the average. We choose *slope* because it represents an opportunistic approach for determining slack, as compared to the conservative approach of *min*.

Figures 4.2a and 4.2b show the results of evaluating our six benchmarks for the *min* and *slope* heuristics, respectively. We simulate each for history sizes $n = 2, 4, 8, 16, 32$. For comparison we include the results for $n = 1$, but this is neither a *min* nor *slope* heuristic. Each figure depicts the performance and lifetime of the heuristic as a percentage of the oracle IPC and lifetime. A few key observations can be made from these figures. First, we notice that *min* does as well or better than *slope* in both performance and lifetime for small values of n ($n = 2, 4$), with >95% of oracle IPC and 60% to 98% of oracle lifetime. For larger values of n, *slope* fairs better in terms of lifetime. However this lifetime improvement comes at the cost of performance degradation, resulting in anywhere from 45% to 84% smaller IPC for some benchmarks. In contrast, performance is relatively insensitive to the *min* heuristic (for $n > 1$). Given these results, we argue that *min* is clearly the better heuristic.

The exception in this comparison is GUPS, which shows no performance degradation but also no lifetime improvement when using the *min* heuristic for



(a) *min*



(b) *slope*

Figure 4.2: Fraction of oracle IPC and lifetime attained by the *min* and *slope* heuristics. The triangle marker indicates IPC. The X marker indicates lifetime.

$n > 1$. GUPS generates random memory accesses and therefore randomizes write slack. We expect that any history-based predictor cannot capitalize on random slack. However it is imperative that in such no-gain scenarios a predictor has no impact on performance. The *min* heuristic demonstrates this behavior exactly.

4.3 Shortcomings of the *min* Heuristic

4.3.1 Performance

The results in Section 4.2 demonstrate that for small values of n, *min* is a appropriate heuristic because it preserves oracle performance while attaining a significant portion of oracle lifetime. However, we cannot say that a specific small value of n is best for all benchmarks. The smallest n always results in the best lifetime, because the result of the *min* operation can only decrease with each larger history size. Conversely, a smaller n can result in worse performance. For a small n, individual variations in T_{next} have a larger impact on the *min* operation. Our results show that the degree to which a small n impacts performance is application-dependent. For the set of SPEC benchmarks (astar, GemsFDTD, lbm, milc) and GUPS, n has little to no impact on performance. However for the PARSEC benchmark Vips, IPC decreases with a smaller n: from 94% for $n = 4$ to 89% for $n = 2$.

Because performance is sensitive to n on an application-by-application basis, we cannot determine n statically as done in the previous section. Instead, we must do so dynamically. Doing so requires the use of a feedback mechanism: the ability to know when n is either too small or too large. Too small a value of n results in

performance degradation. We can easily measure this through *performance mispredictions*, in which *min* predicts a slack value larger than what the oracle shows to be correct. Too large a value of n is measured by the absence of performance mispredictions, in which *min* consistently predicts slack values too far below the oracle's values. Using these observations we can construct a feedback mechanism that dynamically adjusts the history size, n, as the application executes.

To do so, we simulate the oracle and our *min* heuristic simultaneously for the Vips benchmark. *Min* predicts a slack value at time $T_{current}$ which the oracle validates at time T_{next}, from which we obtain the magnitude of the performance misprediction. By counting the number of performance mispredictions within a given period, we can determine if n should be modified. Determining this requires defining four additional parameters: the magnitude of mispredicted slack that constitutes a performance misprediction, $Slack_{miss}$, the period between recalibrations of n, T_{recal}, the number of performance mispredictions in T_{recal} above which n is increased, $Miss_{inc}$, and the number of performance mispredictions in T_{recal} below which n is decreased, $Miss_{dec}$.

For our simulation, we define $Slack_{miss}$ as twice tWP_{min}. We define T_{recal} in terms of the number of write requests enqueued by the memory controller and set its value to $T_{recal} = 10000$ writes. For the Vips results presented in Section 4.2, *min* achieves the best lifetime for $n = 2$ and the best performance for $n = 4$. It follows that in each period T_{recal}, any feedback mechanism should select between these two values of n, depending on the performance requirements of the application at that time. In our simulation we deliberately choose values for

$Miss_{inc}$ and $Miss_{dec}$ that cause *min* to select between $n = 2$ and $n = 4$, namely $Miss_{inc} = 255$ and $Miss_{dec} = 63$.

