ABSTRACT

Title of Dissertation: SRTP: PREDICTING STORE REUSE TIME TO IMPROVE RERAM ENERGY AND ENDURANCE

Devesh Singh
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

ReRAM is an attractive main memory technology due to its high density and low idle power. However, ReRAM exhibits costly writes, especially in terms of energy and endurance. Prior studies demonstrate that retention can be traded off for write energy and endurance by employing *soft write operations* with lower currents. But given their reduced retention times, soft writes require refresh operations to prevent data loss. Unfortunately, a large number of refreshes are needed in between writes to infrequently updated data. Given the large capacity of ReRAM, always doing soft writes is not viable, as the exorbitant cost of refreshing every cell will outweigh any benefits from soft writes. Hence, a non-volatile memory system with soft writes still needs traditional *hard writes* and a way to choose between them.

Whether or not soft writes provide a benefit depends on the amount of time between back-to-back writes to the same data, which we call the *overwrite time*. As long as the cost for the soft write and its refreshes within the overwrite time window is less than the cost for a hard write,

then the original soft write is profitable. Otherwise, it would have been better to perform a hard write in order to eliminate the refreshes.

We propose SRTP, a predictor that learns the overwrite time between back-to-back writes to main memory and associates them with static store instructions in a prediction table. As dynamic stores execute, SRTP predicts whether a soft or hard write is best based on the magnitude of the predicted store reuse time. This soft write decision is placed in the cache hierarchy and eventually informs the writeback to the main memory to use either a soft or hard write. Our results show SRTP provides 2.6x - 4.1x improvement in endurance and 2.9x - 4.2x improvement in write energy over a state-of-the-art predictor. We also show that SRTP is within 18.5% of the Oracle policy. Furthermore, we integrate SRTP with a prior wear leveling technique, called Ouroboros, and show that SRTP improves actual memory system lifetime by 6.4x over a baseline that only performs hard writes. Finally, we introduce a new guiding metric for wear leveling in the presence of variable intensity writes and show that it improves wear-leveling efficacy by 5.7% to 19.4%.

SRTP: PREDICTING STORE REUSE TIME TO IMPROVE RERAM ENERGY AND ENDURANCE

by

Devesh Singh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Donald Yeung, Chair/Advisor
Professor Ankur Srivastava
Professor Manoj Franklin
Professor Alan Sussman
Professor Martin Peckerar

© Copyright by
Devesh Singh
2022

To my parents and sister.

Acknowledgments

First and foremost, I want to express my gratitude to Dr. Donald Yeung for being such an amazing advisor. His guidance, support, and encouragement have helped me become a better researcher by allowing me to explore novel ideas without worrying about failing. He is an exemplary researcher and teacher who leads by example. I have always admired his work ethic, and it is an ideal I will strive for. His emphasis on clear, concise writing, effective presentations, and communication helped me significantly develop my non-technical skills. I sincerely appreciate all that he has done for me as my Ph.D. advisor, critic, and mentor.

I am grateful to my collaborators, Dr. Bruce Jacob and Dr. Martin Peckerar for their insights and constructive feedback. Our group meetings and discussions have taught me so much. I cannot thank Dr. Ankur Srivastava enough for being a fantastic mentor. Additionally, I would like to express my gratitude to the committee members for their time, effort, and insightful criticism during the preparation and defense of my dissertation. I will forever be in debt to all of my teachers and mentors. Especially Dr. Shouri Chatterjee and Dr. Saif K. Mohammed, for igniting my interest in research.

I consider myself fortunate to have worked with so many brilliant peers during my Ph.D. My gratitude to my lab colleagues Candace, Daniel, and Yinuo for everything- from reading group discussions and research brainstorming to career guidance- cannot be adequately expressed. I would be remiss if I didn't Thank Dheeraj, Shang, Meena, and Luyi for their assistance and en-

couragement. I am equally lucky to have an amazing friend circle outside of work. Thank you, Mohit, Manas, both Nishants, Mack, Keri, Ananth, Alejandro, Ankit, Omid, Deshveer, Nirat, Suraj, Amit, Kartik, Chirag, Shubham, Drew, and Kate, for your companionship. You have made the last half-decade memorable.

I owe my deepest thanks to my parents, my sister, and my entire extended family for their unwavering love and support every step of the way. I owe them all that I am and all that I will be. Thank you, Maa and Phui, for the unconditional love, trust, support, and encouragement. Having you in my corner is my greatest strength. Thank you, Priya, my sister, my partner in crime, and Dee Dee to my Dexter. With you around, there is never a dull moment, and despite how many of my experiments you ruin, I am a better person for it.

Last but not least, I want to thank the ECE department staff for looking after me all these years.

Since it is hard to remember everyone, I sincerely apologize to those I unintentionally left out.

Table of Contents

Bibliography 151

List of Tables

List of Figures

List of Abbreviations

SRTP	Store Reuse Time Predictor
ReRAM	Resistive Random Access Memory
PCM	Phase Change Memory
DRAM	Dynamic Random Access Memory
RRM	Region Retention Monitor
MLC	Multi Level Cell
LLC	Last Level Cache
PC	Program Counter
CPU	Central Processing Unit
TLB	Translation Look aside Buffer
RTD	Reuse Time Detector
SWP	Soft Write Predictor
CF	Conductive Filament
SWI	Soft Write Interval
OOO	Out of Order
LRU	Least Recently Used
WPKI	Writebacks Per Kilo Instructions

Chapter 1: Introduction

With the rise of artificial intelligence and the era of big data, the capacity requirement of modern workloads is increasing at a drastic rate. To sustain the increasing demand of these workloads, hardware improvements need to keep pace, especially in the main memory systems. Figure 1.1, for instance, shows the trend of deep learning model sizes. Machine learning models rely upon more layers, more weights, and larger datasets to improve accuracy. Over the last three years, the largest models have grown over 1000x to over half a trillion parameters, while GPU memory has only increased by 5x (16 GB to 80 GB) [9,10]. This causes issues in both training and deployment of models. Less memory per machine means more nodes need to be added to fit the larger data sets and models. This reduces training throughput significantly due to high communication overhead between nodes [9]. Once trained, even the inference stage will require more capacity on the device side because of increasing model sizes.

DRAM, the technology of choice for main memory over the last several decades, is facing major challenges in capacity scaling. It is already lagging far behind the 3 nm technology node of CMOS transistors at 18 nm, and its scaling beyond 11 nm is unknown as per International Roadmap for Devices and Systems(IRDS) 2021 [11]. Traditionally, reducing the component size also means being able to fit more in the same die area leading to a reduction in the cost of each cell or getting more at the same price. But due to the increased complexity of the DRAM

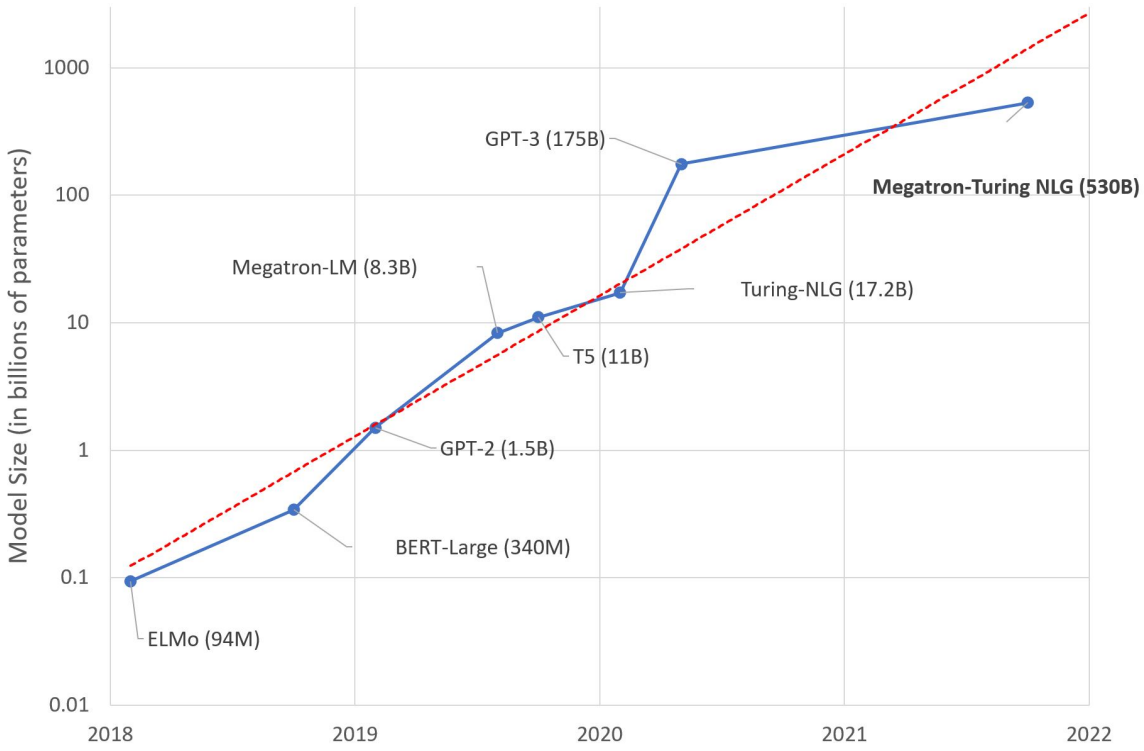

Figure 1.1: Natural language processing(NLP) model sizes' historical trend [2].

fabrication process, any device level shirking the manufacturers are able to achieve costs a lot more, causing the cost per bit of DRAM to plateau for the past several years. So even if DRAM manages to scale to the smaller feature sizes, the larger capacity comes at a proportionally higher cost, unlike in the past when the price came down as well. While the cells are not shrinking to fit more DRAM in the same space, increasing the capacity by allocating more space for DRAM chips is a non-starter too. As the capacity of DRAM is increased at the same technology node, the frequent global refresh operation becomes the bottleneck, with the memory system spending almost half of its power and time just to retain the data [12].

Emerging non-volatile memory (NVM) technologies, such as resistive random access memory (ReRAM) and phase change memory (PCM), offer much higher memory capacities compared to DRAM. Not only are their individual memory cells denser than DRAM at the same

technology node, but they are much more scalable as well. For example, ReRAM can certainly scale to 7 nm, and going further down to 5 nm is possible [13]. At the same time, they also provide improved power efficiency since their non-volatility eliminates the need for global refresh operations. Given these benefits, non-volatile memories have been studied extensively in research [14–22], and they have also become commercially available (*e.g.*, Intel's Optane memory [23,24]), leading to their deployment in real systems. This makes them an excellent candidate for main memory to keep pace with the demands of new workloads.

The main disadvantage of non-volatile memories, however, is the high cost of their writes. Not only do their write operations incur greater latency than read operations, but they also consume significantly more energy and cause the memory system to wear out. Higher latency can be tolerated since writes are buffered and do not block the execution of cores. Energy and wear out are the main limiting factors for non-volatile memory. Because of the significantly higher write energy consumption, expensive cooling systems would be needed, or the processor's limited power budget would be reduced. At the same time, non-volatile memory cells exhibit limited endurance, which is much smaller than the virtually infinite endurance of DRAM (ReRAM's endurance is reported to be in the range of 10^6-10^9). Writing indiscriminately to the NVM could lead to memory failure very quickly. This has inspired a lot of work on wear leveling techniques for NVMs to evenly spread out the writes across memory lines in an effort to maximize lifetime [6, 7, 25, 26].

One approach for addressing these problems is to reduce the frequency of writes to the non-volatile memory. *Hybrid memory systems* take such an approach by employing DRAM along with the non-volatile memory [19, 27–34]. For example, the DRAM can be placed in front of the non-volatile memory, acting as a cache that automatically filters the write stream [19]. Or, the

DRAM can be placed side-by-side with the non-volatile memory, relying on the programmer [28] or the operating system [27, 29] to explicitly allocate the most frequently written data in DRAM. This works well if DRAM is able to accommodate frequently written data. But in cases when the dataset doesn't fit, it will ping-pong between DRAM and NVM, amplifying the writes even further. Hence, this approach is constrained by DRAM's scaling challenges.

Rather than reduce the write frequency, another approach for addressing the high cost of writes in non-volatile memories is to give up some retention. Writes to ReRAM or PCM normally employ high currents, long write pulses, and/or multiple write cycles to switch the resistive medium (either metal oxide for ReRAM or chalcogenide glass for PCM). This ensures the data persistence that is expected for storage systems. (For example, the ReRAM from Crossbar, Inc. has a retention of 10 years [35]). However, in main memory, the majority of writes by a program are confined to a set of lines that are written multiple times over its execution. For all such lines, the usual years-long retention time is overkill since the written data will be over-written anyway before the retention time expires.

This opens the door to trade-off retention for improved write energy and endurance. In particular, writes to ReRAM or PCM can be performed using lower currents, shorter write pulses, and/or fewer write cycles. Such *soft writes* consume less energy and incur less wear, but they do not fully switch the resistive medium. For example, a soft write to ReRAM forms a smaller conductive filament (CF) within the metal oxide layer which is susceptible to dissolution [36–38], and hence, results in lower retention times. Chapter 2 covers this phenomenon in more detail.

Given their reduced retention times, soft writes require refresh operations to prevent data loss. While the refreshes themselves can also use soft writes, a large number of refreshes are needed in between writes to infrequently updated data, which defeats the purpose of using soft

writes in the first place. Hence, a non-volatile memory system with soft writes still needs traditional persistent or "hard" writes. Moreover, the memory system also needs a way to decide when to use soft writes and when to use hard writes. Having the option of doing a hard write will also allow ReRAM to avoid the refresh bottlenecks observed in DRAM. Our work intends to selectively perform soft writes only on the lines where they are worthwhile, thus obviating the need for refreshing the whole memory. Only the softly written lines need to be refreshed, keeping the refresh overhead in check as the memory capacity scales.

Recently, researchers proposed the Region Retention Monitor (RRM) [8] for controlling the selection of soft versus hard writes in Multi-Level Cell (MLC) PCM memory systems. Unlike our work which tries to improve energy and endurance, RRM uses soft writes to boost performance by reducing the number of write cycles, and hence the latency, of soft writes compared to hard writes. RRM tracks write frequency at the page level and uses soft writes only for the most frequently written pages. These "hot pages" are kept in a hardware table which issues refresh operations to the softly written data. For all other pages (not found in the table), RRM performs hard writes instead, thus eliminating many harmful refresh operations.

Although RRM was originally designed for performance, we adapted it to also improve energy and endurance. Unfortunately, we find that RRM does not translate well to these other objective functions. The problem is RRM performs soft / hard write selection using an ad hoc heuristic which is oblivious to the actual relationship between store locality and retention time that governs the effectiveness of soft writes.

Specifically, following every soft write, refresh operations are needed periodically in intervals equal to the retention time, as shown in Figure 1.2. Whether the soft writes provide a benefit depends on how much time elapses before the same data is written to again–*i.e.*, the time interval

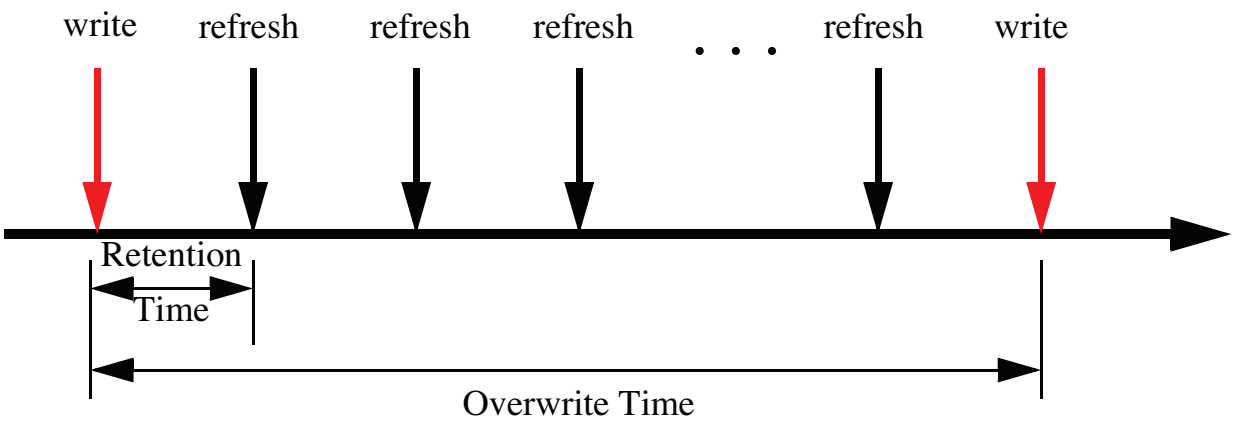

Figure 1.2: The profitability of soft writes depends on the last-touch store reuse time, and hence, the number of refreshes needed before the next write.

between writebacks to the same memory block from the last-level cache (LLC). In our work, we associate this reuse time back to the last store instruction that caused the initial writeback, so we refer to it as the *last-touch store reuse time*, or simply the *overwrite time*. As long as the cost for the soft writes occurring within this time window is less than the cost for a hard write, then the original soft write is profitable. Otherwise, it would have been better to perform a hard write in order to eliminate the refreshes. For instance, if one considers write energy as the cost metric, then soft writes should be performed whenever the following inequality holds:

$$\frac{OverwriteTime}{RetentionTime} < \frac{E_{HdWr}}{E_{SfWr}} \tag{1.1}$$

where, E_{SfWr} and E_{HdWr} are the soft and hard write energies, respectively. (Inequality 1.1 is the criterion for selecting soft writes to optimize endurance. Chapter 2 will discuss optimizing endurance versus energy in more detail.)

To achieve the greatest gains, a separate selection decision should be made at *every* dynamic store instruction. But RRM is ineffective at such fine-grain control because it aggregates memory-

side access information at a coarse page granularity. Our work makes the key observation that *store reuse times are correlated to store PCs*. Although store locality can vary significantly across different memory accesses, the accesses performed by the same static store instruction tend to exhibit a single dominant store reuse time. By training a CPU-side predictor on the dominant reuse time for each static store, Inequality 1.1 can be predicted accurately for every dynamic write to the main memory. We call this predictor the *Store Reuse Time Predictor, or SRTP*.

Numerous prior techniques have leveraged locality information for optimizing systems. In particular, many examples exist in the literature, including cache and TLB replacement policies [39–46] or cache partitioning techniques [47–49]. However, our work differs in that the reuse times we try to predict are much, much longer. Previous techniques mainly studied cache reuse, so the reuse times involved were relatively short–on the order of milliseconds. In contrast, SRTP predicts reuse for main memory that potentially spans multiple soft write retention intervals, which can last *several seconds*.

Training a predictor on multi-second events could result in long training times. Moreover, infeasibly large predictors might be necessary to track the huge number of events that occur over such long time horizons. Fortunately, store reuse tends to be stable on a per-store PC basis, as mentioned earlier. Also, a small number of static store instructions typically accounts for a large fraction of the dynamic stores executed. SRTP employs a small buffer, called the *Reuse Time Detector (RTD)*, tracking only two dynamic instances per static store instruction. The RTD can learn the most important store reuse times even when they last for a second or more.

Besides predicting Inequality 1.1, our technique must also issue refreshes to the softly written data. Unlike our prediction mechanism which resides on the CPU side, we track refresh operations on the memory side at the page level, similar to RRM. One difference, however, is

that our refresh tracking information is much larger because SRTP capitalizes on many more soft write opportunities. Whereas RRM combines the prediction and refresh information in the same table, we decouple them and move the refresh information–which is much larger but accessed less frequently–to the main memory.

Our work also integrates SRTP with wear leveling. We modify an existing wear leveling technique, Ouroboros [50], to account for the asymmetric wear across soft and hard writes. Advanced wear-leveling techniques like Ouroboros already store per-page address translation tables in the main memory. We integrate SRTP's refresh tracking information into these already existing data structures. Although SRTP's refresh information is substantial, it only adds 3.8% more memory on top of Ouroboros' address translation tables.

1.1 Contributions

This thesis makes the following contributions to enable non-volatile main memory systems by addressing two major concerns related to their write endurance and energy.

Adapt State of the Art: We adapt RRM, which was originally designed for performance, to control soft / hard write selection for improving energy and endurance in ReRAM-based memory systems. We also perform sensitivity studies on different RRM parameters to choose the best-performing values for comparison against our work.

Oracle-Based Policy: We introduce an oracular scheme that has perfect knowledge of the time when future writes are going to arrive at each line. Based on this future information, Oracle can predict soft vs hard writes with 100% accuracy. This limit study helps us estimate the maximum endurance/energy gain possible for each benchmark and demonstrates that RRM

leaves a lot of room for improvement.

Store Reuse Time Predictor: We propose a novel hardware predictor, called *SRTP*, to select soft versus hard writes at the CPU side on a per-dynamic store instruction basis. This allows our technique to make these decisions at a much finer granularity than RRM and get closer to the performance of the Oracle.

Wear Leveling Integration: We integrate our hardware predictor with a wear leveling scheme to benefit both techniques. SRTP uses wear leveling tables to track softly written data, so it can be refreshed to maintain reliable operation with low retention time. This keeps the refresh tracking overhead in check. Further, there are no wear leveling schemes that take different types of writes into account. We show this reduces the effectiveness of wear leveling techniques. Making soft write information available to wear leveling enables improved efficiency. To our knowledge, this is the first time wear leveling has been considered in the context of soft write techniques.

Experimental Evaluation: We undertake a simulation-based evaluation of SRTP, and show that it beats RRM by 410% and comes within 18.5% of the Oracle. We also demonstrate that SRTP can improve lifetime by 6.4x when coupled with practical wear leveling.

The rest of this thesis is organized as follows. Chapter 2 presents background on retention versus energy and endurance trade-offs. Then, Chapter 3 provides motivation, and Chapter 4 describes our SRTP technique. Next, Chapter 5 discusses experimental methodology, and Chapter 6 presents the results. Chapter 7 goes over the new guiding metric for wear-leveling and lifetime results, and Chapter 8 covers the additional benefit of customizing the device-level trade-offs for each program. Finally, Chapter 9 discusses related work, and Chapter 10 concludes the thesis.

Chapter 2: Background

This chapter gives the background necessary for the thesis. It starts with a brief review of various memory technologies that can be used for main memory. This includes traditional technology like DRAM as well as emerging non-volatile memory technologies like STT-RAM, PCM, and ReRAM. We motivate why ReRAM is the best candidate out of all of them and then move on to its device-level characteristics for trading off retention to improve write energy/endurance, which enable this work. Then, the chapter closes by summarizing wear leveling techniques used with emerging non-volatile memories.

2.1 Memory technologies

Main memory is a crucial component of any computing system, providing fast, high bandwidth, and high capacity storage for programs and data actively used by the processors. The penalty for missing in main memory is pretty high, making it important that the capacity of the memory system scales with the size of workloads. Table 2.1 shows the parameters of various devices that can be used to implement main memory. In this section, we discuss the benefits and limitations of each one of them.

	SRAM	DRAM	STT-RAM	PCM	ReRAM
Non-volatile	N	N	Y	Y	Y
Cell Area(F^2)	50-120	6-10	4-20	6-12	<1
Read latency(ns)	1	30	2-20	20-50	<50
Write latency(ns)	1	50	2-20	50-120	<100
Endurance	10^{16}	10^{16}	10^{10}-10^{12}	10^{6}-10^{8}	10^{6}-10^{8}

Table 2.1: Comparison of different memory technologies [1].