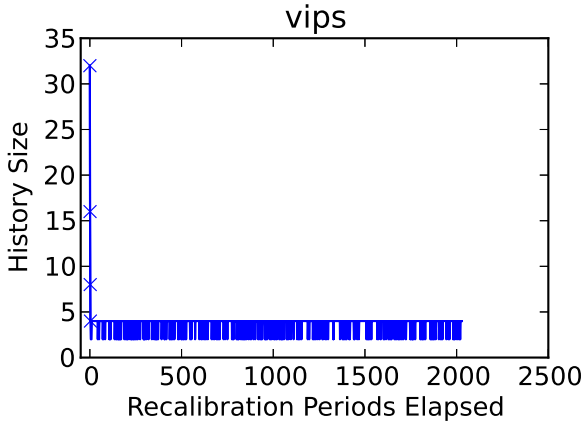


Figure 4.3: History size (n) selected by the feedback mechanism per recalibration period T_{recal}, for the Vips benchmark.

Figure 4.3 shows our feedback mechanism's selection of n per recalibration period T_{recal}. We initially set n to its maximum value $n = 32$, as this is the least likely to impact performance. The results indicate that the feedback mechanism quickly decreases n to its optimum values, $n = 4$ and $n = 2$, within three recalibration periods or 0.15% of the write operations for the Vips benchmark. Figure 4.4 compares the lifetime and performance results of our mechanism against the static scheme used in Section 4.2. The red circle marks fractional performance, while the teal circle marks fractional lifetime. We notice that performance is slightly better than statically choosing $n = 2$, and lifetime is slightly better than statically choosing $n = 4$. These results indicate that our feedback mechanism is successful. It correctly identifies the steady-state values of $n = 2$ and $n = 4$ for best

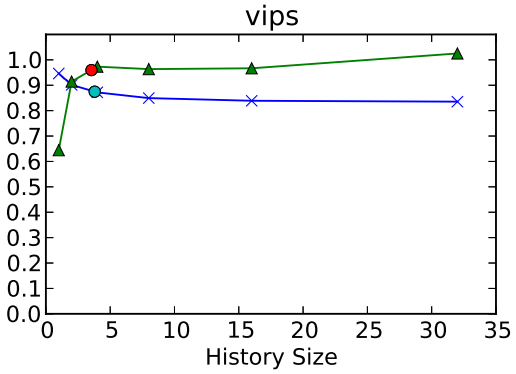


Figure 4.4: Comparison of selecting n dynamically vs. statically. The red circle marks the dynamic method's performance. The teal circle marks lifetime.

performance and lifetime in the Vips benchmark.

4.3.2 Lifetime

The degree to which *min* achieves the oracle lifetime varies from benchmark to benchmark. In GemsFDTD *min* achieves 98% of oracle lifetime, while in astar it achieves only 66%. This gap between *min* lifetime and oracle lifetime is essentially due to *lifetime mispredictions*–we predict a slack value smaller than what the oracle shows to be correct. To close the lifetime gap we must decrease the number of lifetime mispredictions.

To quantify the impact of lifetime mispredictions, we simulate the oracle and our *min* heuristic simultaneously as done in the above section. *Min* predicts a slack value at time $T_{current}$, and the oracle validates it at time T_{next}. Figure 4.5 shows the cumulative amount of memory wear introduced for each lifetime misprediction magnitude, for each benchmark. We notice that for astar, GemsFDTD, and milc,

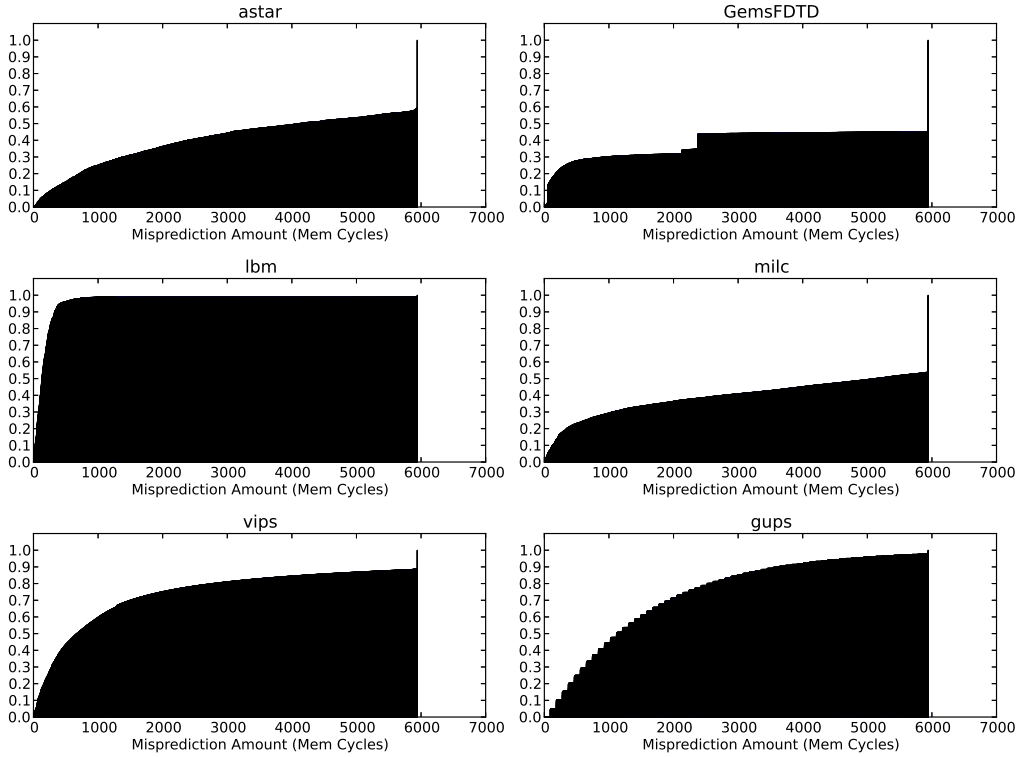


Figure 4.5: Cumulative distribution of additional wear due to lifetime mispredictions, per write pulse width.

roughly 50% of the wear results from lifetime mispredictions of 5940 cycles (i.e. $Slack_{max}$). Thus, 50% of the additional wear is contributed by writes that *min* issued with $tWP = tWP_{min}$, that should have been issued with $tWP = tWP_{max}$.

We argue that the misprediction behavior seen in astar, GemsFDTD, and milc is indicative of end-of-burst conditions. For access patterns composed of bursts of high-rate accesses followed by long periods of quiescence, any time-based history mechanism is a poor predictor. This is because access time history (the burst) is in no way indicative of the future (quiescence). Therefore, we must look to other

mechanisms to predict this end-of-burst condition.

One such mechanism is access location prediction, i.e. address prediction. Because GemsFDTD and milc are scientific applications, it is reasonable to assume that memory accesses within either application follow some well-defined pattern, such as the stride between elements in a matrix. It follows that mispredictions, which correspond to specific accesses within the application, should adhere to the same pattern or stride. Furthermore, any unique stride between mispredictions in an application is observable because we can observe exactly *when* a misprediction occurs.

To validate this claim, we again simulate our oracle and *min* heuristic together, recording the stride in bytes between lifetime mispredictions of size 5940. Table 4.2 shows the prominent strides in each of the three benchmarks. As suggested above, both GemsFDTD and milc have a single prominent stride value: 8192 bytes. Thus given an address for an initial lifetime misprediction of size 5940, we can accurately predict 93% of lifetime mispredictions of this type in GemsFDTD and 43% of this type in milc.

Benchmark	Stride	Freq.
astar	64B	0.25%
GemsFDTD	8192B	93%
milc	8192B	43%

Table 4.2: Prominent strides in the astar, GemsFDTD, and milc benchmarks.

Address prediction cannot predict the end-of-burst conditions in astar, however, as shown by its lack of a prominent stride value. This is unfortunate, as

astar has the most to gain from lifetime improvement of any of the benchmarks. Indeed, the lifetime improvement generated by our address prediction mechanism for GemsFDTD and milc is insignificant: 0.8% for GemsFDTD and 2% for milc.