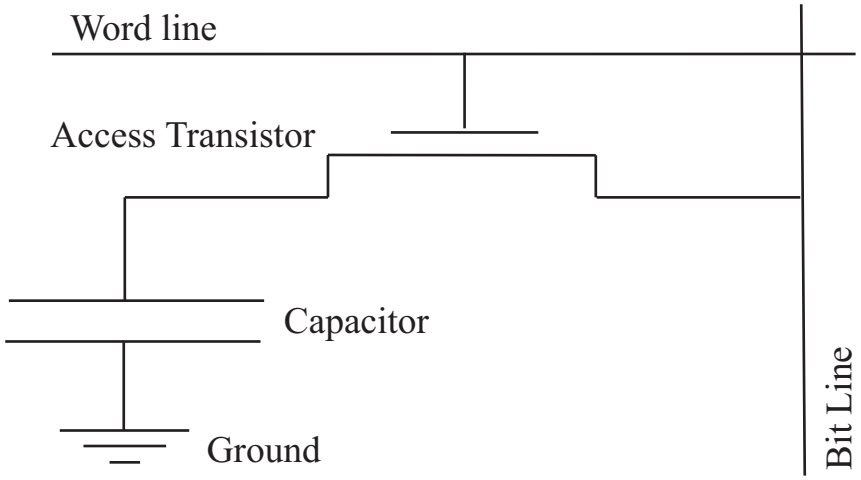

Figure 2.1: DRAM cell.

2.1.1 DRAM

Over the past several decades, DRAM has been the technology of choice for implementing main memory. As seen in Figure 2.1, each DRAM cell is made up of an access transistor and a capacitor. Data is stored in the DRAM cell by altering the charge in the capacitor. A logic '1' is indicated by charging the capacitor, while a logic '0' is indicated by the capacitor having no charge. The capacitor tends to leak charge over time, causing the DRAM cell to lose its stored data gradually. The volatile nature of DRAM necessitates global refresh for all of its cells at regular time intervals (typically 64 milliseconds).

Most main memory technology requirements, including low latency, low energy, and vir-

tually limitless endurance, are met by DRAM. The disadvantage of DRAM is that it exhibits relatively low density, which is caused by the requirement of per-cell access transistors. Worse yet, DRAM suffers from scaling issues, which are caused by the capacitor's inability to properly maintain charge after having its area lowered past a certain point [51].

2.1.2 STT-RAM

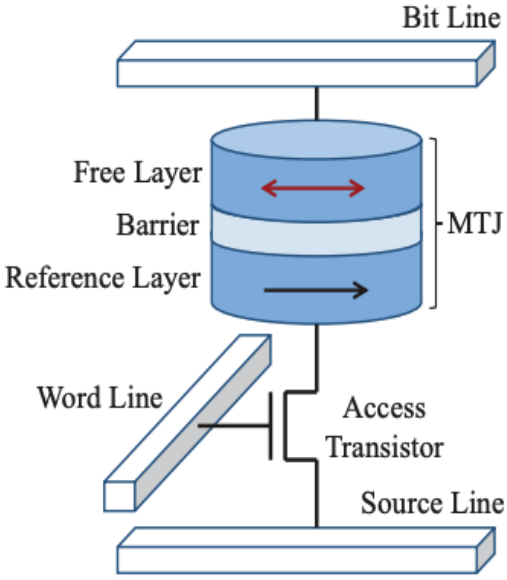

Figure 2.2: STT-RAM cell [3].

Spin-transfer torque random access memory (STT-RAM) uses a magnetic tunnel junction (MTJ) to store information. An MTJ contains two ferromagnetic layers (a reference layer and a free layer) and one tunnel barrier layer shown in Figure 2.2. The reference layer has a fixed magnetic direction, while the free layer's magnetic direction can be changed by applying a programming current. This, in turn, alters the resistance of the MTJ. If the magnetization of both ferromagnetic layers is in the same direction, MTJ is in a low-resistance state (LRS), and different magnetization of the layers puts the MTJ in a high-resistance state (HRS). Data is stored

by using LRS and HRS to indicate '0' and '1', respectively. STT-RAM cell is composed of one access transistor and one MTJ. Because STT-RAM is not volatile, it does not require frequent refreshes like DRAM or expend energy to maintain the data like SRAM. As shown in Table 2.1, STT-RAM is faster than DRAM and denser than SRAM, making it a good candidate for caches. Like other non-volatile memory technologies, it also suffers from a disproportionately high cost of writes in terms of both energy and latency, as well as limited write endurance. Although we concentrate on main memory candidates in this study, the methodology may be applied to any non-volatile memory that can trade off retention for better write performance. STT-RAM may also benefit from our findings because it has also shown retention trade-offs comparable to what we examine in this work [52].

2.1.3 Phase Change Memory

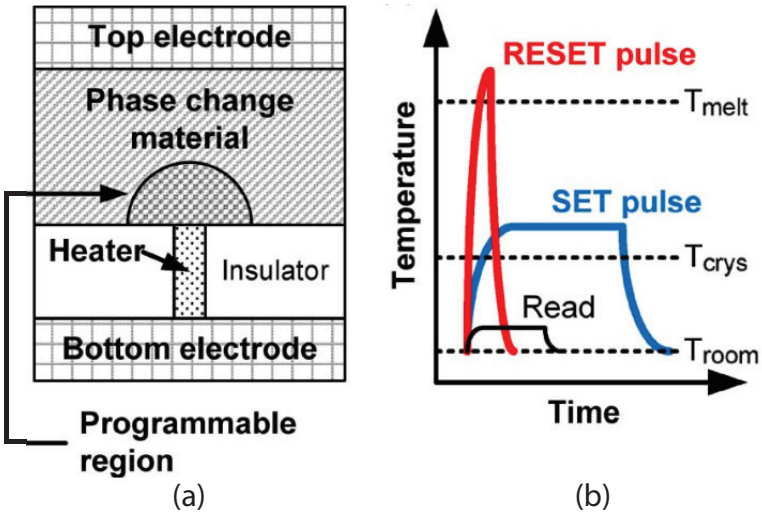

Figure 2.3: (a) Cross section of PCM cell. (b) Programming and reading pulses for PCM. [4]

Phase change memory, or PCM, is a non-volatile memory that makes use of a large resistance difference between the amorphous and crystalline states of phase change materials. A PCM

cell, as shown in Figure 2.3, consists of phase change material between the electrodes and a heating element that extends from the bottom electrode. The phase change is induced by injecting current into the cell. Depending on the magnitude and duration of the current, the phase of the material can be switched between amorphous and crystalline. A medium-sized, long-duration electrical pulse is used to heat the PCM cell above crystallization temperature but below melting point in order to set it into a crystalline state that corresponds to a logical '1'. To RESET the device into a high resistance amorphous state (logical 0), a large current is applied to melt the material, and it is then cut off abruptly.

PCM has higher latency and comparable density to DRAM. It is non-volatile with limited write endurance. There has been evidence of trading off retention for endurance improvement in PCM, too [53].

2.1.4 ReRAM

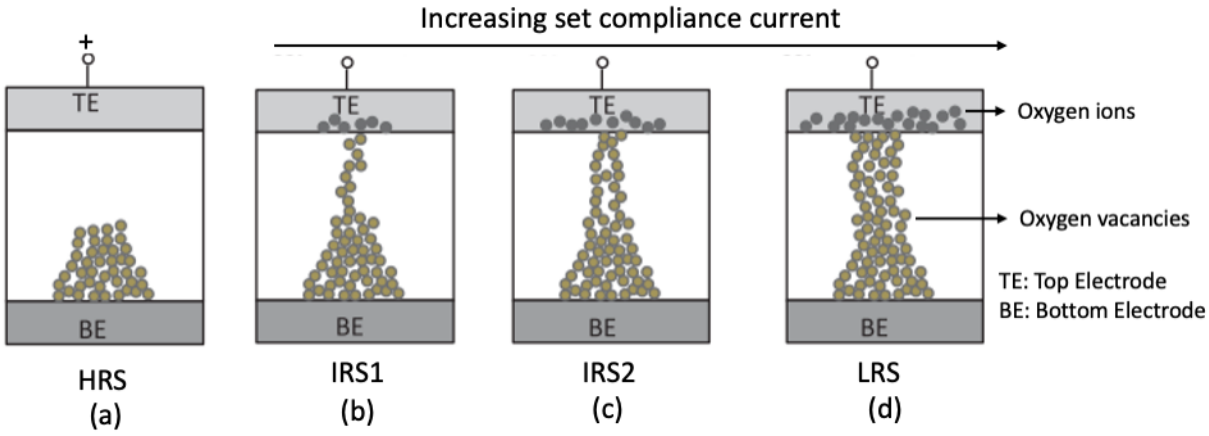

Figure 2.4: Switching in a ReRAM bitcell (from [5]).

A ReRAM cell consists of a metal oxide layer sandwiched between two metal electrodes, as shown in Figure 2.4. Without any external influence, this device is in the High Resistance State

(HRS) and acts as an insulator, Figure 2.4 (a). A sufficient voltage applied across the terminals leads to oxygen ions moving from the metal oxide layer to the positive electrode leaving behind a *conductive filament* (CF) of oxygen vacancies. This *set operation* puts the device in Low Resistance State, or LRS, as shown in Figure 2.4 (d). The device can switch back to HRS by applying voltage with an opposite polarity that makes the oxygen ions travel back into the metal oxide layer and recombine with the vacancies. This *reset operation* dissolves the CF. Traditionally, the LRS and HRS are treated as '1' and '0', respectively, to store data in ReRAM. To efficiently read data from a ReRAM cell, a small voltage that will not disturb the state of the cell is applied to determine the resistance of the cell.

ReRAM and PCM have the advantage of embedding a selector device inside a cell that obviates the need for per cell access transistor. The memory cells in series with the selector device are placed at the intersection of word and bit lines, as shown in Figure 2.5(a). The biasing scheme to read data from a selected cell is shown in Figure 2.5(b). The word lines and bit lines of the selected cell are biased such that a voltage difference of V appears across it. The rest of the lines are kept at V/2, which ensures no other cells have a voltage drop of V across their terminals. The high selectivity of the selector device guarantees that only the cells with a voltage drop of V across them are read or written while the rest are cut off. The selector does not come into the picture for endurance-retention trade-offs, so it is omitted from Figure 2.4 for clarity.

As shown in Table 2.1, ReRAM cells offer a good compromise between reasonable latency and high capacity. The extremely low cell area can be attributed to compact device size and the absence of per cell access transistor. The technology is scalable beyond 7 nm allowing the capacity to be increased even further.

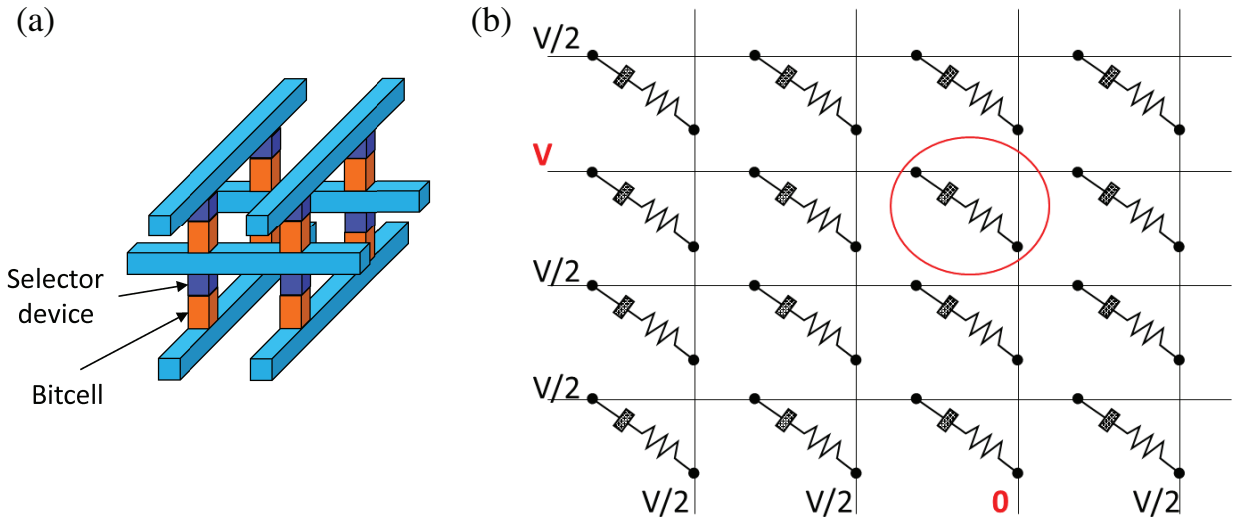

Figure 2.5: Crosspoint array with selector device and biasing scheme.

2.2 ReRAM retention vs Energy / Endurance Relationship

In this work, we focus on ReRAM-based memory technology, but the architectural techniques can be adapted to other non-volatile technologies that exhibit similar trade-offs. The operation of the ReRAM cell was previously discussed, and this section goes over the mechanism behind trading off retention for energy / endurance gains. Data from this chapter will be used to guide experiments in the simulation study.

It has been observed that the low resistance state of ReRAM is less stable than the high resistance state. This instability can be attributed to the CF dissolving naturally over time (even without an explicit reset) due to the random recombination of oxygen vacancies with mobile oxygen ions, and migration of oxygen vacancies [38]. These events do not happen frequently but end up dissolving the CF switching the ReRAM cell to high resistance state. The time it takes to lose the stored data depends on the size, or cross-sectional area, of the CF: the wider the filament, the longer the data lasts, whereas the narrower the filament, the sooner the data is lost [38,54,55].

The CF is formed during the set operation, and its size depends upon the set compliance current value. A larger set current leads to the formation of a wider, more stable filament, as shown in Figure 2.4(d). The exact relationship is, the cross-sectional area of the filament, CF_A, is *linearly proportional* to the set current, I_{SET} [56]. A wider filament means more energy is required in the reset operation to dissolve it. A reset operation would need to use a reset current, I_{RESET}, that is linearly proportional to the CF_A in order to disintegrate the filament [56]. Equation 2.1 demonstrates the relationship of the CF area with set and reset currents.

$$CF_A \propto I_{SET} \propto I_{RESET} \tag{2.1}$$

The switching voltage at which the CF forms or dissolves is a property of ReRAM cells and remains fixed. The latency is decided by the compliance current value where the set operation is stopped. Forming and rupturing a thinner filament will take less time, but for this work, we conservatively assume writes employ fixed latency. At constant voltage and latency, write energy is proportional to the I_{SET} and I_{RESET}, and hence, to CF_A.

In addition, the endurance of ReRAM bit cells is inversely proportional to the total write energy absorbed [57–59]. So, endurance is also inversely proportional to CF_A. In other words, we have the following relationship:

$$CF_A \propto Write\ Energy \propto Endurance^{-1} \tag{2.2}$$

ReRAM is traditionally deployed as a non-volatile memory, so significant energy is expended during write operations to form stable CFs that last for years. But Expression 2.2 shows that soft writes can use smaller I_{SET} and I_{RESET}, forming and dissolving narrower CFs, as illus-

trated by Figures 2.4(b) and (c), to both reduce energy and increase endurance [56].

As mentioned above, a narrower CF is less stable, and thus exhibits reduced retention time. In particular, the retention time of a ReRAM bit cell is *exponential* with the diameter of the CF, CF_D [36], leading to the following relationship:

$$RetentionTime \propto e^{p*CF_D} \tag{2.3}$$

where "$\mathtt{p}$" is a device-specific constant. Worst yet, the ReRAM bit cells within a large memory system will be subject to variations, so the actual retention time exhibits a distribution [38, 60]. The retention time for bit cells at the tail of this distribution can be further reduced by up to 1000x [38]. In our work, we use the model from [36] to derive retention time for a given CF size, and then further reduce it by 3 orders of magnitude to take the reliability of the tail bits into account.

In particular, our work employs a soft write that incurs 10x less energy compared to the basic hard write. As per Expression 2.2, this will also yield a 10x increase in write endurance. To achieve this, CF_A would need to reduce by 10x as well, leading to a 3.2x reduction in CF_D. Using Expression 2.2 and factoring in 1000x for the tail bits, there would be a 52,000x reduction in retention time. We assume a baseline hard write retention time of 10 years, which is based on Crossbar's non-volatile ReRAM technology [35]. So, the retention time for soft writes goes down to 10 seconds. This represents a worst-case retention time (due to the 1000x for the tail bits), making our energy and endurance results conservative.

2.2.1 Soft Write Criterion

Figure 1.2 illustrates the overhead of doing soft writes and its relation to the store to store reuse time or overwrite time. More time separation between consecutive writebacks to the same lines means more refreshes are needed to avoid data corruption. If the overwrite time value is known, a soft write decision can be made by weighing the benefit of soft writes and their added refreshes over hard writes. We call the overwrite time value where soft writes and hard writes break even the *Soft Write Interval* (SWI). Any overwrite time value less than SWI warrants performing a soft write. In this section, we derive the relationship of SWI with device parameters for optimizing endurance and total energy consumption.

2.2.1.1 Optimizing For Endurance

In Equation 2.2, we saw endurance is inversely proportional to write energy. So, to optimize endurance, we need to choose an SWI value such that write energy of soft write and its corresponding refreshes is less than the hard write energy. In other words, our objective for optimizing endurance is to find an overwrite time threshold such that the following inequality holds:

$$E_{SfWr} + E_{RefWr} < E_{HdWr} \tag{2.4}$$

where, E_{SfWr} and E_{HdWr} are the soft and hard write energies, respectively, and E_{RefWr} is the combined write energy of all refreshes incurred by the soft write. The reader should note that a single refresh operation involves reading the data and writing it back. So, it is equivalent to a

read and a soft write operation combined. The write energy of refresh is the same as a soft write, but the total energy of a refresh operation would be the sum of read and soft write energy. When optimizing endurance, we only consider the write energy portion of refresh because only writes have an effect on endurance. If ref_count refers to the number of refreshes incurred by the soft write under scrutiny, then we can rewrite Equation 2.4 as follows:

$$E_{SfWr} + ref\_count \times E_{SfWr} < E_{HdWr} \tag{2.5}$$

We also have the following relationship between the overwrite time and the number of refreshes.

$$ref\_count = \left\lfloor \frac{OverwriteTime}{RetentionTime} \right\rfloor \tag{2.6}$$

where, $\lfloor\rfloor$ is the floor operator, which ensures that the refresh count is always an integer. Substituting Equation 2.6 into 2.5.

$$E_{SfWr} + \left\lfloor \frac{OverwriteTime}{RetentionTime} \right\rfloor \times E_{SfWr} < E_{HdWr}$$

$$\left\lfloor \frac{OverwriteTime}{RetentionTime} \right\rfloor + 1 < \frac{E_{HdWr}}{E_{SfWr}} \tag{2.7}$$

Substituting the following identity for a floor operator:

$$\lfloor x \rfloor > x - 1 \tag{2.8}$$

$$\frac{OverwriteTime}{RetentionTime} < \frac{E_{HdWr}}{E_{SfWr}} \tag{2.9}$$

Inequality 2.9 provides the overwrite time criterion for selecting soft versus hard writes by weighing the benefit of soft writes over hard writes, while accounting for the additional refreshes that soft writes require. We call the threshold overwrite time when soft writes and hard writes break even normalized by the retention time the *soft write advantage (SWA)*. SWA appears on the right-hand side of Inequality 1.1 and changes according to the optimization objective. It is also the number of refreshes up to which performing soft write is advantageous (see Expression 2.6). SWA for optimizing endurance–*i.e.*, SWA_{end} is shown in Expression 2.10, which is also shown in Inequality 1.1. Because endurance is proportional to write energy (Expression 2.2), SWA_{end} is just the ratio of hard and soft write energies, which is 10x, as discussed in Section 2.2.

$$SWA_{end} = \frac{E_{HdWr}}{E_{SfWr}} \tag{2.10}$$

2.2.1.2 Optimizing For Total Energy

Another optimization objective of interest is total energy. The SWA for total energy, or SWA_{egy}, is not as large as SWA_{end} because the refresh operations shown in Figure 1.2 must first perform a read before performing a soft write. This read doesn't affect endurance but does affect energy. So, our objective for optimizing total energy is to find an overwrite time threshold such that the following inequality holds:

$$E_{SfWr} + E_{Ref} < E_{HdWr} \tag{2.11}$$

where, E_{Ref} is the total energy of refresh operations associated with the soft write. Since refresh is a combination of a read and soft write, the total refresh energy can be replaced by read energy (E_{Read}) and soft write energy.

$$E_{SfWr} + ref\_count \times (E_{SfWr} + E_{Read}) < E_{HdWr} \tag{2.12}$$

$$ref\_count = \left\lfloor \frac{OverwriteTime}{RetentionTime} \right\rfloor < \frac{(E_{HdWr} - E_{SfWr})}{(E_{SfWr} + E_{Read})} \tag{2.13}$$

Using Identity 2.8.

$$\frac{OverwriteTime}{RetentionTime} - 1 < \frac{(E_{HdWr} - E_{SfWr})}{(E_{SfWr} + E_{Read})} \tag{2.14}$$

$$\frac{OverwriteTime}{RetentionTime} < \frac{(E_{HdWr} + E_{Read})}{(E_{SfWr} + E_{Read})} \tag{2.15}$$

Equation 2.15 gives us the reuse time criterion for optimizing total energy. It also gives us the following formulae for SWA_{energy}:

$$SWA_{egy} = \frac{E_{HdWr} + E_{Read}}{E_{SfWr} + E_{Read}} \tag{2.16}$$

2.2.1.3 Figure of Merit

In this work, we are interested in improving the endurance of the ReRAM-based memory system, which will, in turn, improve the memory system's lifetime. The traditional metrics of Instructions Per Second (IPC) or execution time are not adequate for defining the performance

of relevant methodologies. Some previous works employ lifetime to evaluate the effectiveness of their schemes. While this is the end goal of the techniques, lifetime results are a combination of several factors along with soft writes. We will see in Chapter 7 wear leveling policies impact lifetime too. Hence, we need a metric that quantifies the impact of soft write schemes in isolation in order to exclude interference from other factors in the study.

We introduce a performance equivalent term for endurance savings by soft writes and use it to contrast soft write methodologies. The endurance of ReRAM cells is inversely proportional to write energy (Expression 2.2), or in other words, endurance wear caused by a write operation is directly proportional to its energy. If a soft write technique reduces the write energy by a certain amount, the endurance wear is also reduced by the same amount. Keeping that in mind, we use *effective* SWA_{end} to quantify the endurance impact. Effective SWA_{end} is the ratio of the total write energy for baseline where all writes are hard writes and total write energy for the soft write technique. This metric quantifies the factor of reduction in wear out provided by a soft write technique. Chapter 7 will empirically demonstrate that the soft write policies improve the lifetime by a factor roughly the same as the effective SWA_{end}. (Given the parameters from Chapter 2, the maximum effective SWA_{end} is 10x, which occurs when every write is a soft write, and there are no refreshes).

$$effective\ SWA_{end} = \frac{Write\ energy\ for\ Baseline}{Write\ energy\ for\ soft\ write\ technique}$$

$$effective\ SWA_{end} = \frac{WriteCount_{Baseline} * E_{HdWr}}{HardWriteCount * E_{HdWr} + SoftWriteCount * E_{SfWr}}$$

where $WriteCount_{Baseline}$ is the total original writes for baseline without any soft write schemes. $HardWriteCount$ and $SoftWriteCount$ are the total hard and soft writes, respectively, performed by the soft write technique being evaluated. Any auxiliary writes introduced by the soft write policy will be folded into the correct category, *e.g.*, refreshes are included in $SoftWriteCount$. The expression can be further simplified using Equation 2.10.

$$effective\ SWA_{end} = \frac{WriteCount_{Baseline} * SWA_{end}}{HardWriteCount * SWA_{end} + SoftWriteCount} \tag{2.17}$$

2.3 Wear leveling background

As shown in Table 2.1, limited write endurance is ubiquitous for non-volatile memory technologies. This is a major challenge for both main memory and storage applications; as the memory lines start wearing out, the whole device could become unusable very fast. Unfortunately, the memory access pattern of programs has been shown to be highly non-uniform. Some memory locations are written very frequently, while others might not get any writes whatsoever. This will cause the frequently written lines to fail much sooner, curtailing the memory system's lifetime to unrealistic values. On average, the memory system lifetime is reduced by 20 times due to non-uniform writes, even if a significant number of spare lines are present [6]. Non-volatile memory systems can also be targeted by malicious attacks that repeatedly write the same line, wearing them out within minutes [61].

Wear leveling is a mechanism that migrates heavily written data from more worn-out physical memory locations to lightly used ones in order to make writes more uniform. The 20x

realizable gain in the lifetime makes wear leveling indispensable for non-volatile memories. This section briefly discusses the common wear leveling techniques for non-volatile main memory systems and then describes the one integrated with our approach. We call the incoming addresses at the memory controller *logical* addresses. These logical addresses are mapped to certain *physical* addresses that point to the actual memory locations where the data is stored. Wear leveling techniques intercept this logical-to-physical mapping to direct writes meant for the same logical location to different physical locations. Effectively, by moving the data around, wear leveling techniques add another level of indirection in address translation at the main memory level after the page tables. The wear leveling techniques can be broadly categorized into two groups: *restricted algebraic remapping based* and *table based*, depending on how they store and resolve their migration-generated indirections. The following sections describe both techniques with representative examples.

2.3.1 Table-Based Wear Leveling

Secondary storage systems based on NAND Flash memory use table-based wear leveling, and there has been significant work in extending it for emerging NVM technology at the main memory level [17, 26, 62, 63]. The techniques in this category have the following properties in common. They divide memory into regions of a certain size, called blocks, and keep track of the wear out of each physical block as well as the write intensity of the logical blocks mapped to them. Then periodically, the logical blocks with high write intensity are migrated to target physical blocks to spread out the hotspots and uniformly distribute the writes. Typically, less heavily written physical blocks are chosen as targets, but some techniques also mix in randomly selected

physical blocks to make wear leveling resistant to attacks. Since this migration changes the physical location of the data frequently, these techniques also need a fully associative mapping table from logical blocks to physical blocks in memory, hence the name.

The advantage of table-based wear leveling is that it is aware of heavily used regions which allows it to perform more targeted migrations for wear leveling. The downside is it can only do the wear leveling at a coarse granularity to amortize the tracking overhead with large block sizes. This works well for Flash-based storage since its wear-out causing erase operations can only be done for large block sizes (1MB-8MB) [64, 65]. Erasing all lines in a block together makes them wear out uniformly. But for byte-addressable emerging NVMs, uneven wear out can occur inside the coarser blocks. Unfortunately, table-based techniques alone can handle wear leveling between larger memory regions well, but not within them.

2.3.2 Restricted Algebraic Remapping Based Wear Leveling

These are the simplest wear leveling techniques that do not use tables to keep track of the remapped data. They instead utilize deterministic, easy-to-compute algebraic expressions [6, 25, 66–69]. These techniques relocate the data in a constrained fashion such that indirection can be resolved via an algebraic function. This obviates the need for a tracking table, reducing the hardware overhead of the scheme significantly. *Start-Gap* [6] is a representative technique from this category, and we use it to explain the advantages and shortcomings of this group.

Wear leveling is achieved by periodically moving each line to its neighboring location such that, eventually, it leads to all lines getting moved up by a unit and causing a circular shift in the whole region. This remaps the logical addresses to new physical addresses and frequently written

locations write to a new location evening out the spread of writes.

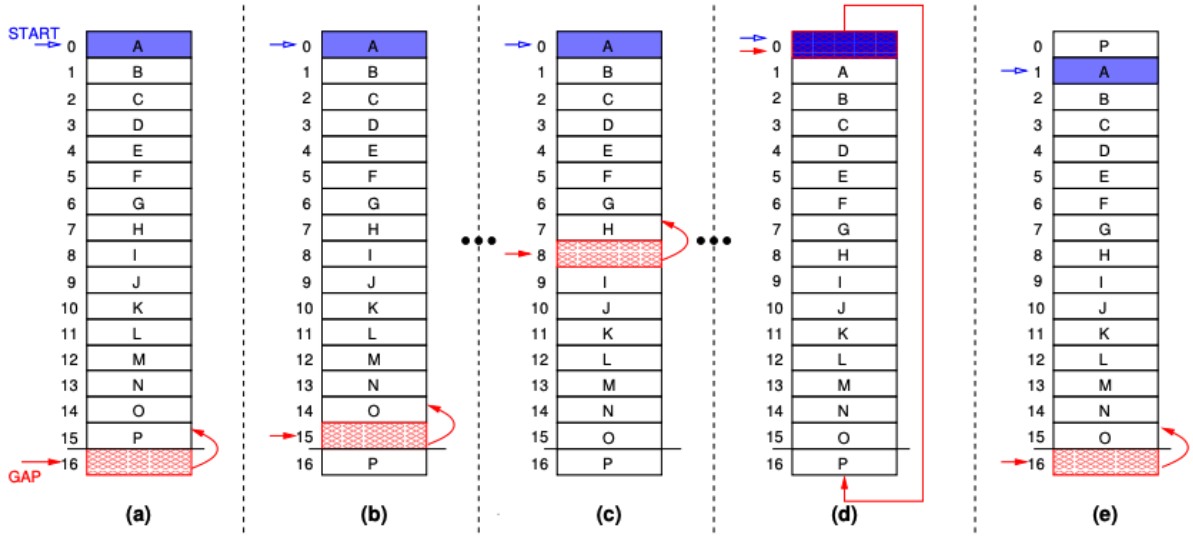

Figure 2.6: Start-Gap wear leveling on a memory containing 16 lines [6].

Figure 2.6 shows a memory region consisting of 16 lines with Start-Gap wear-leveling. An extra line is added to the region called the *Gap Line* to help the migrations. The only extra hardware needed for the whole region are two registers called *Start* and *Gap*. The Start register tracks the location of the top line of the region in the beginning, and the Gap register tracks the index of the gap line. After the region receives a certain number of writes (usually 100), a gap movement is triggered. During the gap movement, the contents of the line on top of the Gap Line (gap register - 1) are copied into the Gap Line, as shown in Figures 2.6 (a,b) and 2.7 (a). This shifts a single line to a new physical location, and its writes are directed to the new location now. The Gap register is also decremented since the gap line moved up. After a certain number of gap movements, the gap line reaches the top of the region shown in Figure 2.6 (d). The next gap movement swaps the gap line with the very last line of the region leading to a circular shift of the memory region in Figure 2.6 (e). This is called a *Gap Rotation*, and it increments the Start register and resets the Gap register. After gap rotations equal to the number of lines in the

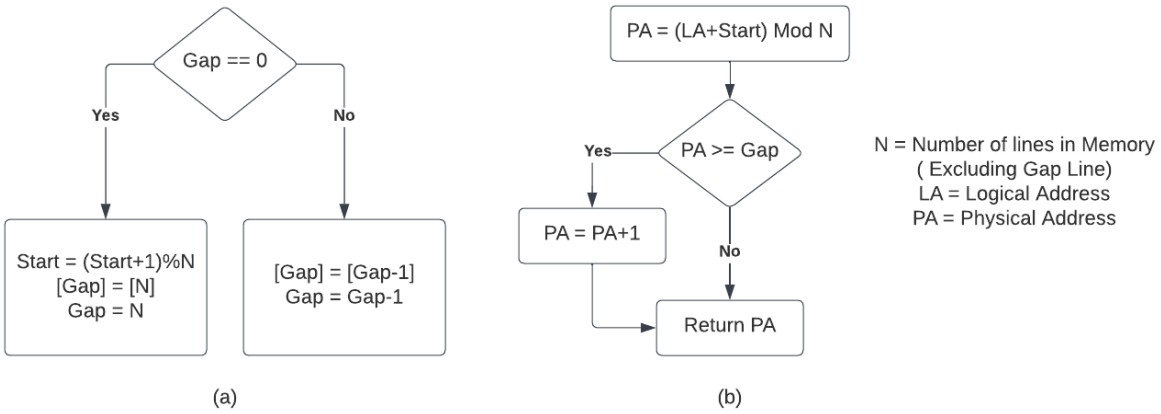

Figure 2.7: (a) Flow chart of Gap Movement operations. (b) Restricted algebraic mapping of Logical Address to Physical Address in Start-Gap.

region, each logical line would have been mapped to every physical line of the region, making the write distribution more uniform. The advantage of this approach is that we can translate the logical addresses to their physical locations at any point using simple calculations shown in Figure 2.7(b) without needing a translation table that keeps track of mapping from logical to physical lines. It only needs to store the Start and Gap registers as well as a write counter for each region. So, the overhead per region goes from 2*N to 3, where N is the number of lines in the region.

The main advantage of Start-Gap, as discussed earlier, is the small overhead per region and fast translation. The overhead is also independent of the size of lines being moved around, allowing the scheme to perform migrations at a very fine cache block granularity. The shortcoming of the technique stems from the very same source. Since we are not tracking writes on a per-line basis, the wear leveling technique moves every line in the region rather than targeting just the frequently written addresses. The heavily written lines are moved only once every Gap-Rotation, and the size of the region determines the Gap-Rotation period. If the period is too long, fre-

quently written lines could possibly wear out before they are relocated since the moves are not targeted. These techniques work really well for smaller regions, but as the region size increases, their efficacy goes down. Start-Gap can be done independently on multiple smaller regions. This will make the writes uniform within each region but will prevent wear leveling from exploiting uneven wear across different regions.

2.3.3 Hybrid wear leveling schemes

The two general schemes discussed above are complementary. The algebraic scheme is good at wear leveling in small regions, and table-based schemes excel at wear leveling across larger region sizes. The hybrid wear-leveling schemes combine the two in a hierarchical fashion such that they cancel out each other's weaknesses. *Ouroboros* is one such scheme that is used in this work. It is a two-level wear leveling scheme that migrates data at the local and global levels. Local wear leveling uses Start-Gap for each frame to keep wear out of memory cells within it balanced. At the global level, a table-based scheme is used for wear leveling across frames. The global wear leveling occurs at epoch boundaries, when the total writes since the last epoch exceed a Global threshold (10^8). This triggers a reorganization event where a small set of logical blocks are migrated to different physical blocks. During an epoch, local wear leveling continues within each physical block. When the number of writes to a physical block exceeds a local threshold (100), a start-gap movement is triggered shown in Figure 2.7 (a).

Figure 2.8 shows the entries of the table for this scheme. The page number field from the incoming logical address is used to index into the translation table. The translation table has the physical page number of the memory location where the logical address currently maps to.

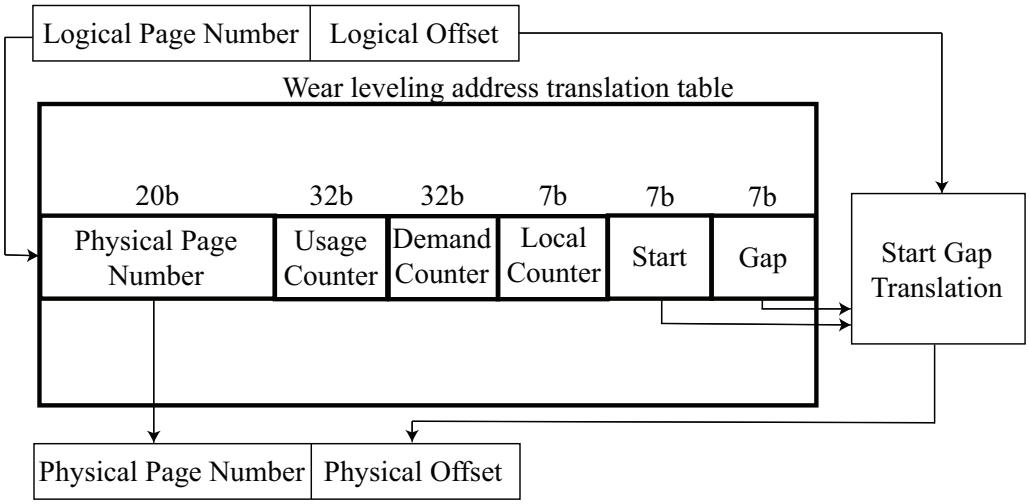

Figure 2.8: Translation table of Ouroboros [7].

The table also stores Start, and Gap registers for each page. The logical offset to index into the logical page is transformed into the physical offset by using these registers and performing the operations specified in Figure 2.7 (b). The physical page number and offset are combined to get the location of the data being requested in memory, as shown in Figure 2.8.

The translation table also has several counter fields. The*usage counter* keeps track of the wear out of a physical page by counting the writes to it. The*demand counter* keeps track of the writes to a logical page to help determine frequently written data. The*local counter* also counts writes to the physical page and is used to trigger gap movement within the page. Once the gap movement is triggered, the local counter is reset to zero. At the global threshold, a small subset (10) of high-demand logical pages are selected and migrated to low-usage physical pages. Details of the migration policy can be found in the original work [7].

Chapter 3: Motivation

This chapter motivates our work by quantifying how much room exists for soft write techniques to improve energy and endurance. We consider a state-of-the-art soft write selection technique, RRM, adapting it for write energy and endurance gains. Then, we present the Oracle policy that provides the upper bound and compare RRM against it. Finally, we discuss insights into closing the gap between them, which will lay the foundations for our approach.

3.1 Region Retention Monitor

In general, soft writes should be employed for frequently written data, whereas hard writes should be used for infrequently written data. A natural strategy, therefore, is to profile the write frequency and to use the profile for driving soft / hard write decisions. The Region Retention Monitor (RRM) [8] is a recent soft write technique that adopts this approach.

Figure 3.1 illustrates RRM. RRM employs a hardware table between the LLC and the memory controller (MC). It gets two inputs from the last level cache and sends two outputs to the memory controller. The RRM table observes the writes to dirty cache blocks in the LLC and makes a soft write decision on evictions as they write back to an NVM main memory. In order to keep the overhead in check, each entry in the RRM table tracks information per page. The "write_counter" field is used to figure out if a page is frequently written and hence

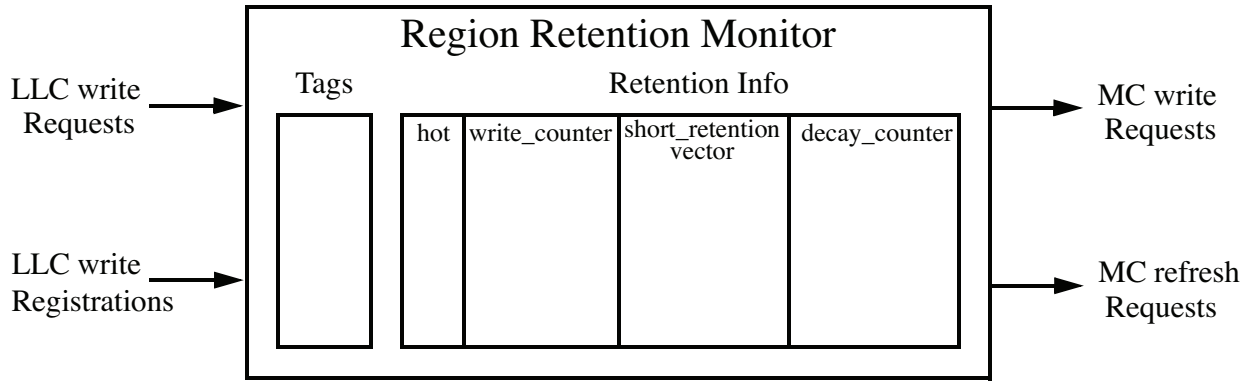

Figure 3.1: Region Retention Monitor hardware table. (Adapted from [8]).

a good candidate for soft writes. Whenever an LLC write is issued to a dirty cache block, a "write_registration" request is sent to the RRM table. If the corresponding page entry is found in the table, its write_counter is incremented. Otherwise, a new entry is inserted into the RRM table according to the least recently used (LRU) replacement policy. Only dirty blocks are considered for measuring write frequency to rule out pollution by streaming writes. Streaming writes exhibit spatial locality, but RRM is interested in capturing temporal locality. When the write_counter exceeds a certain "hot_threshold", the page is considered frequently written, and its "hot" bit is set. By default, writebacks from the LLC employ hard writes, but once a page's hot bit is set, the writebacks switch to using soft writes. When a dirty block is evicted from the LLC, the write request is sent via the RRM table. If the page corresponding to the memory address being written is found, RRM uses the hot bit to issue a soft or hard MC write request. This adds the RRM table to the critical path of the writebacks, and a delay equivalent to RRM lookup is added to memory writes. Since writes are non-blocking, this does not impact the performance in any significant way.

Besides controlling the soft and hard writes decisions, the RRM table is also responsible for tracking softly written data and issuing refreshes. At every retention interval, RRM checks the

hot bit of its entries. For each entry in the RRM table with the hot bit set, a refresh request for its corresponding page is issued to the memory controller. RRM handles refreshes within a page at memory block granularity. For that purpose, it tracks memory blocks within a page that have been softly written using a bit vector, called the "short_retention_vector". Refreshes are only issued to the memory blocks marked in the bit vector. Since the circuitry to issue refreshes is present in the CPU die, RRM also adds a refresh queue at the memory controller to queue these requests. The memory controller drains this queue at a higher priority to make sure all refreshes happen before the retention time expires. Issuing all refresh requests in this manner has the potential for creating bursts of memory accesses at retention intervals where RRM issues refresh requests for all of its hot pages. These bursts can create congestion as the regular memory requests line up behind the refresh requests that need to be handled at higher priority. Unfortunately, RRM does not circumvent this problem by spreading out these refresh requests throughout the retention interval.

After identifying the hot data for soft write candidates, it is also necessary to detect if and when the hot data transitions back to being infrequently written; otherwise, the refresh operations could nullify the benefits of the soft writes. In RRM, this is implemented using the "decay_counter." Periodically, the decay_counter is incremented until it rolls over, at which point the write_counter is re-checked to see if it still meets the threshold for a hot page. If so, the write_counter is reduced in half, and the decay_counter is reset. If the page stops receiving write-backs, then the write_counter will eventually drop below the hot threshold. When this happens, the hot bit for the page is cleared, and hard writes are issued to all softly written data. This is known as a *reset hard write*, which persists the data and allows the RRM table to stop issuing refreshes to the page. In the original RRM work, the rollover duration for the decay counter was

the same as the retention time. This only allows a page with a single retention time interval to accumulate enough write_counter increments. The reset needs to be done when the cost of soft writes, including refreshes, becomes larger than the cost of hard writes. We observe that a lot of these premature resets are not needed, and they cause a lot of unnecessary reset hard writes. From Section 2.2.1, we know that the actual reset threshold is SWA times retention time since that is the overwrite time when soft writes and hard writes break even. Hence, in this work, we update the decay threshold such that the decay counter rollover period is equal to $SWA * retention\ time$. We also verified the improved performance of the new rollover period experimentally.

3.2 Oracle Soft Write Selection

To assess the effectiveness of RRM, we developed the Oracle policy for selecting soft versus hard writes. As shown in Figure 1.2, the best soft / hard write decision depends on how far apart back-to-back writes to the same memory location are. In our experiments, we simulate an oracle mode which records the timestamp for every LLC writeback so that this overwrite time can be computed precisely. More specifically, when performing a writeback, the simulator looks up the timestamp for the last writeback to the same memory block, and computes the overwrite time. The simulator then uses this overwrite time to evaluate Inequality 1.1 and determine whether the *previous* writeback should have used a soft or hard write. (The Oracle currently optimizes for endurance; we could also evaluate Inequality 2.15 to optimize for total energy). Accordingly, the simulator attributes (after the fact) the write energy for either a soft or hard write to the previous writeback. In the case of a soft write, the simulator also attributes the write energy for $\lfloor \frac{OverwriteTime}{RetentionTime} \rfloor$ number of refresh operations as well. Then, the simulator updates the memory

block's timestamp with the current time, and continues execution.

Effectively, the oracle scheme knows the time when each future write is going to arrive for every line and uses this information to make a perfect soft/hard write decision. It also handles refreshes at cache line granularity. RRM and, as we will show in Chapter 4, even our technique handles refreshes using a global timer. Both of these policies will refresh softly written data whenever the global timer completes a retention period. This could lead to unnecessary refreshes. For example, if the overwrite time of stores to a line is shorter than the retention time, then refreshes are not needed because the soft writes will keep on refreshing the data within the retention period. Each memory line in the oracle scheme has its own timer. This timer is reset whenever the line receives a write. As a result, it avoids any unnecessary refreshes incurred by having a global timer. Hence, the oracle technique not only provides the best possible soft write decision for a given benchmark and optimization objective, but it also accumulates the minimum refreshes too. In short, the results of the oracle scheme provide us the limits of endurance/energy improvements affordable by soft writes. The implementation of the Oracle policy is described in Algorithm 1. We add it to our simulator, and compare RRM against it. (Chapter 5 will provide more details on our simulator).

Figure 3.2 shows the endurance results achieved across several SPEC CPU2017 benchmarks [70]. It plots the effective SWA_{end} for the soft write techniques (RRM and Oracle). As discussed in Section 2.2.1.3, effective SWA_{end} quantifies the factor of reduction in wear out provided by a soft write technique, and it is capped at 10x for our parameter values. For RRM, two sets of bars are reported: "RRM-base" which employs an RRM table with 4096 entries, matching the original RRM paper [8], and "RRM-aggr" which employs a more aggressive 32,768-entry table. Parameter values and overheads for both versions of RRM are provided in Table 5.3.

Algorithm 1 Oracular scheme algorithm.

while Write issued (to line i) **do**
    if ! the first write to line i **then**
        $overwrite\_time \leftarrow current\_time - previous\_write\_time_i$
        if $overwrite\_time \geq overwrite\_time\_threshold$ **then**
            record hard write for previous write to line i
        else
            $ref\_count \leftarrow floor(overwrite\_time/retention\_time)$
            record soft write and ref_count refreshes for previous write to line i.
        end if
    end if
    $previous\_write\_time_i \leftarrow current\_time$
end while
    ▷ When the program ends final write to each line should be hard to make the data persistent.
for all lines **do**
    if There was a previous write **then**
        record hard write for the line.
    end if
end for

As Figure 3.2 shows, there is significant room for improving RRM. RRM-base has an effective SWA_{end} of just 1.5x. One reason why these gains are so modest is that RRM-base can only track 4096 hot pages, which is not enough for our SPEC 2017 benchmarks. When the RRM table is increased to 32,768 entries–*i.e.*, using RRM-aggr–the effective SWA_{end} jumps to 2.4x. However, Figure 3.2 shows the Oracle policy achieves an effective SWA_{end} of 7.6x. Not only is this fully 5x better than RRM-base, but it is also over 3x better than RRM-aggr. This motivates our study to bridge the gap between prior techniques and what is theoretically possible.