It is important to note that these results are not necessarily indicative of *all* address prediction mechanisms, however, or all access location mechanisms. The complexity of the memory system provides a wealth of options to explore: access prediction on a per-bank basis, access prediction using access history, etc. Such explorations are outside the scope of this paper and are the subject of future work.

Chapter 5

Bank Configuration Sensitivity

In using arrival time per bank as an indication of T_{next}, our implementation naturally becomes sensitive to the timing and location of bank accesses. This itself is sensitive to the memory configuration, because the mapping of addresses to banks determines which banks service which accesses at what time. This makes our implementation potentially sensitive to bank configuration, i.e. the number of banks in a memory and their size.

To determine the magnitude of this sensitivity, we reevaluate our *min* heuristic using six different bank configurations for the 8GB PCM: 32 256MB banks, 16 512MB banks (the baseline configuration), 8 1GB banks, 4 2GB banks, 2 4GB banks, and 1 8GB bank. We divide the banks among the four ranks used in the baseline configuration, except for the 4GB and 8GB bank configurations which use 2 ranks and 1 rank, respectively. In each simulation, we use the optimal history size determined in Section 4.2. Because maximum write slack is sensitive

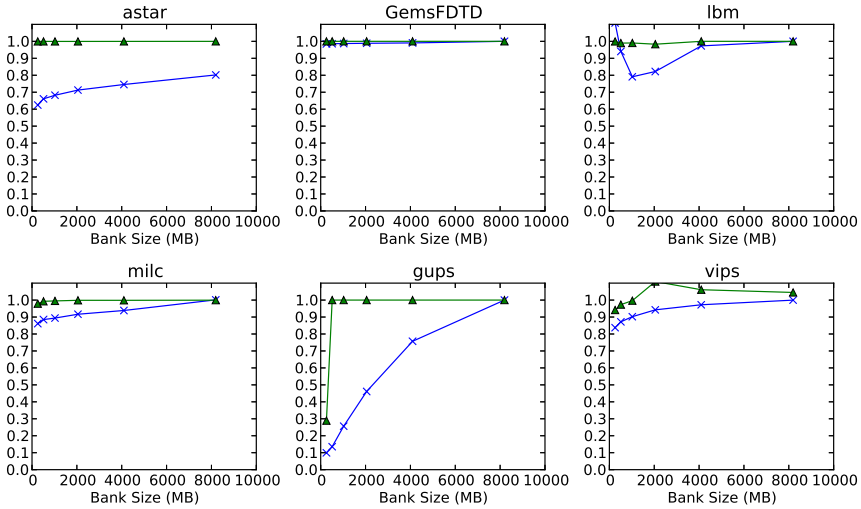


Figure 5.1: Fraction of oracle IPC and lifetime attained by the *min* heuristic for various bank sizes, using the optimal history size.

to bank configuration, it is necessary to reevaluate the oracle for each of bank configuration as well.

Figure 5.1 shows the results of this sensitivity analysis. For all benchmarks (except GUPS) it is clear that the performance of *min* is relatively insensitive to bank size. In each, the fraction of oracle IPC is >90%. Lifetime, however, is sensitive to bank size, in that the fraction of oracle lifetime increases as bank size increases. This does not necessarily indicate that the lifetime improvement over the baseline improves, however. This is because as bank size increases, bank count decreases. This increases the memory pressure per bank, as less banks must service the same number of requests, which in turn equates to less write slack available (as reported by the oracle). Based on these observations, these results demonstrate two important features of the *min* heuristic. First, for decreasing

write slack reported by the oracle, *min* is still able to achieve a large fraction of this slack. Second, *min* can adapt to varying degrees of memory pressure. Performance does not degrade with increasing bank size, showing that *min* correctly predicts the slack (or lack thereof) in each simulation.