3.3 CPU-Side Reuse Time Prediction

The Oracle policy from Section 3.2 knows the future overwrite time for every LLC writeback, and uses it to evaluate Inequality 1.1 at each dynamic write. Our work tries to emulate this fine-

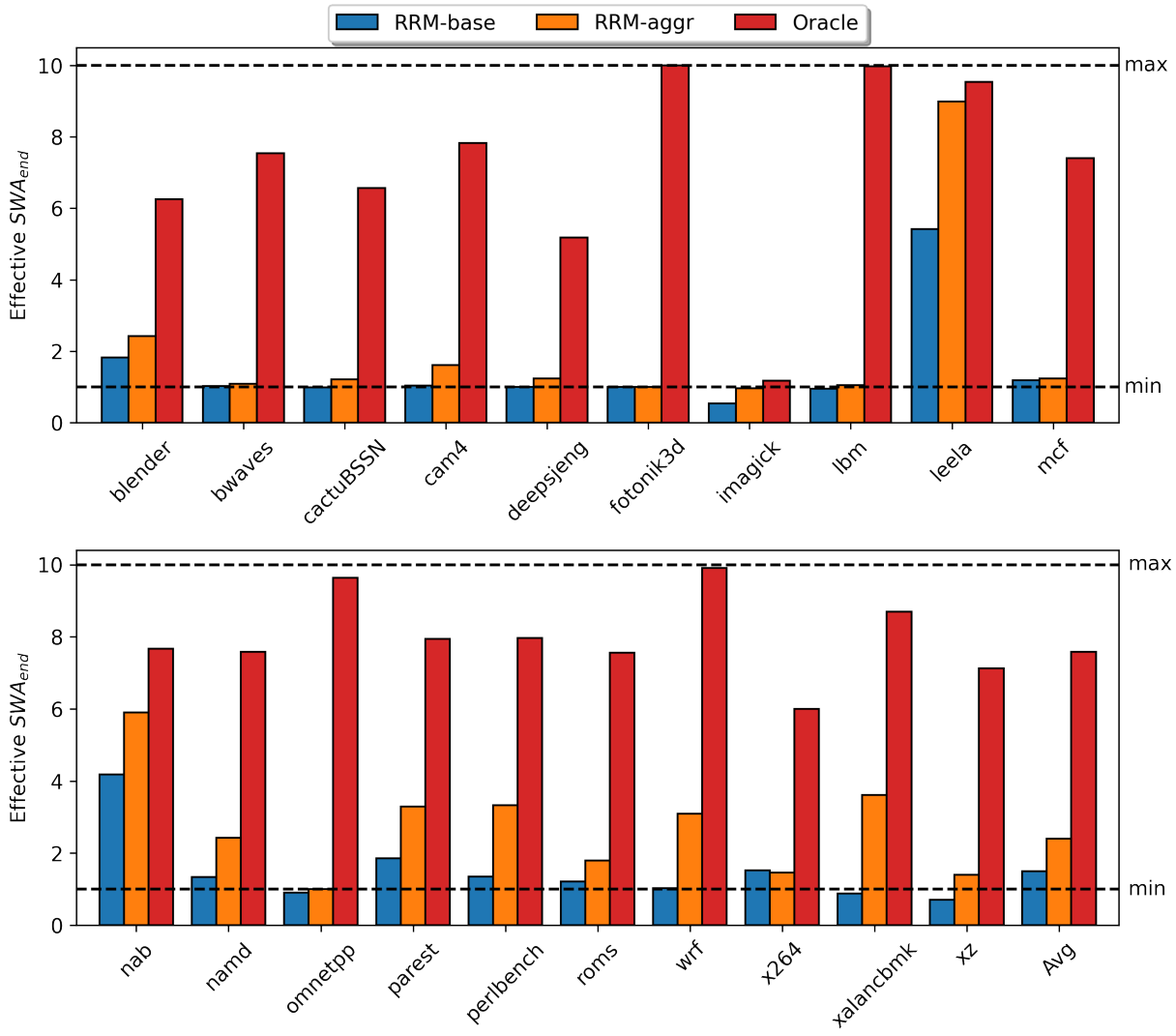

Figure 3.2: Effective SWA_{end} for RRM and Oracle assuming 10 second retention time and 1/10th wear for soft writes.

grained soft write selection performed by the Oracle. The overhead of doing so for each memory line would be prohibitively expensive, and the results would not be as good without oracle's foresight. We make an important observation that argues in favor of performing fine-grained soft write decisions at every write instruction executed instead of at their corresponding writebacks. In order to do so, we observe the overwrite time for certain dynamic writes (which the Oracle omnisciently knows for all writes) and use these samples to *predict* the soft write criterion (either

Inequality 1.1 or Inequality 2.15) for every dynamic store. Our technique is called the Store Reuse Time Predictor, or SRTP. If high prediction accuracy is achieved, then SRTP can approach Oracle's performance.

Unfortunately, as shown in Figure 3.2, RRM is not effective at such predictions. The problem is that RRM keeps memory access information on the memory side. In order to make decisions like the Oracle, RRM would have to track access counts on a per-memory location basis which is completely out of the question. RRM instead aggregates access counts at the page level, but even so, the amount of predictor state can still be very large. This not only results in large predictors, but it also incurs long training times (training is needed for *every page* in memory), which reduces the number of soft write opportunities that prediction can capitalize upon.

Rather than make predictions from the memory side, SRTP performs the predictions from the CPU side instead. It observes time intervals between writes to the same main memory location, and uses them to directly assess either Inequality 1.1 or 2.15. *SRTP associates the resulting soft write decisions back to the static store instructions linked to those writes*. As the store instructions execute again, SRTP predicts for every dynamic instance, either soft or hard write, using the soft write decision previously made for the corresponding static store. The technique is driven by the following insights into program behavior and store reuse time:

1. At the time of eviction, a dirty cache block could have been modified by multiple instances of different static store instructions. The only one that matters is the last store instruction to hit the cache block prior to eviction/writeback from LLC- *i.e.*, *the last-touch store*.

2. Usually, there could be a lot of unique store instructions in a program but only a few last-touch store instructions. For example, in the SPEC CPU2017 benchmarks studied in

the thesis, on an average only 76 static last-touch store instructions account for 90% of writebacks to the main memory. Because there are a much smaller number of static last-touch store instructions than there are memory locations stored to, SRTP's predictor state is significantly reduced compared to RRM.

3. We find that CPU-side prediction can be very accurate thanks to *last-touch store correlation*. While overwrite times can vary significantly across different dynamic writes to main memory, those linked to the same static last-touch store instruction tend to exhibit a single dominant overwrite time. To illustrate, Figure 3.3 shows the overwrite time histograms associated with the top-three static last-touch store instructions (PC0, PC1, and PC2) across five different SPEC CPU2017 benchmarks (bwaves, cam4, lbm, wrf, and x264). Overwrite time correlation around a single "peak" is clearly visible in each histogram. While the peak can be wide for some benchmarks (cam4, PC2) or accompanied by secondary peaks in other benchmarks (x264, PC0), there is nevertheless a single dominant overwrite time in all cases. Hence, there is also a single soft write decision dictated by Inequality 1.1 or Inequality 2.15 that is appropriate for each static last-touch store instruction as well. If we learn the overwrite time peak associated with each static last-touch store PC, it can be used to make soft write decisions for all the lines touched by it. By training our predictor per last-touch store PC, we can exploit this correlation to achieve high prediction accuracy.

4. Unlike prediction, which SRTP performs at the CPU side, refresh and reset hard writes are tied to specific memory locations, and must be performed at the memory side. Worse yet, as we approach the Oracle's performance, the amount of softly written data will increase, making the state for tracking refreshes and resets significantly larger. Fortunately, this state

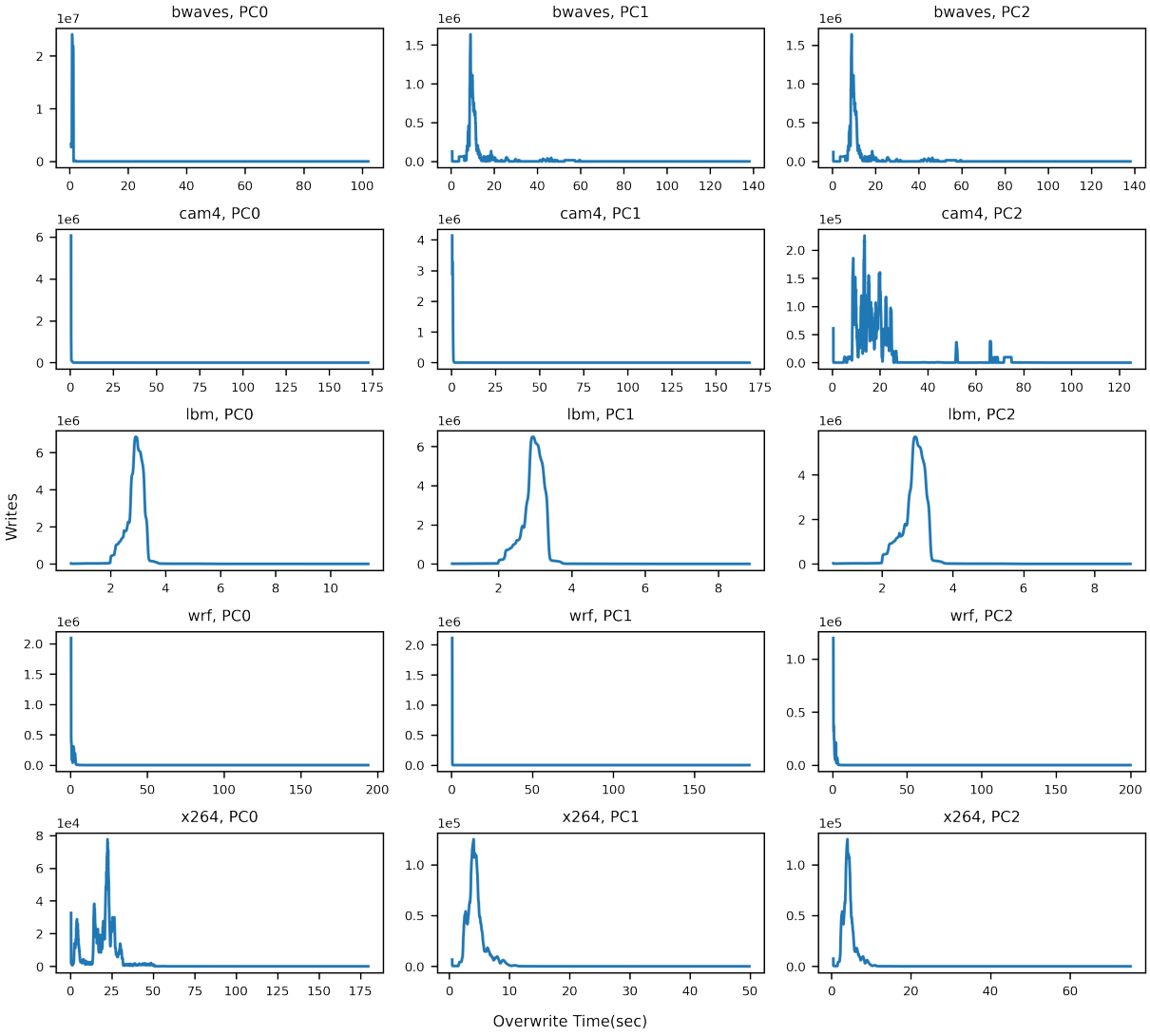

Figure 3.3: Reuse time histograms for the top three last-touch stores from three SPEC CPU2017 benchmarks.

does not need to be accessed frequently. SRTP decouples the refresh and reset information from its predictor, and moves it into main memory to take advantage of higher capacities. Later, we show how this information can be integrated into wear-leveling data structures.

Chapter 4: Store Reuse Time Predictor

Having motivated potential benefits, we now present the details of our technique. SRTP adds new hardware components to a multicore CPU, modifies its existing cache hierarchy, and also enhances the tables of the wear leveling controller. First, the Reuse Time Detector (RTD) is added between the LLC and the memory controller. It learns the best soft write decision for each static store instruction by observing store reuse times. Second, the Soft Write Predictor (SWP) is replicated per core at the same level as their private L1 caches. SWP uses the soft write decisions learned by the RTD to predict either soft or hard write for every dynamic store instruction. Each SWP module predicts for its associated core. Figure 4.1 shows a multicore CPU with SRTP modules integrated. Third, SRTP adds additional tracking state to all caches that enables RTD to associate overwrite times to static store instructions as well as SWP to propagate its write decisions. Finally, SRTP maintains information for driving refresh and reset hard writes in main memory, which can be integrated with wear leveling. In this chapter, we go over each component individually and then show how they function together.

4.1 Cache Enhancements

Before discussing the new components added to the processors, let's go over the enhancements to an already existing one, the cache hierarchy. Figure 4.2 shows the additional state added to

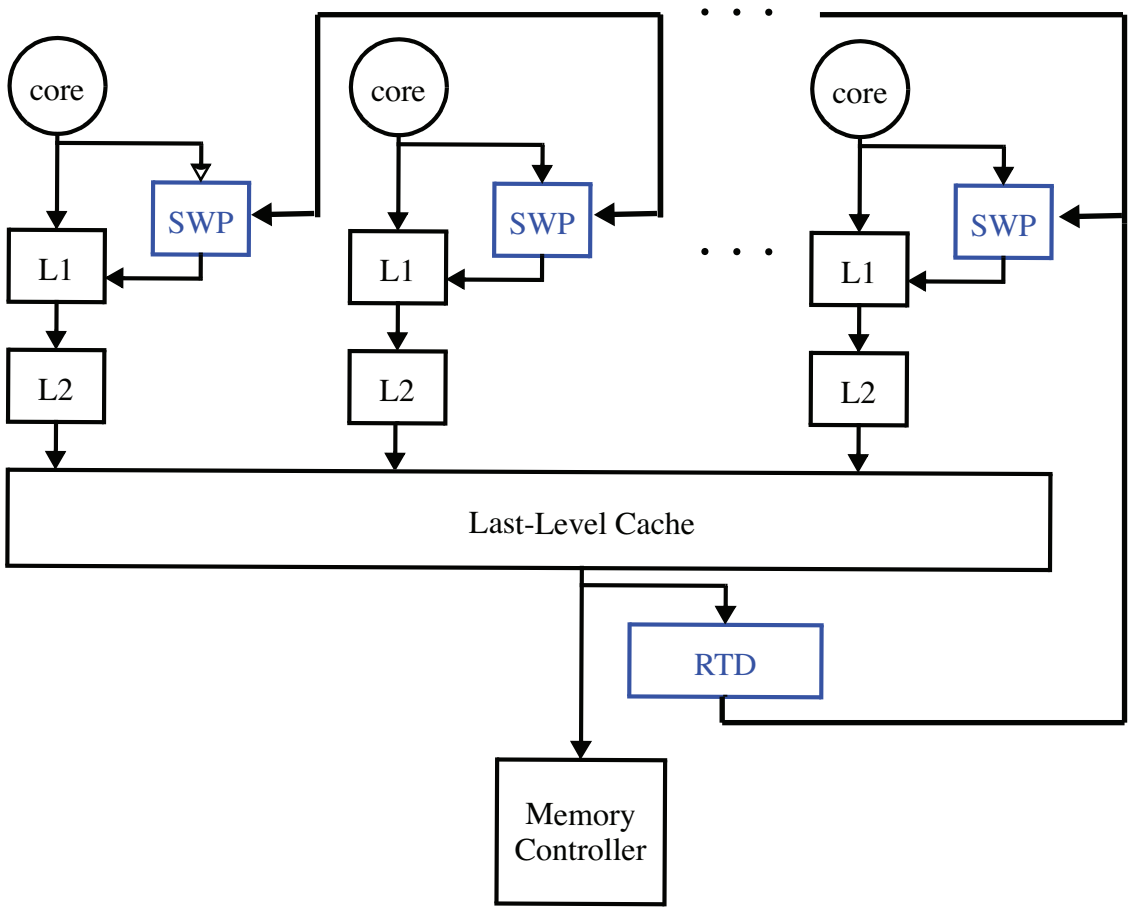

Figure 4.1: The hardware modules of the Store Reuse Time Predictor (SRTP) and their input-output paths.

the cache tags in SRTP. As discussed in Section 3.3, SRTP observes the overwrite time between writes to main memory and associates it to the store instruction generating the writes. This allows for making accurate soft write decisions for all future writes by those store instructions. To collect the last touch store instruction and make it available to SRTP, we extend cache tags in the L1, L2, and LLC with a store PC field. Each time a store hit occurs to a tracked cache block in the L1, we write the instruction's address into this field. Multiple store hits to the same L1 block simply overwrite each other's store PC. As dirty L1 cache blocks are evicted, they transfer their store PC to the L2 and then to the LLC. Thus, on an LLC writeback, the store PC field will have the last

store instruction to hit the cache block prior to the writeback, *i.e.*, *the last-touch store*.

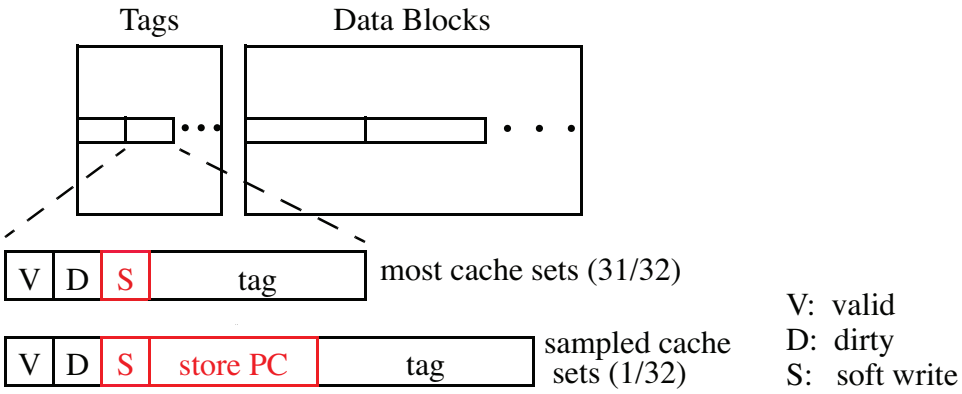

Figure 4.2: Additional state added to caches (L1, L2, LLC) shown in red.

One issue with this approach is the high overhead of adding a PC field to the tags of all cache blocks. For the cache sizes simulated in this work from Table 5.1, the PC fields would require 1.68 MB. Such a high space overhead of more than 10% of the LLC size is unreasonable. We lower the tracking overhead of the last-touch stores in two ways. First, since the actual number of distinct last-touch PCs is very small, it is unnecessary to track all 48 bits. We only track the 20 least significant bits of the PCs allowing for some aliasing to occur in RTD. The instances of aliasing are so infrequent that they do not adversely affect the results, as can be seen in Chapter 6, where SRTP keeps up with the Oracle. Even with reduced bits per PC, the overhead is still too large at 700 KB. Second, we use *dynamic set sampling* (DSS) [47, 71] to further reduce the storage overhead. The key idea behind DSS is that the behavior of the cache can be approximated by sampling only a few sets. We noted in Section 3.3 that for a particular last-touch PC, the overwrite time is fairly consistent across all the cache blocks written by it. So, only sampling a few cache blocks should suffice to capture this dominant overwrite time. In this work, we sample one out of every 32 cache sets. This further reduces the space overhead of the PC fields in the cache to a more manageable 22KB (0.14% of LLC). Section 6.3.1 evaluates the

impact of sampling on SRTP's performance.

Another modification to the cache fields is the addition of a *soft write bit* (S) to the tags for tracking the soft write decisions corresponding to each dirty block. As we will see in Section 4.3, each write from a core is accompanied by the SWP's prediction for the soft write decision corresponding to it. The S bit is set to 0 or 1 in L1 when its corresponding cache block is written, and SWP predicts hard or soft write, respectively. The S bit is propagated to the lower cache levels as the blocks are evicted from L1. Eventually, when the cache block is written back to the main memory, it uses soft or hard write based on the final value of the S bit at the time of eviction from the LLC. While we can use set sampling for the last-touch store PC field since it is only involved in training, the S-bit must be carried in all cache tags so that every dirty cache block can be written back to main memory with the appropriate write strength (either soft or hard).

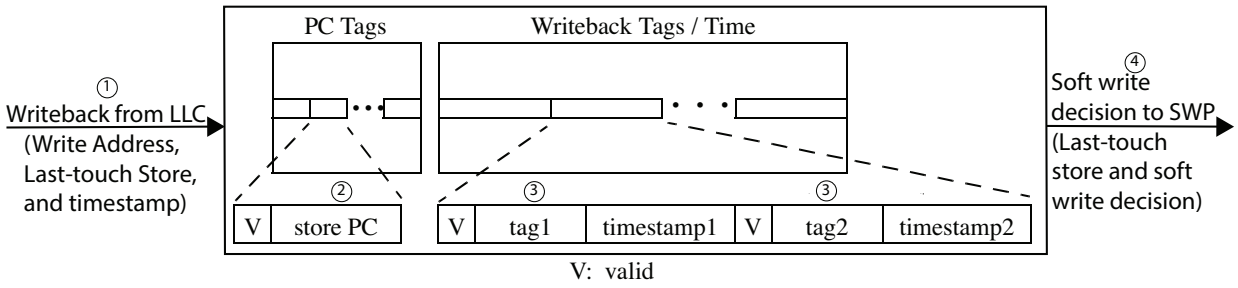

Figure 4.3: Reuse Time Detector (RTD) module components, inputs, and outputs.

4.2 Reuse Time Detector (RTD)

The RTD module and its components are illustrated in Figure 4.3 and Figure 4.4 summarizes the functioning of RTD via a flowchart. It is an associative buffer that performs training for the SWP modules by observing reuse times (or overwrite times) between LLC writebacks from the same last-touch store instruction. As specified in the previous section, caches are enhanced to

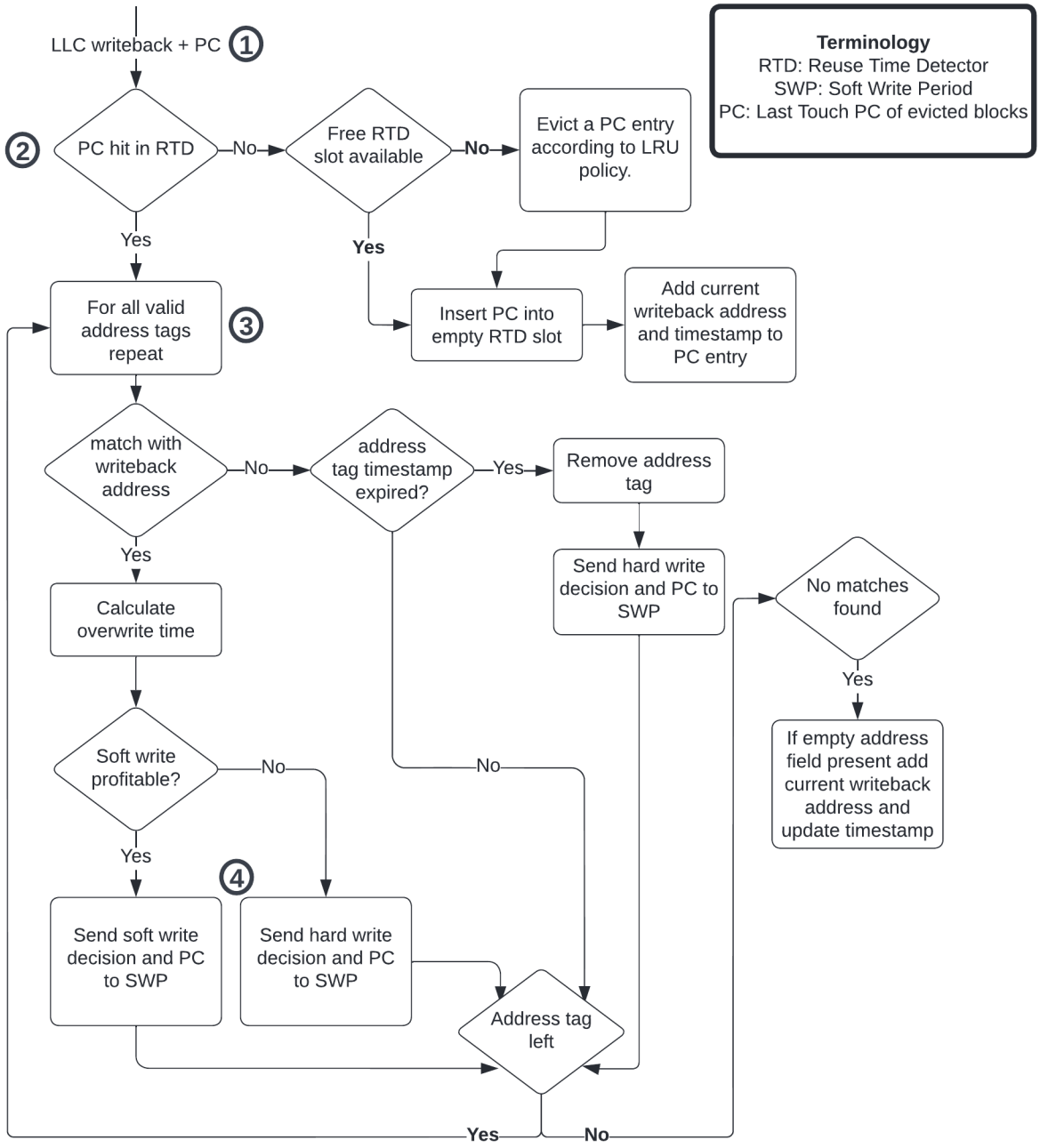

Figure 4.4: Reuse Time Detector(RTD) flowchart.

track the PC of the last-touch store instruction of sampled dirty blocks. When a dirty LLC cache block writes back to the main memory, it is also passed as an input to the RTD along with its last-touch store PC and the timestamp at eviction (label ① in Figures 4.3 and 4.4). The RTD

access happens in parallel with sending the writeback to the main memory. So, it does not delay the writebacks, unlike RRM, which adds its access latency to writebacks.

RTD entries are tagged with the PC of the store instruction associated with the writeback. The last store PC of the incoming writeback is matched against the store PCs already stored in RTD (label ② in Figures 4.3 and 4.4). In case of a miss, the incoming PC is inserted into the RTD, and evictions are handled using the least recently used (LRU) policy. If a match occurs, RTD moves on to compare the current writeback address against the address tags in the matched entry (label ③ in Figures 4.3 and 4.4). (Notice, with both store PCs and memory blocks as tags, an RTD lookup involves two matches: the store PC is used to both index the RTD and match one of its ways, and then the memory block address is used to match one of the data tags associated with the store PC). If no matching address tag is found, the writeback block address is filled into the RTD as long as a free entry exists. PC miss will create a new entry with no address tags, which results in the writeback address always being added. Whenever a block address is added to a tag field, the time at which the writeback happened is filled into the timestamp field adjacent to the tag. Later on, if the same memory block is written back again, a match in the RTD will occur, allowing the elapsed time between the writebacks to be computed by subtracting the previously stored time from the time on the matching lookup. This writeback-to-writeback reuse time or overwrite time can then be used to evaluate Inequality 1.1 or Inequality 2.15, thus determining whether the original writeback should have been a soft write or hard write. Whenever the overwrite time is calculated, RTD has an output for the soft write decision. This decision can be linked with a static store instruction from the last-touch store PC associated with the RTD entry producing the soft write decision. The output of the RTD is then used to train SWP modules by giving them a store PC value and corresponding soft write decision. This is indicated by label ④

in Figures 4.3 and 4.4. Section 4.3 will discuss this training procedure in more detail.

A writeback from LLC is directed to RTD only if it is from a sampled set described in the previous section. Not only does this reduce the tracking overhead in the cache, but also greatly simplifies the design of the RTD module, as its input bandwidth only needs to be a fraction of the write bandwidth from LLC. For instance, RTD only needs to support $\frac{1}{32}$ of the LLC write bandwidth in this work. Because of this, the designer would be able to avoid many bandwidth-improving techniques like pipelining, multi-banking, non-blocking accesses, etc., which significantly increase the complexity of the control circuits [72].

One reason why soft write decisions are so challenging to learn is that overwrite times can be extremely long, on the order of multiple seconds. In that length of time, a significant number of dynamic store instructions forming a huge number of writeback reuses will occur. Fortunately, the RTD module only needs to track a tiny fraction of these events for a couple of reasons. First, the majority of dynamic last-touch stores can be traced back to a few static store PCs. As stated in Section 3.3, for SPEC CPU2017 benchmarks, 90% of writes can be attributed to 76 last-touch store PCs. Thus, the RTD only needs to keep track of a few store PCs. For the SPEC CPU2017 benchmarks, we find that 512 store PCs (*i.e.*, RTD entries) are sufficient to capture the "working set" of static store instructions responsible for most last-touch stores. To minimize conflicts, we employ 16-way set associativity in the RTD.

Second, as discussed in Section 3.3, writebacks associated with the same static last-touch store instruction tend to exhibit a single dominant overwrite time thanks to store correlation. Thus, RTD does not need to track all the lines written by a store PC, and sampling a small number of writeback-to-writeback reuse events suffices. However, sampling too few reuse events might also be problematic. Since overwrite times can be very large (multiple seconds), it takes

RTD some time to learn those. This allows some soft write opportunities to slip through while RTD is trying to train SWP. So, tracking multiple memory blocks per store PC is beneficial as it allows for faster training of SWP. The baseline RTD implementation tracks two memory blocks per store PC, as shown in Figure 4.3. This makes it possible to detect two overwrite times for each store PC at a time.

RTD uses replacement policies for the PC tags that are akin to caches. We employ the LRU scheme, but other policies like RRIP, SHIP, and Hawkeye are also viable alternatives. However, a different approach is necessary for the management of each PC's address tag entries. Due to the long duration of writeback-to-writeback reuse events, the memory block tags must be allowed to linger in the RTD for a long time. Hence, *we never evict memory block tags* as long as the PC entry is not evicted. Instead, only the following three circumstances cause the address tags to be taken off.

1. Should the last-touch store PC entry be evicted. In this instance, its corresponding address tags and timestamps are also removed to make room for the incoming PC entry. This case does not happen frequently as the number of last-touch PCs is small enough that RTD is able to accommodate all of them.

2. When a matching lookup takes place for address tags, and an overwrite time calculation is completed. This may not necessarily improve the training speed, but it makes the training process more robust to outliers. For example, if an address tag was not evicted once it yields an overwrite time value, then RTD will continue to use it for future calculations too. There is a small chance that the address is an outlier and its overwrite time is different from the dominant peak of the corresponding last touch store instruction. Consequently, keeping the

same outlier address tag around would mean that RTD would keep on feeding the wrong training data to SWPs. Evicting the address tag once it yields an overwrite time candidate would make the space available for other addresses. Given the majority of addresses will have the dominant overwrite time, the likelihood of consecutive addresses being outliers is very small. This allows the RTD to adapt to learning wrong overwrite time values.

3. Even if the address tag does not match, RTD also detects when a tag's timestamp expires- i.e., the timestamp becomes so old that Inequality 1.1 or Inequality 2.15 (whichever one is being used) is guaranteed to be false even though we have yet to see an overwrite occur. In that case, we conclude that hard write is the correct decision and eagerly remove the tag, saving the need to wait for the time remaining before the next writeback reuse actually occurs. This check can be triggered on an address tag miss.

4.3 Soft Write Predictor (SWP)

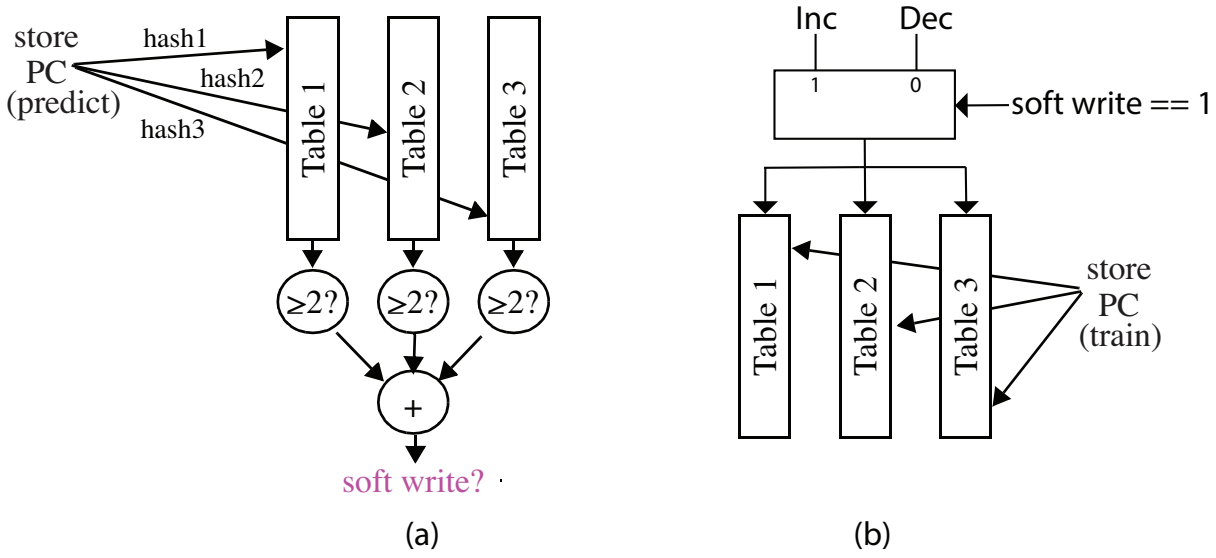

Figure 4.5: Soft write predictor module (a) prediction (b) training.

Each SWP module, illustrated in Figure 4.5, is a predictor that makes soft or hard write predictions for every dynamic store instruction executed in one of the CPU's cores. There are multiple SWP modules in the CPU–one for each core, as shown in Figure 4.1. A core accesses its local SWP module when it executes a store instruction to predict the type of write to perform. SWP's soft write prediction must be recorded in the caches in order for the appropriate type of write to be executed upon LLC eviction. To do so, it updates the soft-write bit or S-bit in the tags array of the L1 cache, which was covered in Section 4.1.

SWP needs to update the soft write bit by the time the store instructions finish, so it inadvertently becomes part of their critical path. SWP accesses must execute with a latency that is equal to or lower than the fastest execution of store instruction, *i.e.*, a write hit in the L1, in order to avoid slowing down the execution of programs. The write hit operation in the L1 cache can be broken down into two steps. The first step is tag lookup, which checks if the block being written is already present in the cache, and the second is data update, which writes to the block if it is found in the cache. SWP's soft write prediction needs to be done by the time the L1 cache's tag lookup finishes. Then, S-bit can be updated in parallel to the cache block data modification. This rules out using an SWP structure that is similar to a cache because doing so would force the SWP cache to complete a full access (look up and read) in the same time as the L1 cache's tag lookup. Assuming we are using the fastest implementation for the L1 cache, this would imply that SWP's cache is even faster because it can complete the entire access in the L1 cache's lookup stage, which contradicts the assumption. The scenario where the SWP cache misses and the L1 cache hits would be even worse, possibly stalling the store instruction while the SWP cache does an L2 or LLC lookup.

For predictions faster than tag lookups, we adapt SWP from the skewed predictor organi-

zation of the 'gskew' branch predictor [73]. Here we only have a table that stores the confidence scores for each PC entry, and the lookup is replaced by indexing using a hash function. There has been significant work in fast and effective hash functions that entail simple bit manipulation operations. Aliasing, where different PC values index to the same entry, is a problem with this scheme and could interfere with training and prediction. To mitigate aliasing, three separate tables are employed in each SWP module, with each table indexed using a different hash function. Additionally, the functions are also chosen to minimize the likelihood of collisions in all three tables for any pair of unique PC values. Section 4.3.1 explains the indexing functions used in SWP, and Section 4.3.2 goes over the implementation details of the whole module.

4.3.1 SWP indexing

We employ the hash functions from the gskew branch predictor and the skewed associative caches [73, 74]. The objective is to take information, like store instruction PC values in this case, and hash it to create multiple indices such that collisions between different information values are kept to a minimum. Consider the incoming binary representation of a PC value is a vector of bits, 'P.' This vector can be further split into three bit strings (P_3, P_2, P_1). P_1 comprises the 'n' least significant bits of P, P_2 has the n bits after that, and P_3 is made up of the remaining bits in P, where 2^n is the total number of entries in each table being indexed. Consider the following definition of a function *H*, which operates on bit strings of length n ($y_n, y_{n-1}, \ldots, y_2, y_1$):

$$H : 0, 1, \ldots, 2^n - 1 \rightarrow 0, 1, \ldots, 2^n - 1$$

$$H(y_n, y_{n-1}, \ldots, y_2, y_1) = (y_n \oplus y_1, y_n, y_{n-1}, \ldots, y_3, y_2)$$

$$H^{-1} : 0, 1, \ldots, 2^n - 1 \rightarrow 0, 1, \ldots, 2^n - 1$$
$$H^{-1}(y_n, y_{n-1}, \ldots, y_2, y_1) = (y_{n-1}, y_{n-2}, \ldots, y_2, y_1, y_n \oplus y_{n-1}) \tag{4.1}$$

Essentially, H involves one bitwise exclusive OR operation($\oplus$) between the most significant and least significant bits and then shifting the input right to insert the result of the XOR operation as the most significant bit. It is a fairly simple and fast operation. The inverse operation, H^{-1}, is equally straightforward, as shown in Equation 4.1. H can be inverted by performing a bitwise XOR operation between the two most significant bits and then shifting the input left to insert the result of the XOR operation as the least significant bit. These two functions are used to define three mapping functions, $f_1, f_2, \ and \ f_3$, that index into different SWP tables as follows:

$$\begin{aligned}
f_1 &: P \rightarrow 0, 1, \ldots, 2^n - 1 \\
f_1(P) = f_1(P_3, P_2, P_1) &= H(P_1) \oplus H^{-1}(P_2) \oplus P_2 \\
f_2 &: P \rightarrow 0, 1, \ldots, 2^n - 1 \\
f_2(P) = f_2(P_3, P_2, P_1) &= H(P_1) \oplus H^{-1}(P_2) \oplus P_1 \\
f_3 &: P \rightarrow 0, 1, \ldots, 2^n - 1 \\
f_3(P) = f_3(P_3, P_2, P_1) &= H^{-1}(P_1) \oplus H(P_2) \oplus P_2
\end{aligned} \tag{4.2}$$

An important property of these functions is that if two distinct vectors A(A_3, A_2, A_1) and B(B_3, B_2, B_1) map to the same entry in one of the tables, they will not conflict in the other two tables if $(A_2, A_1) \neq (B_2, B_1)$. This significantly mitigates the impact of conflicts in SWP. Also, for a PC bit vector P, only the least significant 2n bits are used by the hash functions to calculate the indices, where 2^n is the size of each table being indexed.

4.3.2 SWP Implementation

The internal structure of SWP is illustrated in Figure 4.5(a,b). SWP has three tables, each containing a column of 3-bit saturation counters. Store PC is passed through the hash functions from Equation 4.2 to calculate a different index per table. The baseline implementation in this work has 1024 entries per table. So, only the 20 least significant bits of a PC are used by the hash function. Apart from the space savings, this is another reason why we only track 20 bits of PC in the sampled sets of cache, as described in Section 4.1.

Training of SWP modules is shown in Figure 4.5(b). As the RTD computes soft write decisions, it trains the SWP by looking up the three counters associated with the tracked store PC and incrementing/decrementing each one of them for soft write/hard write decisions. In the beginning, all the counters are initialized to 0, corresponding to a hard write prediction, but flip to a soft write prediction as the count reaches a certain "soft write prediction threshold" (T). The SWP modules are at the same level as the L1 cache, and RTD resides near the LLC. This allows the training data from RTD to SWP to be transmitted using the cache interconnect, obviating the need to set up separate communication channels for it.

Simultaneously with training, each CPU core looks up the SWP whenever a store instruction executes using the store's PC, illustrated in Figure 4.5(a). Each of the three counters in the predictor associated with the store PC is then compared to the threshold value (2) to determine whether the counter reflects a soft or hard write prediction. A majority vote across the three counters determines the final soft write prediction for the store. (By using all three counters, destructive aliasing in one table will not negatively affect the final prediction).

The SWP's soft write prediction must be recorded in the caches so that the correct type of

write can be performed upon LLC eviction. We add a *soft-write bit* or "S-bit" to the tags in all the caches. The SWP writes the S-bit in the L1 tag with its prediction. The S-bit is then copied into the L2 and LLC tags as the cache block is evicted. When the dirty blocks are removed from the LLC, the S-bit regulates the strength of the write to main memory.

4.4 SRTP operation

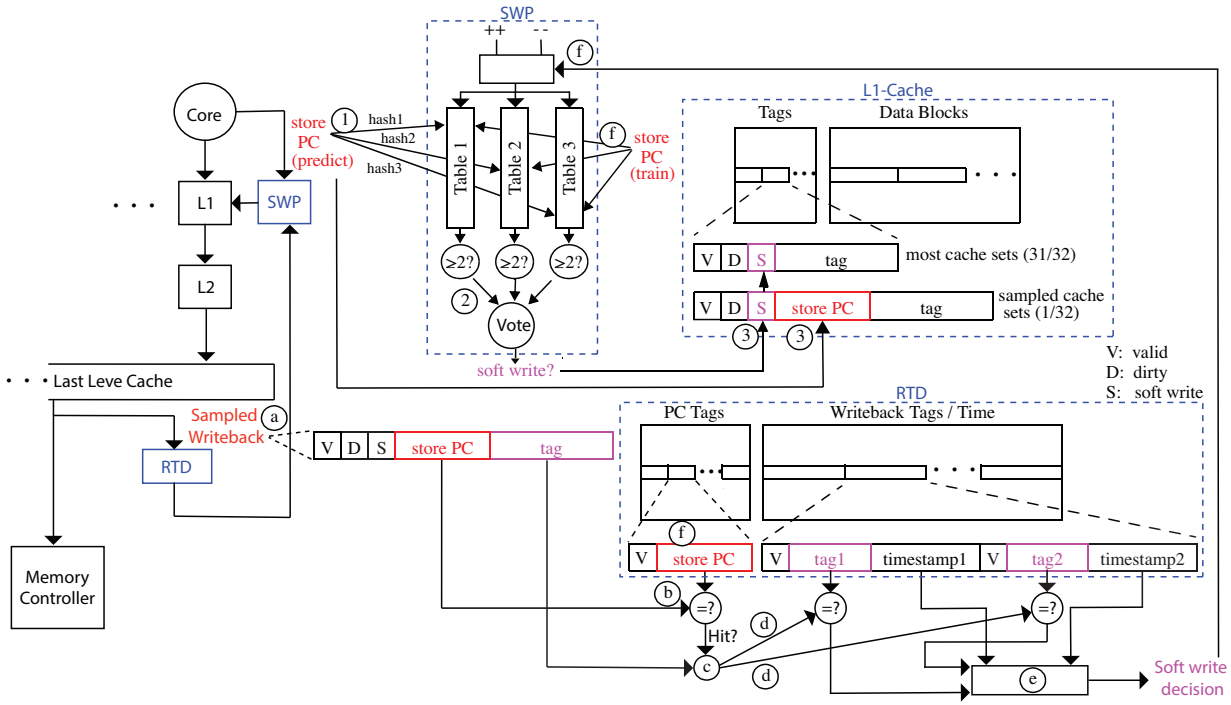

Figure 4.6: Interaction of SRTP modules.

Now that we have discussed the individual roles of the SRTP components, the next step is to examine how they function together. Figure 4.6 shows the components of SRTP along with how they interact with each other. Prediction, training, and refresh are the three main pathways within SRTP mechanisms. The first two are situated on the CPU die and interact with each other. Refresh, on the other hand, is relatively isolated on the main memory side. So we discuss it in the following section with wear leveling. The prediction pathway is primarily handled by the top

half of Figure 4.6, while training resides in the bottom half.

The prediction pathway begins at the Core when it executes a store instruction (label ① in Figure 4.6). The store PC is used to index into the tables of the local SWP module. Depending on whether or not their counter values exceed the soft write threshold (2), the tables either support or oppose soft write (label ② in Figure 4.6). The majority vote is taken into account when writing the soft write bit (S) in the L1 cache for the data that is being modified. At the same time, if the cache set is a part of dynamic sampling, the store instruction's PC value is also recorded (label ③ in Figure 4.6). On eviction from L1-cache, the blocks carry this information along to the lower levels of the cache hierarchy. The soft write bit determines the type of write that must be done when the block is finally evicted from the last level cache, and the stored PC value helps with the training pathway.

The training mechanism is activated when a writeback from a sampled block is issued from the last level cache. The relevant tag information is sent to the RTD from these writebacks for overwrite time estimation (label ⓐ in Figure 4.6). The store PC from writeback is compared against the RTD entries first (label ⓑ in Figure 4.6). A new entry is allocated in the RTD on a store PC miss, as described in Section 4.2. The remaining portion is triggered only if a store PC hit occurs (label ⓒ in Figure 4.6). The memory address of the writeback is compared against the address tags stored in the RTD entry with matching store PC (label ⓓ in Figure 4.6). On an address match, the timestamp of the matching address tag and current time can be used to estimate overwrite time. The overwrite time, in turn, gives the appropriate soft write decision for the store PC (label ⓔ in Figure 4.6). The store PC value and soft write decision are communicated to SWP modules from RTD for training (label ⓕ in Figure 4.6). Depending on whether soft or hard writes were deemed appropriate, the relevant counters are incremented or decremented in

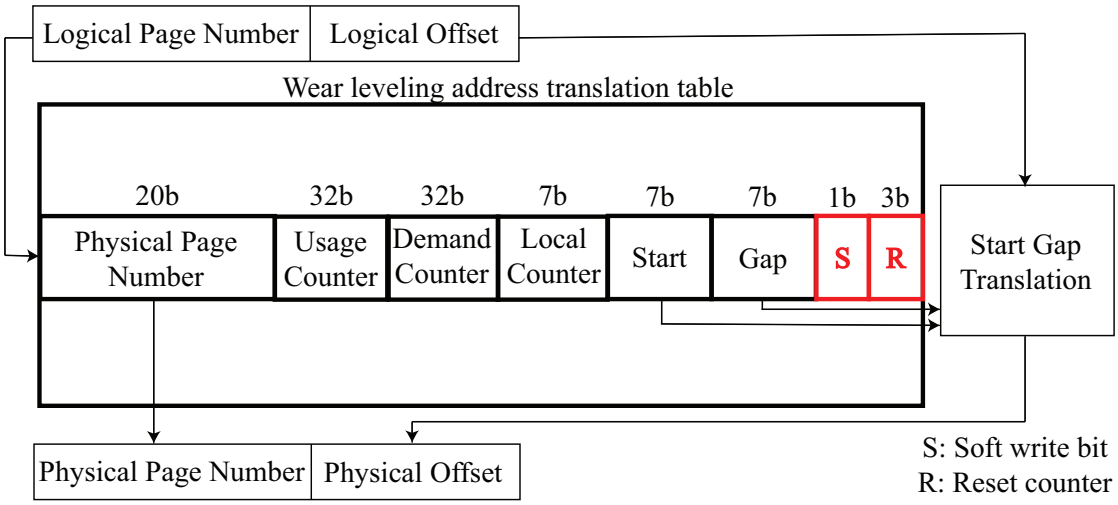

Figure 4.7: Wear leveling table with added fields (in red) for SRTP refresh and reset operations.

the SWP (label ⓕ in Figure 4.6). At the same time, the right counters are selected by indexing into SWP using the store PC from RTD (label ⓕ in Figure 4.6). Training of SWP modules begins only when a soft write decision is confirmed. For that, a match must occur for the store PC and address tag in the RTD.

4.5 Refresh, Resets, and Wear Leveling

In addition to soft write prediction, SRTP must also track the memory pages receiving soft writes for refresh and reset purposes. As our results will show, SRTP performs many more soft writes than RRM, which increases the number of pages that must be tracked. We find SRTP can have 100s of thousands of softly written pages (compared to just 4K or 32K pages for RRM-base and RRM-aggr). Due to its size, we implement the tracking data structure in the main memory.