The exception to these observations is, again, GUPS. For a bank size of 256MB, *min* achieves only 29% of oracle IPC. This performance degradation is the result of a poorly chosen history size, $n = 2$. We argue that performance degradation, unlike lifetime degradation, can always be corrected by increasing the value of n. Figure 5.2 validates this claim. It shows the lifetime/performance analysis for a 256MB bank size with statically selected history sizes $n = 2, 4, 8, 16, 32$. We notice that for $n = 8$, *min* achieves roughly 99% of oracle IPC. This underscores the need for a dynamically selected history size, as demonstrated in Section 4.3.1. Using a dynamic scheme, *min* can select an appropriate value of n not just for a specific application, but for a specific memory configuration as well.

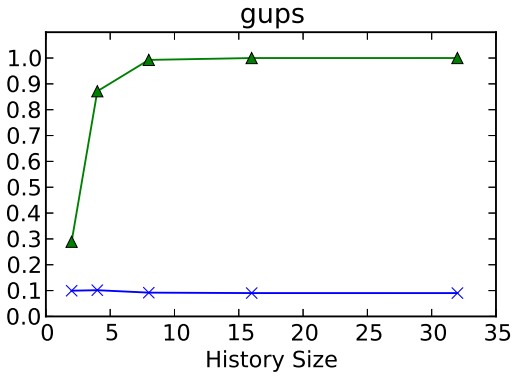


Figure 5.2: Fraction of oracle IPC and lifetime attained by the *min* heuristic using a 256MB bank size, for the GUPS benchmark.

Chapter 6

Future Work

The work presented in this thesis represents only an initial study of temporal wear-limiting. Indeed there are several extensions of the topics discussed in this thesis that should be studied, as well as several new ideas to be explored. First, further analysis should be performed on the dynamic history size concept discussed in Section 4.3.1. In this section we statically determine the threshold values that change the history size ($Miss_{inc}$ and $Miss_{dec}$). As these values are most likely application-specific, this defeats the intent of our idea: a completely dynamic implementation. Further analysis should be done to determine if these static values are acceptable across all benchmarks, or if they must be determined dynamically.

A second extension of this thesis is in regard to our lifetime metric. In this study we do not analyze the spatial distribution of wear for our various implementations. We argue that a good wear-leveling scheme should take *any* spatial wear distribution we create and spread it evenly across the memory. Thus memory

lifetime in this study is proportional the the total wear, not the wear on a line-by-line basis. Determining absolute lifetime requires simulating our *min* heuristic in conjunction with a state-of-the-art wear-leveling scheme such as Start-Gap [3]. As absolute lifetime is defined as the number of writes performed before a single memory line fails, our simulations must be executed until line failure. This is an intractable problem for our timing-accurate simulator, requiring the simulation of billions or likely trillions of total write operations.

We can approximate this timing-accurate simulation as follows: For each of our benchmarks, we capture a trace of the wear induced by writes to specific lines (rows) in memory. We then replay this trace successively and accumulate the per-line wear until an individual line fails. To do so, first we divide the 8GB memory into 64MB Start-Gap regions. Traditionally, Start-Gap defines a region as a set of logically adjacent 256B memory lines. Because swapping adjacent lines will interfere with bank write timing (and therefore bank write slack), we cannot use this method. Instead we define a region as a 64MB set of lines within a bank, thus preserving bank timing information. Because Start-Gap shifts the initial offset of a region for each gap rotation, we cannot simply record the wear induced per-line in a benchmark and then accumulate this information until line failure. Instead, we simulate a benchmark and capture a per-line wear trace for each region. We write out the wear information for every gap rotation in a region, or 26,214,400 writes (average of 100 writes per line in the 64MB region). We then replay and accumulate this per-rotation wear information successively, ensuring that we account for the change in region offset for each gap rotation. This replay

method may take many iterations to reach line failure. We optimize this further by replaying the traces only until the region offset reaches 0 again, for a total of 262,144 trace replays (64MB region / 256B per region). The wear accumulated in this "super" trace can then be accumulated successively with no need to account for the region offset.

The traces can be obtained using the timing-accurate simulation infrastructure described in this thesis and are recorded as $(address, wear)$ tuples. The replay of these traces can be done offline by simply taking $address$ and calculating the correct line within a region using the current region offset. Accumulating the wear for each gap rotation trace requires 262,144 operations (262,144 lines per region). Accumulating the wear for the super trace requires 262,144 gap rotation traces, for a total of approximately 68 billion operations. The number of super trace replays is dependent on the application. One super trace replay requires 262,144 operations.