The tracking data structure contains two fields. First, there is a "soft write bit" for each page that indicates whether the page has been softly written. This bit is similar to RRM's hot bit in that it identifies the pages requiring refresh operations. When a page receives a soft LLC writeback,

its soft write bit is set. The memory controller checks the soft write bits of a few pages every ten μsec such that all pages are checked by the end of a retention period. A refresh command is issued for the pages with their soft write bits set. When performing refresh, SRTP refreshes the entire page–it does not further distinguish softly written memory blocks within a page as RRM does to minimize the tracking overhead. Another advantage of moving the refresh tracking to the main memory is the ability to refresh a small group of pages at a time. As mentioned in Section 3.1, RRM issues refresh requests for all softly written pages together near the end of each retention period. This introduces a burst of requests suddenly into the memory system, resulting in contention, which will degrade performance. SRTP can avoid these bursts by issuing a small chunk of refresh requests at regular intervals within a retention period, thus handling the refreshes without flooding the request queues at the controller.

SRTP's tracking data structure also contains a 3-bit "reset counter" per page. This counter is used to determine when softly written pages stop receiving soft writes. It is incremented when the page is refreshed but is reset to zero after every 100 soft writes to the page. If the page stops receiving soft writes, then the count will continue to rise, signifying that refreshes are being incurred without any soft write benefit. When the count reaches a threshold (SWA), SRTP performs a reset hard write: it issues a hard write to every memory block in the page, clears the soft write bit, and stops refreshing the page. The refresh threshold should be chosen carefully to avoid resetting the pages while they are receiving soft writes. If the value is too small, then the pages with overwrite time values larger than the retention time times reset threshold might be issued resets even though they are still receiving soft writes. This might create a ping-pong effect where the soft write bit is continuously set and reset, amplifying the writes. The value cannot be too large either, as that will lead to a lot of unnecessary refreshes once the page stops getting

soft writes. Keeping the threshold at SWA is the appropriate trade-off. It will cause the resets just beyond the overwrite time threshold, which is the bare minimum number of final refreshes to avoid reset amplification.

To implement the soft write bits and reset counters, a separate standalone table could be created in the main memory. However, we propose to implement SRTP's tracking data structure as part of an integrated wear leveling scheme. Wear leveling is still a problem in soft writable memory systems (soft writes do not affect how the writes are distributed across memory). Yet, to our knowledge, no prior soft write technique has considered integration with wear leveling.

Section 2.3 covered different wear-leveling techniques employed with emerging NVM. In this work, we consider Ouroboros; a hybrid wear-leveling technique. It divides the physical memory into regions and uses table-based wear-leveling across regions. Within a region, Ouroboros uses Start-Gap to provide wear-leveling locally. Figure 4.7 shows the translation table for Ouroboros with the additional bits for handling SRTP's resets and refresh in red. The extension of wear-leveling to handle refresh and resets only incurs a 3.8% increase in the table size.

For each incoming memory request, the translation table remaps it to the actual location of data to take into account the remapping done by wear-leveling. First, address bits corresponding to the logical page number are used to index into the indirection table. The corresponding entry in the table has the physical page number where the data is located. This physical page number replaces the logical page number in the address. Then the logical offset is converted into a physical offset using the start and gap registers of the page entry. For writes, all the relevant counters are incremented, and required migrations are performed as described in Section 2.3.3. The work needed for SRTP integrates well with the local start-gap scheme. There is no need

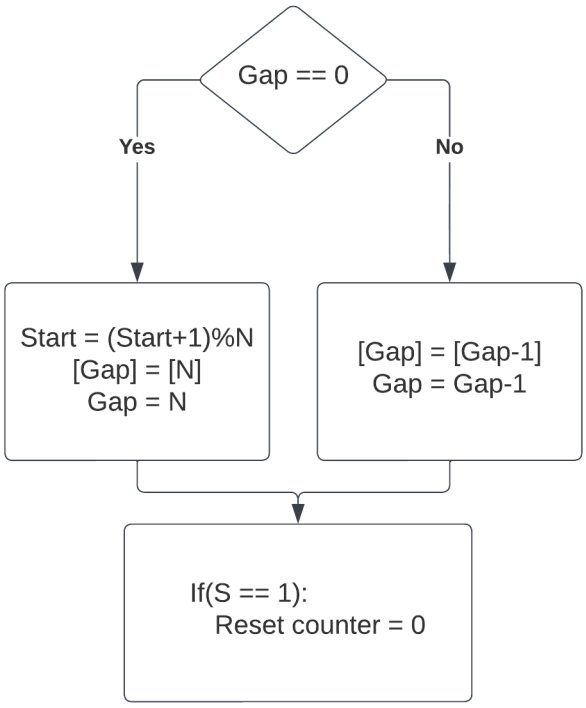

Figure 4.8: Flow chart of Gap Movement operations extended to accommodate the resetting of reset counter for SRTP.

to separately count writes for setting the reset bit to zero. The local counter is already doing that, and we can extend the gap-movement circuitry to set the reset bit to 0. Figure 4.8 shows the extended flowchart for the gap movement. If the memory request is a soft write, then the soft write bit (S) can be set while translating the address. The refresh circuitry iterates over the translation table, issuing refreshes to pages whose S bit is set.

Chapter 5: Methodology

We use simulations to evaluate our technique against RRM and the Oracle. The timescale of events we are looking at differentiates this work from the majority of previous work. Architectural simulators are typically used to study events that exhibit very short time horizons. In this work, we are interested in studying ReRAM retention events that can span several seconds. Hence, high-speed simulation is essential for our evaluation. At the same time, cycle accuracy is important too since we need to faithfully model the overwrite times that drive soft write decisions.

We use the Zsim multicore simulator [75] for our evaluation. Currently, Zsim is one of the fastest simulators available and can be used to study the time scales demanded by our research. Zsim also provides detailed architecture models that enable cycle-accurate simulation with validated out-of-order (OOO) core models. Table 5.1 reports the CPU parameters used in our simulations. We configure the simulator to model a quad-core CPU with out-of-order cores running at a 2 GHz clock. The system also has a 3-level cache hierarchy. Each core has private 32 KB, 4-way instruction and data L1 caches with 3 and 4 cycles hit latencies, respectively. This is followed by private 256 KB, 8-way L2 caches with 7 cycles latency for hits. The L3 cache is shared between all cores at 16 MB size, 16 ways, and 27 cycles hit latency. All caches implement the least recently used (LRU) replacement policy and a MESI coherence protocol with an in-cache directory at the LLC.

CPU	
Number of cores	4
Core	x86, out-of-order, 2GHz
L1-I Caches	Private, 32 KB, 4-way, LRU, 3-cycle hits
L1-D Caches	Private, 32 KB, 4-way, LRU, 4-cycle hits
L2 Caches	Private, 256 KB, 8-way, LRU, 7-cycle hits
L3 Cache	Shared, 16 MB, 16-way, LRU, 27-cycle hits
cache block size	64 B
Main Memory	
Memory	667MHz, 8 KB page size, 8 GB capacity, closed page, FCFS scheduling
Channels	4
Ranks per channel	4
Banks per rank	8
tRCD (read pulse)	140 cycles (210ns)
tWR (write pulse)	280 cycles (420ns)
tRFC (refresh pulse)	420 cycles (630ns)
Endurance	2×10^6 hard writes per bitcell
Read Energy	2 pJ/bit
Hard Write Energy	30 pJ/bit
Soft Write Energy	3 pJ/bit
Refresh Energy	5 pJ/bit
Soft write retention time	10 seconds
Wear Leveling	Local threshold 100 writes, global threshold 10^8 writes, 10 migrations/global epoch, 128 spare frames

Table 5.1: Simulation parameters.

Something lacking in Zsim is models of non-volatile memory. While there exist NVM simulators, none of them are fast enough to be integrated into Zsim; doing so would completely sacrifice the ability to study ReRAM retention events. Fortunately, Zsim exposes many detailed memory timing parameters that can be configured. We set these parameters to match models available in NVMain [76], a recent non-volatile memory simulator, to accurately reflect the timing characteristics of a ReRAM-based main memory system. The major changes are related to the asymmetric read / write latencies and local refresh from soft writes. We adjust the duration of read pulse (tRCD) and write pulse (tWR) to take those into account. We also introduce a local refresh operation whose latency is guided by the tRFC parameter. This operation refreshes row buffer size worth of cells at a time in a bank. Refresh requires reading the data to be refreshed into the row buffer and writing it back. As a result, the refresh latency/energy is the sum of both read and write latencies/energies. Since the row buffer will be unavailable during the refresh operation, the memory bank it corresponds to should be blocked for the operation. We update memory models in Zsim to keep the whole bank occupied while part of it is being refreshed. While a particular bank is executing a refresh command, memory requests can be serviced by other banks that do not share the row buffer.

Many different latency, energy, and endurance numbers have been reported for ReRAM in the literature. Without loss of generality, we use the following values for these parameters in our evaluation. We use a base latency (without contention) of 210ns and 420ns for reads and writes, respectively [77]. These latencies are introduced into Zsim memory models through the timing parameters (tRCD and tWR), as discussed earlier. Read energies in the range 1-2.4 PJ/bit [77, 78] and write energies in the range 4.8-53 PJ/bit [77, 79–81] have been reported. We use 2 PJ/bit and 30 PJ/bit for the read and hard write energies, respectively. Chapter 2 covered that soft write

energy is 10x less than hard write energy. So, we have soft write energy at 3 PJ/bit. Also, the refresh operation involves a read and a soft write leading to 5 PJ/bit energy consumption. Lastly, endurance values between 10^5 and 10^7 have been published [35, 82–85]. We use an endurance of 2×10^6 hard writes per bit cell. The bottom portion of Table 5.1 reports all the timing, energy, and endurance parameters employed in the memory system.

The main modification we made to Zsim is to add support for our SRTP technique. Specifically, we added the RTD and SWP modules described in Chapter 4. A single RTD module is added between the last level cache and the main memory, as shown in Figure 4.1. This module gets the writeback information from the LLC when dirty blocks are evicted from sampled sets. We also implemented one SWP module per core, which resides right next to its L1 cache. By having RTD closer to LLC and SWP near the L1 cache, these modules can use the already existing interconnect in the cache hierarchy to communicate. We also added the RRM hardware discussed in Section 3.1–and modeled two versions of RRM–so that we can compare SRTP against this prior state-of-the-art technique. (The RRM and SRTP parameters are given in Tables 5.3 and 5.4 and will be discussed later in this section). Lastly, to enable a limit study, we implemented the Oracle policy from Section 3.2. This requires allocating a separate region of memory to shadow the main memory, providing space for the per-memory block timestamps that the Oracle requires.

To drive our simulations, we use the SPEC CPU2017 benchmarks [70]. These are standardized kernels developed from real world applications designed to compare compute-intensive performance across hardware platforms. The benchmark suite consists of 23 programs derived from 10 integer and 13 floating point applications. We employ 19 of these benchmarks in our study except for `gcc`, `cactuBSSN`, `povray`, and `exchange2`. While povray and exchange2

are excluded as they have barely any writebacks making soft writes unnecessary, gcc and cactuBSSN fail to run on Zsim. Table 5.2 lists the benchmarks in order of their memory intensity. (The benchmarks are sorted on column two of Table 5.2, which reports the number of LLC writebacks per 1000 instructions, or "WPKI"). We compile the benchmarks using the Gcc compiler with '-O3' flag, which is the highest level of optimization. In every case, we run the benchmarks to either completion or up to 8 trillion instructions, whichever comes first. On average, the benchmarks were simulated for 6.5 *trillion* instructions and exhibited a wall clock time of 1,584 seconds (about 26 minutes). This allows us to observe numerous ReRAM retention time intervals for each benchmark, leading to statistically meaningful results. For our wear leveling experiments, we further extrapolate multiple runs of the benchmarks to determine actual lifetimes. Chapter 7 will describe these extrapolation experiments.

In our experiments, we study both homogeneous and mixed workloads. For the former, we run the same benchmark on all 4 cores of the CPU. This allows us to study the effectiveness of our SRTP technique given each benchmark's unique access patterns, yet, it provides a memory access rate appropriate for a multicore memory system. Mixed workloads, covered in Section 6.1.4, run 4 different workloads together.

5.1 Hardware Cost

Before presenting the results, we compare the hardware costs for RRM and SRTP. Table 5.3 shows the configuration parameters for two versions of RRM that we evaluate. "RRM-base" has 4,096 entries (256 sets $\times$ 16 ways), requiring a 96 KB table. This is identical to the hardware overhead from the original RRM paper [8]. "RRM-aggr" is a more aggressive version of the

Benchmark	WPKI	Inst (T)
lbm	17.8	7.1
fotonik3d	11.4	6.1
omnetpp	5.5	4.2
roms	3.84	8
mcf	3.49	3.9
cactuBSSN	2.25	6.1
wrf	1.86	8
bwaves	1.62	6.6
cam4	1.35	8
xz	0.79	2.3
xalancbmk	0.76	5.1
deepsjeng	0.43	7.4
nab	0.36	8
parest	0.3	8
namd	0.21	8
perlbench	0.17	2.6
leela	0.13	8
blender	0.11	7.1
x264	0.05	7.7
imagick	0.005	8

Table 5.2: Benchmarks used in the study sorted by Writebacks Per Kilo Instructions (WPKI). Simulated instruction counts are reported in trillions.

scheme with 8x more entries–32,768 entries– and 8x the table size–784 KB. This allows the aggressive scheme to track more pages and unearth more soft write opportunities. It should be noted that these tables occupy precious silicon real estate on the CPU die. Overheads comparable to cache sizes are not very realistic, and hence keeping the on-die overhead within 1% of LLC size ($<$160KB) is very important. RRM-aggr is not a practical solution because of this. We simulate it to demonstrate the effectiveness of our scheme.

While RRM adds a single monolithic table, SRTP adds two hardware structures, the RTD and SWP modules, and tracking in the cache tags. These are the structures residing on the CPU die, and they amount to 69.8 KB which is well within the overhead bounds. Table 5.4 shows the breakdown of the overheads for each component. SRTP also requires 512 KB to track refreshes

RRM-base		RRM-aggr	
Sets	256	Sets	2048
Associativity	16	Associativity	16
Hot threshold	4	Hot threshold	4
Decay interrupt	100/16 seconds	Decay interrupt	100/16 seconds
Overhead	**98 KB**	Overhead	**784 KB**

Table 5.3: RRM parameters and overhead.

and resets. This table is large because SRTP is very successful at capitalizing on soft write opportunities. Since this table is primarily used to refresh or reset softly written data, depending upon the condition discussed in Section 4.5, it makes sense to keep it closer to the off-chip main memory. Given the information in this table is not frequently accessed, it can be implemented in the non-volatile memory. Moreover, if we integrate SRTP with Ouroboros, the refresh/reset bits can be folded into the wear leveling tables with minimal overhead (see Section 4.5).

5.2 SRTP Modules Area and Energy

In this section, we quantify the area and energy overheads of the SRTP modules. To estimate these parameters for RTD, we use a cache design tool, Cacti [86–88]. It is an analytical memory modeling tool maintained by HP labs that can calculate delay, power, area, and cycle time for various memory technologies. By setting the configuration parameters to be the same as RTD, its area estimates are 0.44mm^2 at the 22 nm technology node. RTD requires 0.18 nJ of energy per access. This translates to 60 μW of average dynamic power for SPEC CPU2017 benchmarks. The leakage power for RTD is 14 mW. The dynamic power is so low because RTD is activated very infrequently. We can compare RTD's activity to the LLC to get a sense of its power consumption. First, it is only accessed on LLC writebacks which are a small fraction of the overall LLC accesses. Second, sampling further cuts down a significant amount of activity as only $\frac{1}{32}$ of

RTD	
Sets	32
PC Associativity (20 bits)	16
Addresses per PC (48 bits)	2
Timestamps per PC (32 bits)	2
Overhead	**11.7 KB**
SWP	
SWP count	4 (one per core)
#Counter banks	3
Saturating counter entry size	3 bits
Soft write threshold	2
#Saturating counter entries	1024
Overhead	**1.1 KB** (4*3*3*1024/8 B)
Caches	
Soft write bit(S)	1 per entry
Last write PC (20 bits)	1 every 32 entries
Overhead	**57 KB** (35+22)KB
Main Memory	
Soft write bit (S)	1 bit per entry
Reset counter (R)	3 bits per entry
Ovhread	512 KB
Total Overhead	**69.8 KB** SRAM and **512 KB** NVM

Table 5.4: SRTP parameters and overheads.

the overall writebacks from LLC will be directed to the RTD.

The estimation of area and energy for SWP is not as straightforward as RTD because it does not resemble a cache structure. So, Cacti cannot be used to perform these calculations for SWP. But, an individual SWP table is identical to the local predictor of the tournament branch predictor [89]. We configure the parameters for Xeon core's local predictor in Mcpat [90] to be similar to a single SWP table, *i.e.*, 1024 entries of 3-bit saturating counters. By extending the results to three tables, we get 0.028 mm^2 area per SWP module at the 22 nm technology node. The peak power, including both dynamic and leakage, for each SWP module is 0.05 W. All four SWP modules combined will occupy 0.112 mm^2 area and consume 0.2 W peak power. It should be noted that the dynamic component of the peak power was calculated using the activity level of the branch predictor, which is different from SWP modules. The average percentage of store and branch instructions in the dynamic instruction mix of the SPEC CPU2017 benchmarks is 8.6% and 14.7%, respectively [91]. As a result, we actually overestimate the dynamic power making the results conservative at best.

The die size of the intel Ivy Bridge at 22 nm, which is comparable to our CPU, is 170 mm^2 [92]. The overall area of SRTP modules at 0.55 mm^2 is just a tiny fraction of the die area at 0.3%. So, the results shown in Chapter 6 are achieved at a tiny overhead area. Even the energy consumption for the module is so low that we can easily ignore it in the memory system energy calculation of Chapter 6.

5.3 Assumptions

Evaluating a work with architectural simulations comes with certain assumptions. In this section, we outline our assumptions and discuss their plausibility to demonstrate the validity of the findings in the following chapters.

OS impact: As specified earlier, being able to simulate longer run times is crucial for this work since we are studying events at larger than usual time scales. Therefore, simulation speed is important, but it also comes with trade-offs. One such compromise is the inability to simulate in full system mode with an operating system. Since simulators like GEM5, which can do so, are very sluggish, it will take them years to simulate the required run time. As a result, we cannot evaluate the effect of context switching handled by the operating system in this study.

Context switching will affect our results in two different ways. First, running different programs together will cause interference in the new hardware modules, and second, program interruptions will affect overwrite time values.

We have experiments that account for the former by running multiple workloads together on different cores, as shown in Section 6.1.4. The accesses of these workloads will interleave with one another as they go through the hardware modules. We show that this has minimal impact on the performance of SRTP.

We do not, however, assess the effects of program interruptions, which can dilate the overwrite time values. This, in turn, would affect the performance of our scheme. Events like exceptions and interrupts are not significant issues as they are not very frequent and usually last for micro-seconds. Since we are interested in reuse time on the scale of seconds, infrequent perturbations on the micro-seconds level will not change the values significantly.

However, context switching could slow down the execution time of the whole program by making it time-share the core with other processes/threads. Depending upon the number of processes sharing the core, this will increase the reuse times for each one of them. If we have a single dominant workload per core, the reuse time remains unchanged for it, and our results are valid. If more than one dominant workload is present, the additional workloads will get time slices of the core based on the scheduling policy. Usually, scheduling policies like a completely fair scheduler (CFS) in Linux will allocate equal time slices to processes weighted by their priority. This will increase the store reuse time for a process by a predictable amount that depends upon its priority and the number of competing processes allowing our scheme to learn it. Effectively, context switching will shift the store reuse time peaks to higher values. This will lead to more refreshes and fewer soft write opportunities impacting all soft write schemes, including the oracle.

The first assumption we make is that either each core has a dominant workload or we have a completely fair scheduler with predictable time slices for each workload. We leave studying the impact of context switch interruption on the soft write policies and how to adapt our scheme to learn the new overwrite time values for scheduling policies other than CFS for future work.

Dual Write: Another assumption is that the hardware for performing dual writes is feasible. In other words, the access circuitry needed to perform two different types of writes at fine granularity can be implemented without a lot of extra overhead. Usually, ReRAM writes already use a write-verify-write technique [93]. Here, they start with a milder write pulse and check the resistance value. If it is not in the desired state, a stronger write pulse is applied. This verify and stronger write pulse operation repeats until the cells reach desired state. This is partially responsible for long write latencies too. Modifying the circuit to stop this process sooner for soft writes should

not be too expensive.

Readability: Soft write will need not only modification to the access circuits but sensing schemes as well. The Set state of ReRAM after soft writes will have higher resistance values than hard writes by the same factor as SWA_{end}, which is 10 in this work. So effectively, soft writes are reducing the sensing window by a factor of 10x. The reduced the sensing margin between set and reset states for soft writes might impact sensing circuits. For the purpose of this work, we assume that the sensing circuit will be able to distinguish HRS and LRS for soft writes too. Given that the normal sensing window for ReRAM is significantly larger than the 10x reduction soft writes are causing, this is a reasonable assumption.

We hope to motivate further research at circuit and device levels by demonstrating the benefit of soft writes.

Chapter 6: Experimental Evaluation

This chapter presents our experimental evaluation of SRTP and other soft write mechanisms. We first present the results when optimizing for the endurance objective function 2.9. The following section covers the results for optimizing total energy using Expression 2.15. Finally, we offer insights into the various SRTP parameters used in the thesis via sensitivity studies.

6.1 Optimizing for Endurance

As discussed in Section 2.2.1, the threshold store-to-store reuse time value for performing soft writes varies depending upon the optimization objective. The objective function for endurance (Equation 2.9) targets an overwrite time value that minimizes the wear out of NVM cells. This section goes over the simulation results for the endurance objective function and dives into the reasons for SRTP's better performance.

6.1.1 Endurance Savings

Here, we quantify SRTP's improvement in endurance gauged by our figure of merit for endurance, effective SWA_{end} (defined in Equation 2.17). In Figure 6.1, we plot the effective SWA_{end} achieved by SRTP and compare it against the Oracle. The graph also plots the endurance improvement of the two versions of RRM we simulate, "RRM-base" and "RRM-aggr."

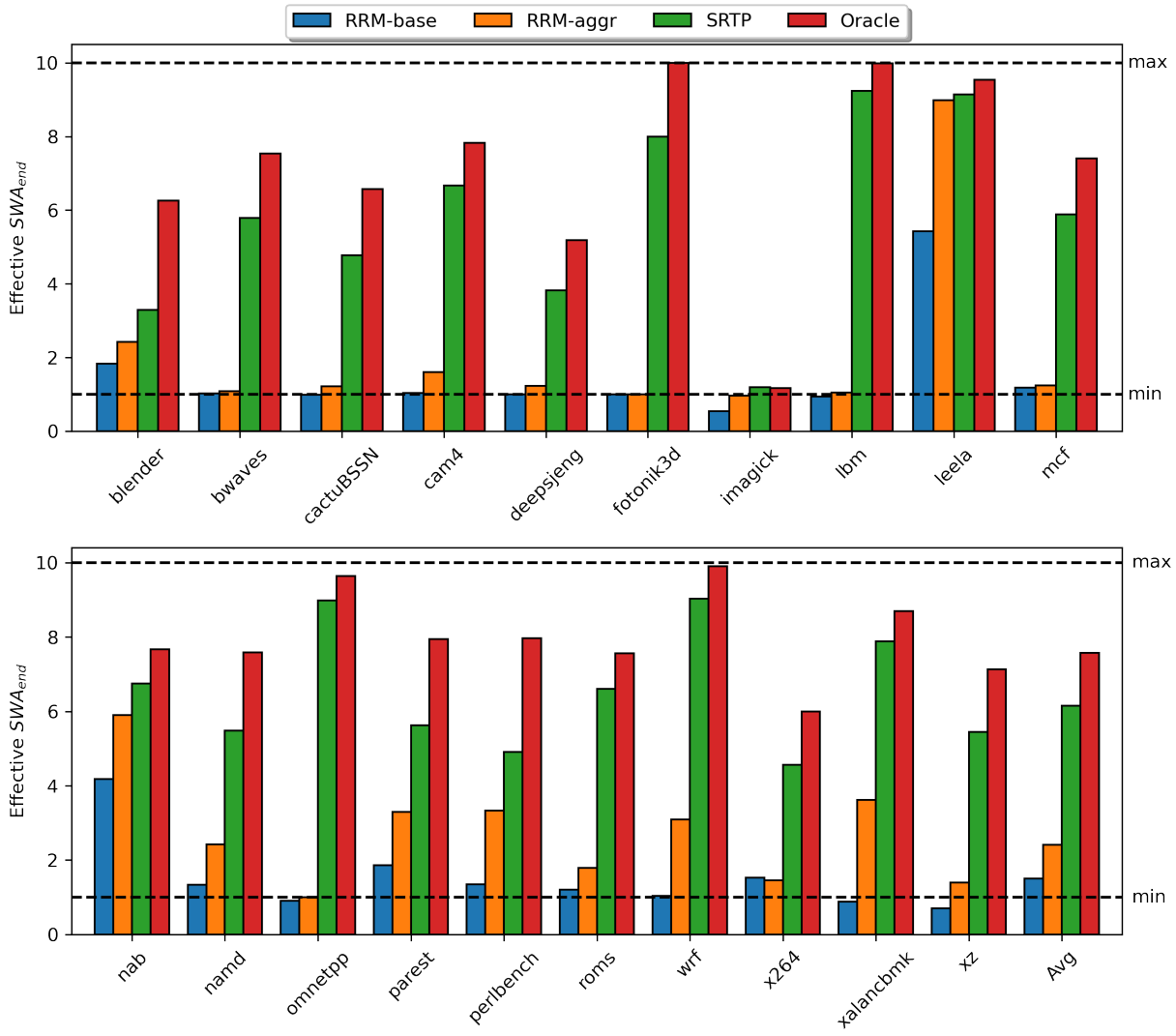

Figure 6.1: Effective SWA_{end} for soft write techniques.

As mentioned in Section 3.2, the Oracle policy achieves an effective SWA_{end} of **7.6x**. Figure 6.1 shows SRTP's effective SWA_{end} is **6.2x**, which is within 18.5% of the Oracle. This is a significant improvement over RRM-base and RRM-aggr, whose effective SWA_{end} are just **1.5x** and **2.4x**, respectively. Overall we achieve **4.1x** better endurance savings compared to the original RRM (RRM-base) and **2.6x** savings over the unrealistically aggressive version of RRM (RRM-aggr).

To provide further insights, Figure 6.2 breaks down the types of writes performed by RRM

(base and aggressive), SRTP, and Oracle. In addition to the soft write, refresh, and hard write categories, Figure 6.2 further breaks down reset hard writes into "eviction" and "decay" categories. The eviction category includes resets due to a shortage of tracking entries. The majority of resets for RRM fall into this category, alluding to a significant shortcoming in keeping track of relevant data. The decay category includes resets that occur when a technique predicts it is no longer profitable to continue issuing soft writes to specific memory locations. All bars are normalized to the total number of writebacks induced by the application. (Because refreshes and resets are in addition to the applications' normal writebacks, the breakdowns add up to more than 100%).

As Figure 6.2 shows, SRTP does two things very well. First, it is able to identify and convert a significant fraction of writes into soft writes. On average, 92% of the application-level writes for SRTP are soft writes, whereas the percentage of soft writes for RRM-base and RRM-aggr is only 30% and 51%, respectively. SRTP is also very close to Oracle, which performs 95% of the application-level writes as soft writes. This is mainly why SRTP's endurance results in Figure 6.1 are superior to RRM and nearly match that of the Oracle. Second, SRTP has a very small number of reset hard writes, just 0.6%, whereas the RRM policies have many more reset hard writes. In particular, RRM-base and RRM-aggr incur 17% and 5.8% resets, respectively, primarily due to evictions. (By definition, the Oracle has zero resets since it omnisciently knows when to perform hard writes).

The improvements over the state-of-the-art RRM can be attributed to several factors. First, the overwrite time metric SRTP tracks (Equation 2.9 or 2.15) is much more accurate compared to a heuristic estimation of hot pages in RRM. Overwrite time between consecutive writes to the same line gives very precise information on whether switching to soft writes is worthwhile. Decisions guided by the hotness of pages determined by counting writes to them are not as

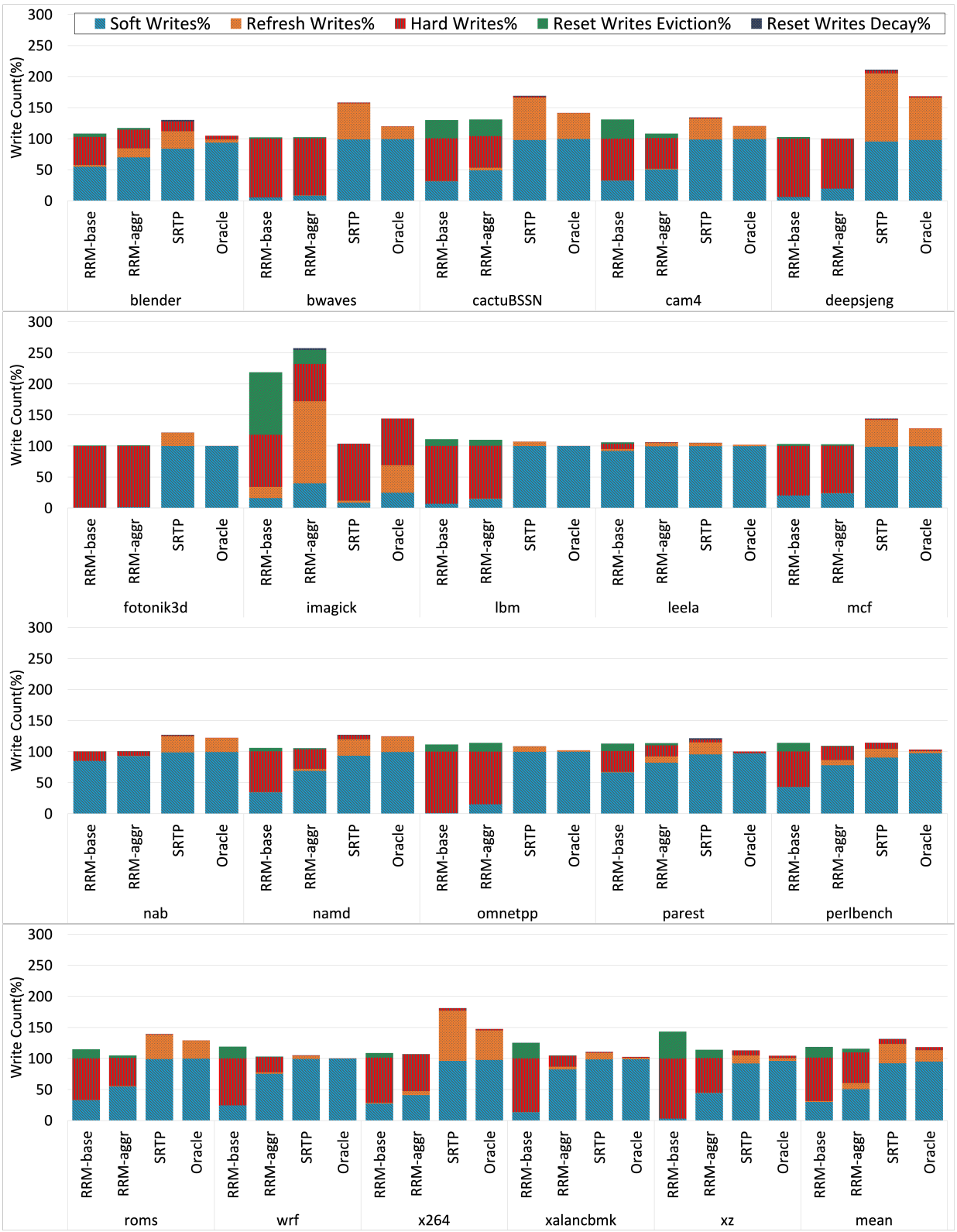

Figure 6.2: Breakdown of different types of writes normalized against the writes without soft write techniques.

accurate. Frequently written pages could be identified, but if a program has a lot of sporadically written pages, they might not be considered hot by RRM, even though the overwrite time is within bounds for soft writes. While on the other hand, SRTP uncovers practically all soft write opportunities because it accurately predicts overwrite time and applies the soft write criterion–Inequality 2.9 or 2.15–to precisely determine the profitability of soft writes for each cache block that is written. This fine-grain manner of soft write decision mimics the behavior of the Oracle, at least for predictions. For refresh and resets, we still track information at a coarser page granularity to reduce overhead. In contrast, RRM's soft write decisions based on coarse-grained page access counts are inaccurate, missing numerous soft write opportunities. For instance, we find that many soft-write candidates may not land on a hot page, particularly when access patterns are sparse. These candidates may still individually exhibit the store locality that warrants employing soft writes.

Not only does SRTP exhibits higher accuracy, but the overhead required to track hot pages in RRM is much larger compared to tracking last-touch store instructions in SRTP, as a single instruction ends up writing to many pages. This is evident from the size of the tables used to track this information in SRTP and RRM. The RTD requires only 11.7 KB, which is 8 and 67 times smaller than the hardware tables used in RRM-base and RRM-aggr, respectively. Despite its small size, the RTD is able to track almost all last-touch store PCs. As a result, we don't observe a lot of evictions from RTD. In contrast, RRM (especially RRM-base) incurs numerous evictions, as evident from the reset eviction components in Figure 6.2. Overall, SRTP's CPU-side prediction mechanism is much lighter weight compared to RRM's memory-side predictor. There are simply many more memory locations written to than there are store PCs writing them.

And lastly, SRTP's lighter weight mechanism also makes it more adaptive. Even if RRM

was able to track every page in memory, it would still need to *train on every page*. Inevitably, some hard writes would sneak through due to long training times. In contrast, SRTP only needs to train on the last-touch store PCs, which can happen rapidly because there are so few of them. Once SRTP has trained for a last-touch store PC, all the pages touched by it will be covered. This allows SRTP to make correct soft write decisions much sooner, as is evident from a much lower fraction of hard writes for SRTP in Figure 6.2.

In conclusion, a PC-based policy is not only able to track soft writes better, but it also has a much more accurate metric for doing soft writes and needs to be trained less frequently. Due to all of these advantages, SRTP comes within 18.5% of the ideal results.

6.1.2 Energy Savings

Although our experiments in this section use Inequality 2.9 to explicitly optimize for endurance (instead of Inequality 2.15 to optimize for energy), they still show significant energy improvements. The threshold overwrite time value beyond which soft writes are not beneficial is 6.4× retention time for optimizing energy instead of 10× retention time for maximizing endurance. Inequality 2.9 performs soft writes for overwrite time between 6.4× retention time and 10× retention time, which when combined with refresh operations, consumes more energy than a single hard write. However, this is just a tiny portion of the writes, and we still get energy savings overall.

Figure 6.3 presents our energy results. In Figure 6.3, we plot the total energy incurred in the memory system normalized to the energy consumed by the baseline memory system that always performs hard writes. The figure also breaks down the total energy into reads and different types

of writes. (Reset hard writes are folded into the hard write category for clarity). We plot the data for RRM-base, RRM-aggr, SRTP, and Oracle schemes along with an additional bar for the baseline system with just hard writes. The energy consumed by the SRTP modules (RTD and SWP) is also part of the soft write energy since they are the enablers of it. However, the energy consumption of these modules is miniscule (see Section 5.2), so their contribution can be safely ignored.

On average, there are 4.5x more reads than writes in SPEC CPU2017 benchmarks. Regardless writes still consume 76% of the memory system energy, on average, as the breakdown of the "Baseline" bars in Figure 6.3 shows. For the most write-intensive benchmark, lbm, writes account for 92% of the energy. The writes need to bring about physical changes in the device. They require higher voltage and current, consuming disproportionately higher energy than reads, as illustrated in Table 5.1. This significantly increases the energy consumption of the main memory system making soft write techniques essential.

Unfortunately, on average, RRM-base provides only a small 3.3% reduction in energy. This is primarily because baseline RRM incurs a great deal of auxiliary hard writes in terms of resets that consume a lot of energy. The situation is significantly worse for some benchmarks with many resets, like imagick, lbm, omnetpp, xz, *etc.*, where RRM-base ends up increasing the energy consumption rather than reducing it. These are mostly decay-based evictions, so even changing the optimization objective to be more favorable to energy reduction will not help. The issue is tied to the tracking capability of RRM-base. RRM-aggr does better, providing a 27% energy reduction on average. The much larger tables in RRM-aggr eliminate a fraction of reset hard writes caused by unnecessary evictions. This results in some energy savings for the majority of benchmarks (energy consumption is still higher than the baseline for imagick).

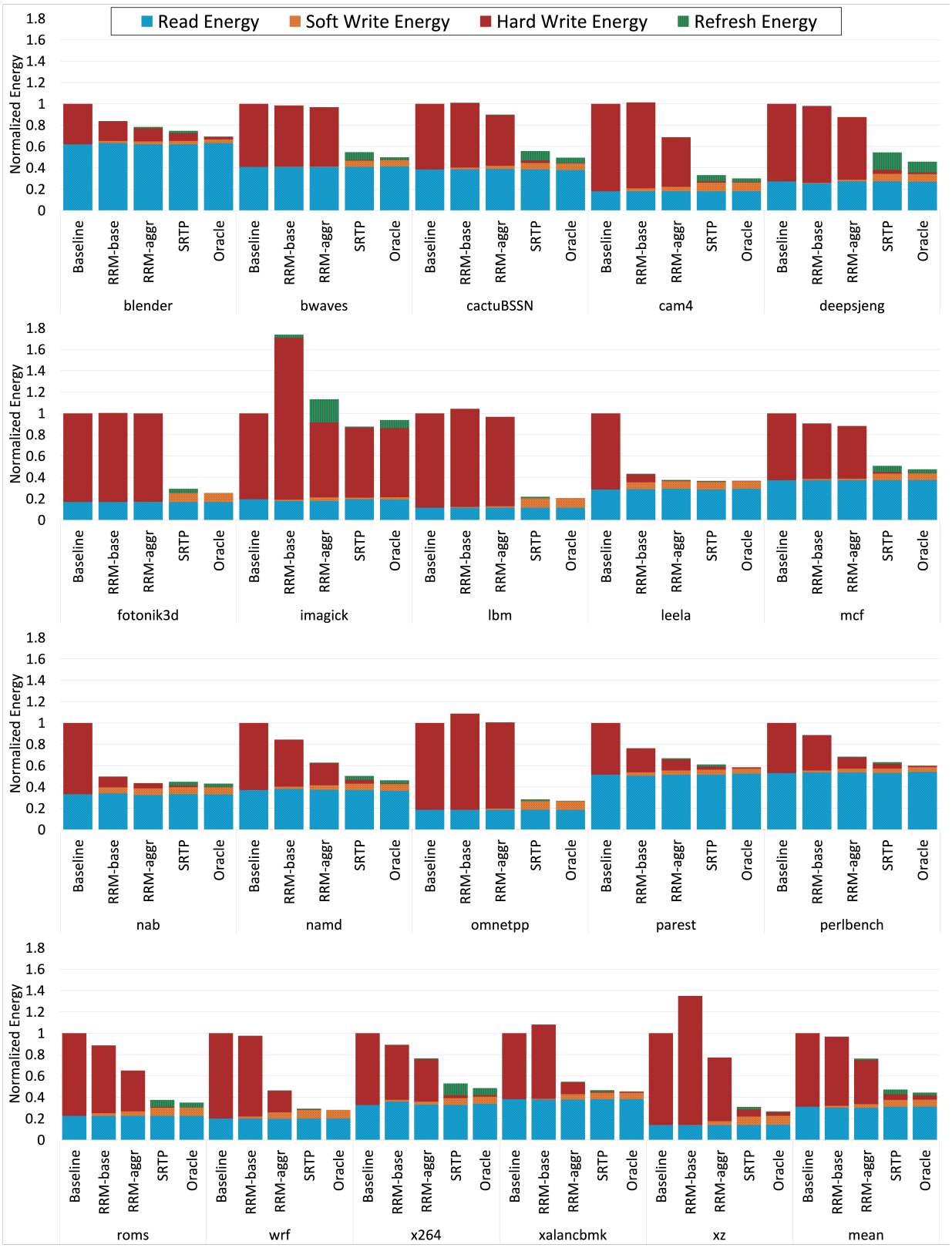

Figure 6.3: Breakdown of memory system energy normalized against main memory energy with only hard writes.

In contrast, SRTP brings forth a significant reduction in memory system energy by reducing the energy consumption of writes by 4.3x. Overall this results in 2.12x reduction of total energy consumed by the memory system for the endurance optimization objective (Expression 2.9). Also, there are no outliers for SRTP where energy consumption increases because of auxiliary writes. The ideal energy savings for the endurance objective function provided by the Oracle is 2.25x, which puts SRTP within 5.8% of the Oracle.

6.1.3 Performance Impact

The performance impact of SRTP is minimal as both of its modules are designed to be off the critical path of the CPU. RTD is accessed for writebacks from the sampled sets of LLC. However, this access is done in parallel to sending the write request to the main memory. Hence, no additional latency is added to LLC writebacks by RTD. The store instructions with SRTP need to update the soft write decision bit along with writing the data in the cache. This puts the SWP modules on the critical path of store instructions being executed on their cores. To avoid any slowdown in the execution of store instructions, it is imperative that the soft write decision is faster than the L1-cache tag lookup. For this reason, we chose a hashed structure for SWP instead of the tagged one in Section 4.3. It is similar to the local predictor from the tournament branch predictor, which comfortably finishes the access in a single cycle [89, 94]. So, the SRTP modules are carefully designed to overlap their latencies with the regular operations of the cores and do not impact their operation directly.

On the other hand, for RRM, all writebacks from LLC must first access its hardware table before being sent to the main memory. So, RRM latency is added to writebacks, but since writes

are buffered and do not block the core, this latency is also easily overlapped. Hence, soft write techniques do not degrade the performance too much compared to the baseline system with only hard writes.

Instructions per cycle (IPC) is a standard metric to quantify a processor's performance. It is the ratio of total instructions executed and the number of cycles consumed while doing so for a particular benchmark. This tells us the average instructions executed per cycle by the processor, so higher is better. Figure 6.4 reports IPC values for various soft write techniques as well as the baseline system with only hard writes.

Soft write techniques improve energy and endurance. However, they also add extra writes in terms of refreshes and resets. Writes are usually buffered, which allows them to be off the critical path of the cores. These additional memory operations are even more separated from the cores since they are handled by the memory controller. Nevertheless, they can still impact the performance by increasing the contention in the memory controller, for example, if a read request queues behind an auxiliary write in a memory bank. In that case, the additional write indirectly blocks a core by delaying its blocking read operation. Figure 6.4 reports the IPC for the different techniques we evaluate. All of the soft write techniques incur a very small performance penalty, indicating that contention effects are minimal. In the case of SRTP, there is a 4.3% performance degradation averaged across all SPEC benchmarks. SRTP's performance hit is the largest since it is also the most aggressive in issuing soft writes (and hence, incurs the most refreshes). But, SRTP's performance impact is minimal relative to the energy and endurance gains it provides.

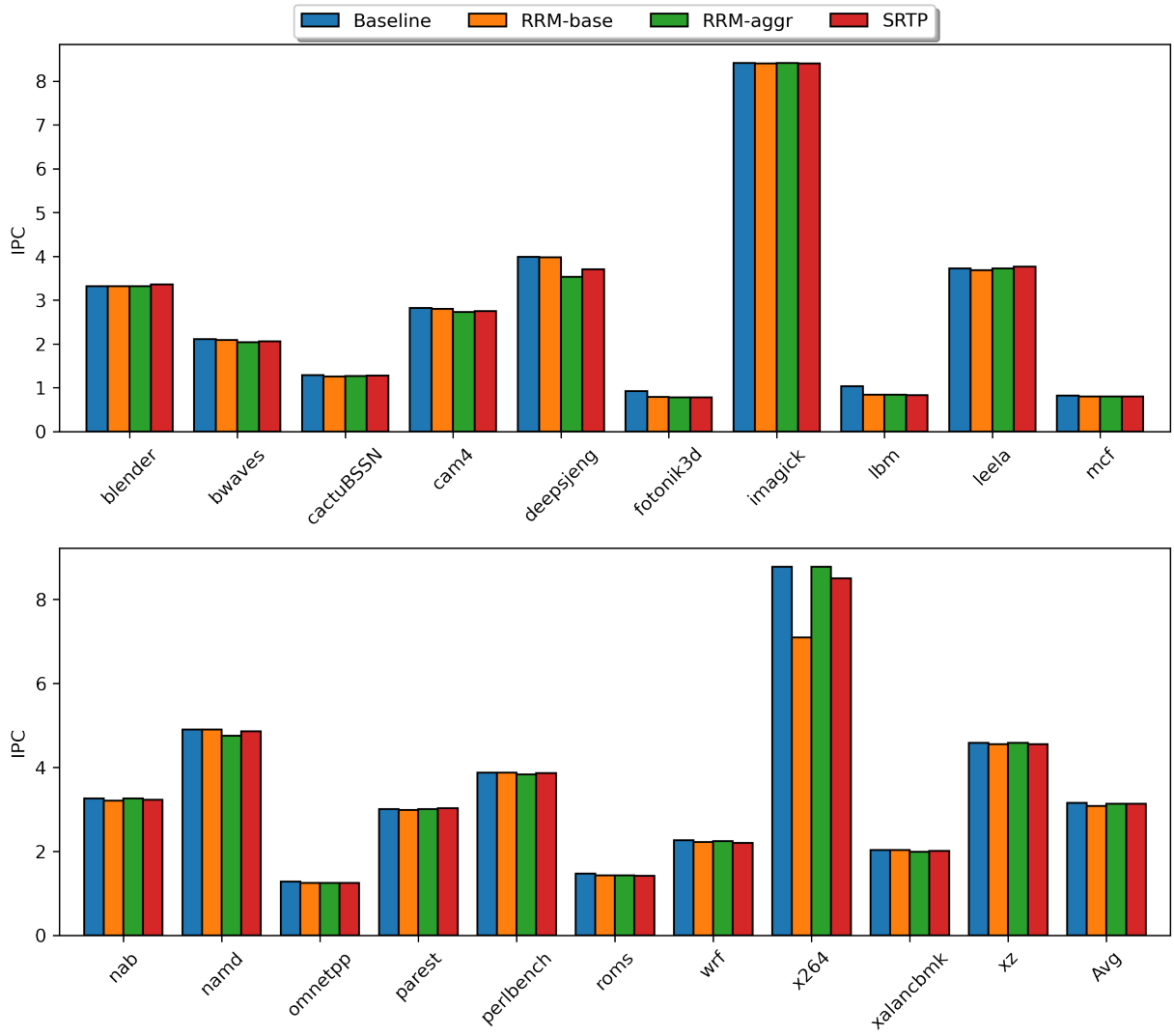

Figure 6.4: Performance impact of soft write techniques.

6.1.4 Mixed Multiprogrammed Workload Impact

Thus far, we have only considered individual benchmarks where the same program runs on all four cores. This provided insights into how various soft write techniques adapted to the access patterns unique to each benchmark. Real-world multicore systems usually have different workloads running together. The performance impact of uneven pressure different workloads put on various shared resources like LLC, network on chip(NoC), main memory, *etc.*, are well docu-

Mixed Workload	Benchmarks	Write Count(Billions)
mix1 (4 High)	lbm, fotonik, omnetpp, roms	63.3
mix2 (4 Medium)	bwaves, cam4, xz, xalancbmk	8.1
mix3 (4 Low)	imagick, blender, x264, leela	0.7
mix4 (2 High + 2 Low)	lbm, fotonik, imagick, blender	19.8
mix5 (2 High + 2 Low)	omnetpp, roms, x264, leela	12
mix6 (2 High + 2 Medium)	lbm, fotonik, bwaves, cam4	53
mix7 (2 High + 2 Medium)	omnetpp, roms, xz, xalancbmk	13.4
mix8 (2 Medium + 2 Low)	bwaves, cam4, imagick, blender	5
mix9 (2 Medium + 2 Low)	xz, xalancbmk, x264, leela	1.3

Table 6.1: Mixed multiprogrammed workloads.

mented [95–101]. Soft write techniques also have shared structures like the hardware table in RRM and RTD in SRTP. So, studying the impact of heterogeneous workloads is crucial to understanding the readiness of these schemes for deployment in real systems.