A third extension of this thesis is exploring alternate methods to quantify and exploit write slack. Section 4.3.2 discusses several possible extensions for decreasing the number of *lifetime mispredictions* using access location prediction. As lifetime mispredictions are mostly attributed to end-of-burst conditions, it is essential to find prediction mechanisms outside the temporal realm. However, it is also important to look beyond simple prediction mechanisms. In this thesis we do not explore the possibility of reordering memory operations to achieve greater write slack. It is conceivable that with knowledge of memory operation priority, a memory controller could buffer and reorder operations to achieve more slack than

our oracle reports. Thus exploiting write slack adds yet another dimension to the complex task of scheduling memory operations.

Chapter 7

Conclusion

Non-volatile memory technologies provide a low-power, high-density alternative to traditional DRAM main memories. However, all non-volatile technologies suffer from some degree of limited write endurance. The non-uniformity of write traffic exacerbates this limited endurance, causing write-induced wear to concentrate on a few specific lines. *Wear-leveling* attempts to mitigate this issue be distributing write-induced wear uniformly across the memory. Orthogonally, *wear-limiting* attempts to increase memory lifetime by directly reducing wear. To our knowledge, this paper is the first to introduce the concept of *temporal wear-limiting*. With it, we make the following contributions:

- We provide an analysis of the physical mechanism that allows for temporal wear-limiting. Specifically, we quantify the trade-off between write latency and memory lifetime.
- We implement a mechanism for exploiting this trade-off. Our *min* heuristic

predicts future write latency using a history of per-bank write slack. We show that *min* is able to achieve roughly 70% to 90% of maximum write slack available per application.

- We show that performance is sensitive to the *min* history size, and provide a mechanism for dynamically determining the optimum size.

- We show that the lifetime improvement unattainable with our *min* heuristic is due to end-of-burst conditions, and make an initial study into using address prediction to identify these conditions.

The write latency / lifetime trade-off we explore in this paper provides yet another design knob for non-volatile technologies. Traditionally, limited memory lifetime is an artifact of the design process. It is a result of having a fixed latency / lifetime ratio. Our *min* heuristic allows lifetime to be defined on a system-by-system basis. By measuring the slack between write operations, we can increase write latency when an application does not demand high memory performance. This performance/lifetime trade-off affords greater flexibility to system designers, opening non-volatile technologies for use in a wider set of applications.

Bibliography

[1] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 362–373, ACM, 2011.

[2] "International technology roadmap for semiconductors," 2013. http://www.itrs.net/.

[3] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 14–23, ACM, 2009.

[4] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 383–394, ACM, 2010.

[5] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

[6] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 14–23, ACM, 2009.

[7] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 347–357, Dec 2009.

[8] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 141–152, ACM, 2010.

[9] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 50–61, Feb 2011.

[10] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, pp. 13–24, Jan 2013.

[11] J. McPherson, J.-Y. Kim, A. Shanware, and H. Mogul, "Thermochemical description of dielectric breakdown in high dielectric constant materials," *Applied Physics Letters*, vol. 82, no. 13, pp. 2121–2123, 2003.

[12] V. G. Karpov, Y. A. Kryukov, I. V. Karpov, and M. Mitra, "Field-induced nucleation in phase change memory," *Phys. Rev. B*, vol. 78, p. 052201, Aug 2008.

[13] D. B. Strukov and R. S. Williams, "Exponential ionic drift: fast switching and low volatility ofthin-film memristors," *Applied Physics A*, vol. 94, no. 3, pp. 515–519, 2009.

[14] K. K. Likharev, "Layered tunnel barriers for nonvolatile memory devices," *Applied Physics Letters*, vol. 73, no. 15, pp. 2137–2139, 1998.

[15] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pp. 392–397, Aug 2012.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[17] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler, "Running parsec 2.1 on m5," tech. rep., The University of Texas at Austin, Department of Computer Science, October 2009.

[18] "Gups," n.d. http://www.dgate.org/ brg/files/dis/gups/.