For this purpose, we created mixed workloads by combining SPEC CPU2017 benchmarks. The benchmarks in Table 5.2 are grouped into high, medium, and low write intensity (WPKI) categories. We picked 4 representative benchmarks from each category and used them to create new mixed workloads. Table 6.1 shows the nine multiprogrammed workloads we created and the mix of write intensities they represent. The first three mixes have different benchmarks but with similar write intensities(all high, all medium, or all low). This will allow us to study the impact of multiple programs for each write intensity category. For the remaining six mixes, we combine benchmarks from different categories. These will have different programs that also vary in their write intensities running together. Figure 6.5 plots the endurance results for these workloads in the exact same format as Figures 3.2 and 6.1.

Figure 6.5 shows that SRTP does even better on mixed workloads. On average, SRTP achieves an effective SWA_{end} of 6.8x, up from 6.2x in Figure 6.1. This might seem counter-intuitive since we were expecting the performance to be worse for heterogeneous benchmarks.

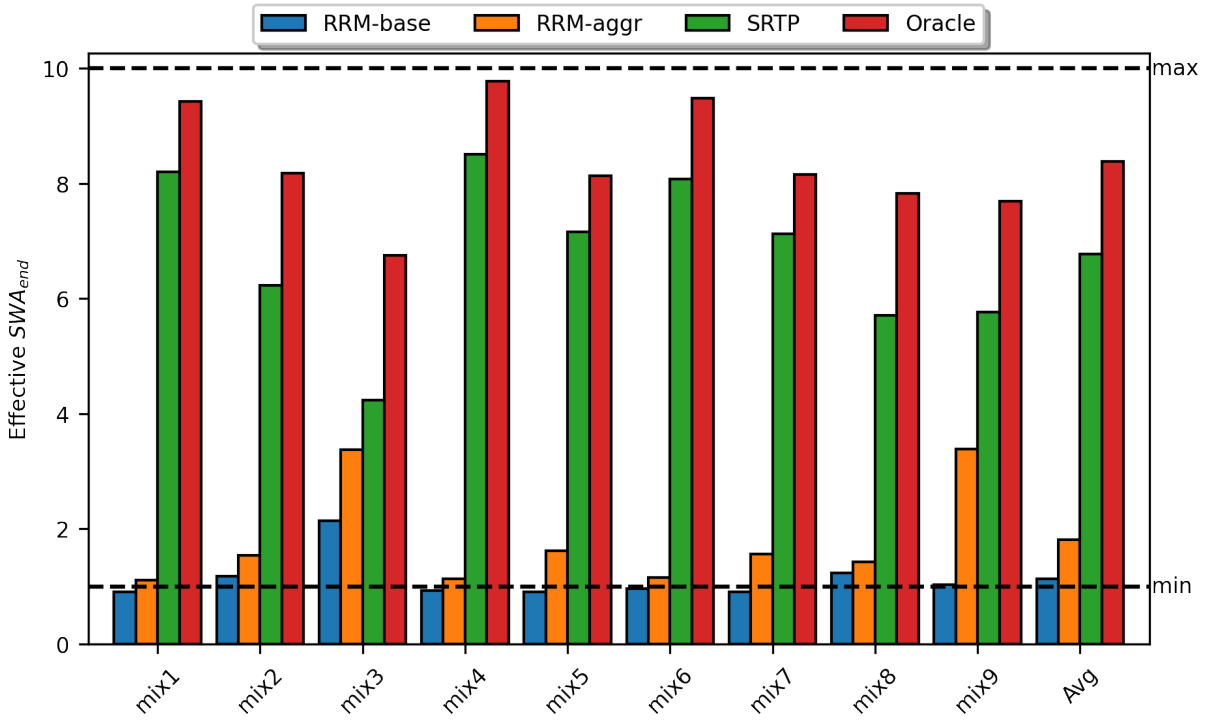

Figure 6.5: Effective SWA_{end} for mixed workloads.

Just looking at effective SWA_{end} is misleading as the Oracle's performance improves as well from 7.6x for homogeneous benchmarks to 8.4x for mixed workloads. So, there are more overall soft write opportunities in the case of these benchmarks and hence the higher endurance numbers for SRTP.

A better way to gauge the effectiveness of SRTP would be to consider its difference from the maximum achievable endurance savings of the Oracle. For mixed workloads, SRTP comes within 19.1% of the Oracle, up from 18.5% for individual benchmarks in Figure 6.5. Since SRTP lags further behind the Oracle for mixed workloads, its performance does suffer but only by a small amount (0.6%). The primary reason behind this is the additional pressure on RTD as, in this case, it needs to track the last-touch store PCs for multiple programs. This increases the amount of state it has to cover, leading to more missed opportunities. Nevertheless, as discussed

in Section 3.3, typically, there are not a lot of unique last-touch store instructions in a program (On average, only 76 last-touch PCs account for over 90% of writebacks). So, the current capacity of RTD at 512 entries is enough to handle the additional pressure without impacting the performance of SRTP too much. Another possible reason could be aliasing in RTD, where PC values from different programs might collide. This can be easily resolved by adding "core-id" bits to the instruction tags that will differentiate the colliding PC values. However, given the small number of important last-touch static stores and a large number of overall instruction addresses available, the chances of these last-touch store instructions having the same PC values for different workloads are negligible. This allows us to omit the core-id bits without worrying about aliasing in RTD. The experiments simulate aliasing, and SRTP is already fairly close to the ideal oracle's performance, supporting this argument.

While the value of SRTP's endurance gains improves for mixed workloads managing to keep up with the oracle, RRM's performance worsens. The effective SWA_{end} for RRM-base is down from 1.5x to just 1.13x for mixed workloads. While RRM-aggr does better than that, its effective SWA_{end} is still reduced from 2.4x to 1.8x. Overall, SRTP is 6x and 3.8x better than RRM-base and RRM-aggr, respectively, up from 4.1x and 2.6x in Figure 6.1. So, RRM's sensitivity to multiple workloads running together is much higher than SRTP.

Another observation we can draw from Figure 6.5 is the behavior of soft write techniques as we move from high to low write intensity (mix1 to mix3). The gains from the Oracle reduce since benchmarks that do not write very frequently often have large store-to-store overwrite time values limiting the soft write opportunities. The Behavior of SRTP and RRM relative to the Oracle is the exact opposite here. SRTP is very close to the Oracle's results for write-intensive benchmarks (SRTP is within 13% of Oracle for mix 1), but as write intensity reduces, the distance

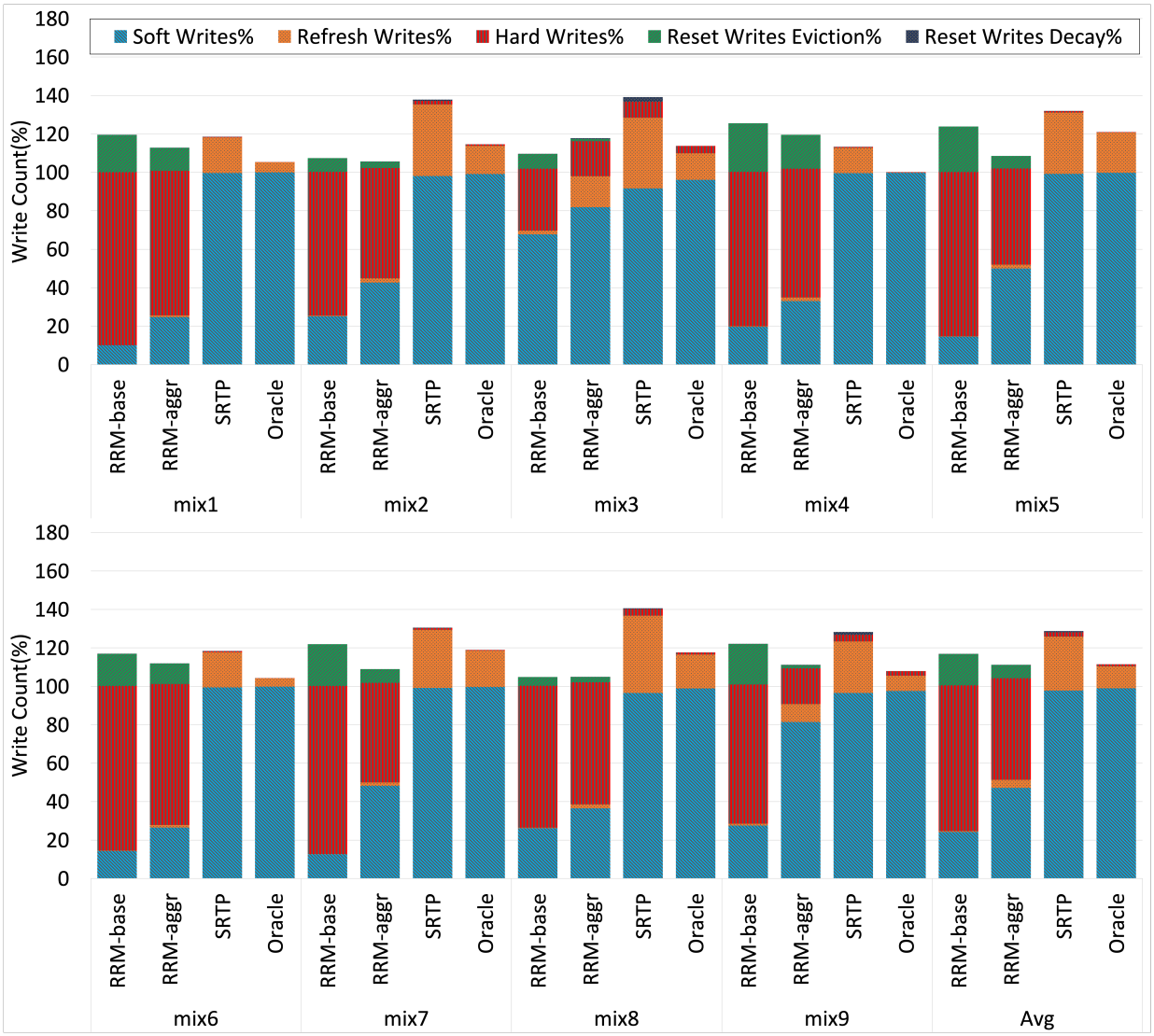

Figure 6.6: Breakdown of different types of writes for mixed workloads normalized against the writes without soft write techniques.

between them grows (SRTP is within 37% of Oracle for mix3).

This is again due to the fact that there are fewer soft write opportunities overall as well as each last-touch PC does far fewer writes for mix3. As a result, SRTP can take less advantage of the overwrite time values it learns for these PCs. RRM, on the other hand, gets barely any benefit for write-intensive benchmarks (RRM-aggr is 8.4x times lower than the Oracle for mix1). However, as the write intensity reduces, it starts to catch up (RRM-aggr is 2x lower than the

Oracle for mix3). SRTP still outperforms RRM in all cases. Just the margin is not as high when less write-intensive benchmarks are running. Usually, if a benchmark is not writing a lot to the main memory, it stands to reason that it will not be modifying a lot of pages. This reduces the amount of state page-based policies need to track, allowing them to catch up.

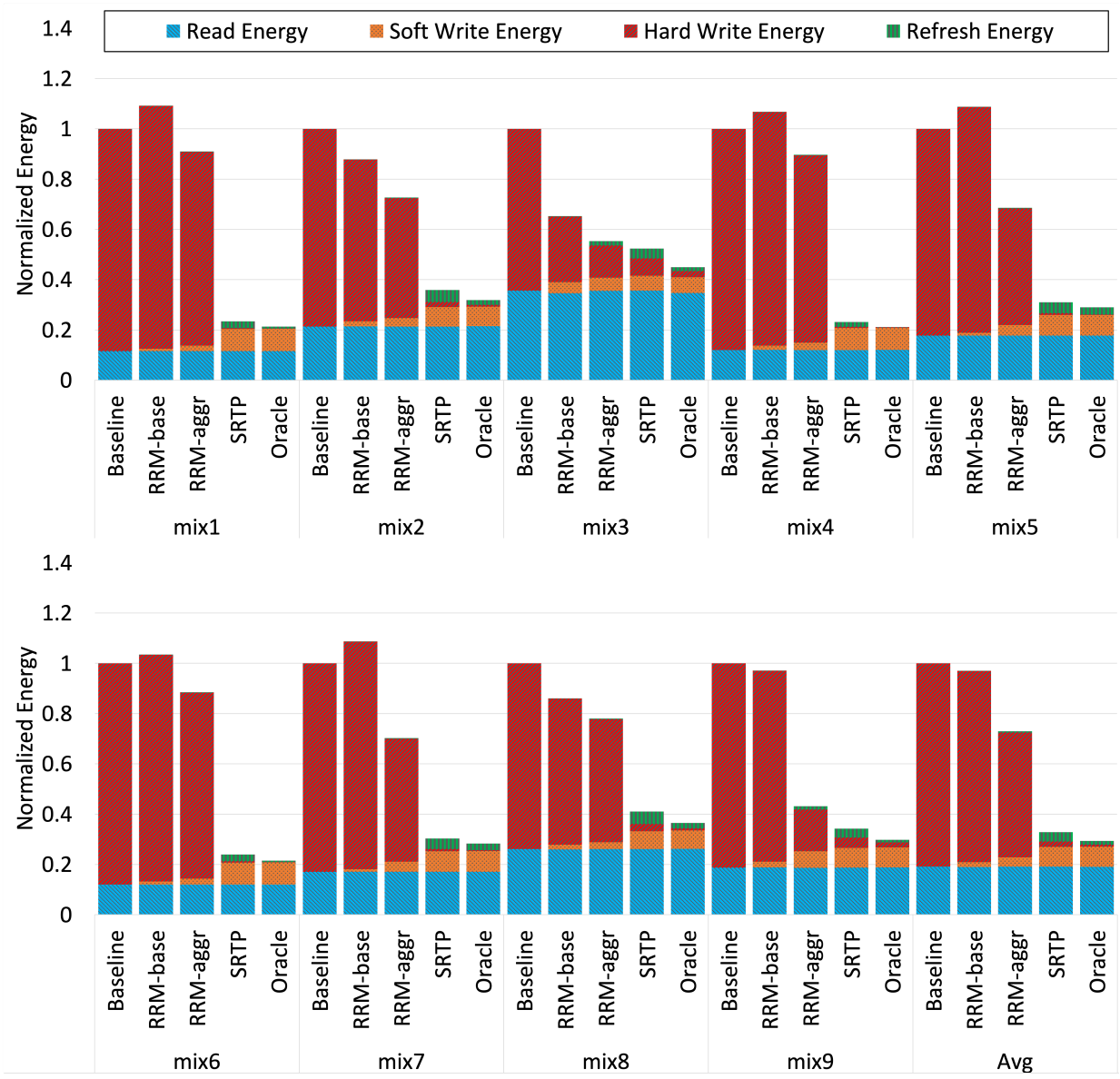

Figure 6.7: Memory system energy breakdown for mixed workloads.

The write-intensive workloads (mix1) issue 90 times more writes than the multiprogrammed workload with low write intensity (mix3), degrading the NVM endurance faster by the same fac-

tor. So soft writes are sorely needed for programs that write a lot to main memory. SRTP is a better technique in this regard since its efficacy improves with the write intensity of the workloads. In contrast, RRM does not deliver endurance savings where it matters the most. As we will see in Sections 7.3 and 7.4.1, the baseline scheme with only hard writes provides a reasonable lifetime for programs with really low write intensity. Soft write techniques are needed mainly for write-intensive programs. This is not limited to the case where all programs are write-intensive. Even if a write-intensive benchmark is running with other programs that do not write as often, it will end up dominating, skewing the write intensity of the mix to the higher side. This is evident for mixed workloads from 4 to 7. SRTP follows the Oracle closely while RRM provides very small endurance gains (RRM-base even makes the endurance worse in some cases). RRM gets some gains when only low or medium write-intensity programs are running. Which we have already established is not the main target for soft write techniques. Also, in a realistic system, there is a good chance some of the programs that are running together at any instant will have high write intensity, rendering RRM ineffective.

Figure 6.7 plots the main memory energy for the mixed workloads normalized to the baseline energy of hard writes only case similar to Figure 6.3. The same issue persists that writes consume a significant portion of energy in the baseline system at 81% and reads only account for 19% of energy consumption. Bringing this asymmetrically high write energy down is imperative to keep the energy budget in check as the storage and workloads scale. RRM-base and RRM-aggr reduce energy consumption by 1.04x and 1.5x, respectively. RRM-base especially struggles when write-intensive benchmarks are present in the mix, and its energy consumption is more than the baseline for all of those cases. This is for the same reason that a write-intensive benchmark tends to modify a lot of pages. The capacity of RRM-base is not enough to keep track of all of

these pages, and this also interferes with other programs running alongside the write-intensive ones. This results in a lot of eviction based resets as well as fewer opportunities for soft writes, as shown in Figure 6.6. RRM-aggr manages to do better because of the unrealistically high capacity afforded to it. SRTP reduces the overall energy consumption by 3x, which is within 11.7% of the Oracle's reduction in total main memory energy of 3.4x. Similar to our discussion earlier about the endurance savings, behavior of RRM and SRTP is very similar in the case of energy savings. RRM works better for the less write-intensive benchmarks, which are not the primary target of this study. On the other hand, SRTP tends to reduce energy consumption by a larger amount for the write-intensive benchmarks where it is desperately needed while maintaining a significant lead over the state-of-the-art in all cases.

6.2 Optimizing for Energy

The last section presented results for soft write policies with the optimization objective geared towards maximizing endurance. As we saw in Section 2.2.1, the optimization criterion for minimizing total energy consumption is different from endurance because it needs to consider total energy while endurance optimization only considers write energy. This results in refresh operations becoming costlier from total energy perspective, as they perform a read and a write. So, in this case, we can afford to do less number of refreshes before they break even with hard writes.

6.2.1 Energy Savings

Equation 2.15 gives the overwrite time threshold for energy optimization. Substituting the energy values from Table 5.1 gives us the SWA_{egy} of 6.4x, which is lower than the SWA_{end} of 10x on the account of added read energy for refresh operations. This means one soft write and 6.4 refreshes break even with a single hard write in terms of total energy consumption. Since we cannot have fractional refresh operation, we set the SWA_{egy} to 6x. This allows soft writes with overwrite time values incurring up to 6 refresh operations before the next write arrives.

Figure 6.8 plots the total main memory energy normalized to the energy of the baseline system with only hard writes. Similar to Figure 6.3, it also breaks down the total energy into reads and different types of writes. The baseline bar is for the system without soft writes, which highlights energy spent in reads and writes. We also plot data for RRM-base, RRM-aggr, SRTP, and Oracle schemes. We can safely ignore the negligible energy consumed by SRTP modules (see Section 5.2) in the same manner as in Section 6.1.2.

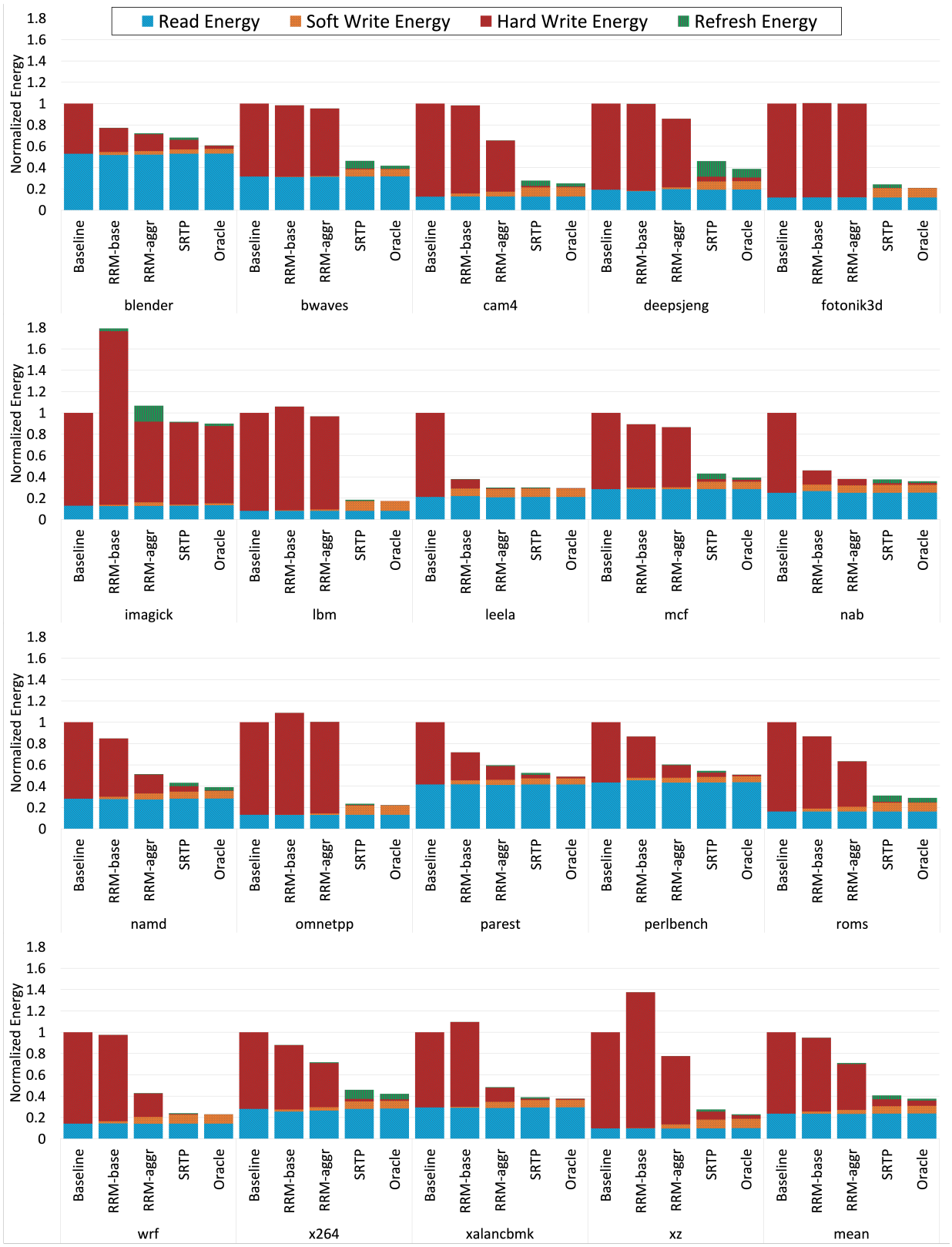

Figure 6.8: Breakdown of memory system energy normalized against main memory energy with only hard writes for total energy optimization objective.

For SPEC CPU2017 benchmarks, the energy optimization results are not that different from the results for endurance objective. This is due to two factors. For starters, the number of writes with overwrite time in the range between energy and endurance objectives (60 to 100 seconds) is small. On an average, only 1% of soft writes get converted to hard writes when we reduce the overwrite time threshold from 100 seconds to 60 seconds. Second, writes with long overwrite time provide diminishing results. So, switching them between soft and hard writes does not make a lot of difference. For example, consider two write streams with overwrite time values of 1 and 80 seconds, respectively. Converting the faster stream from hard writes to soft writes will reduce their energy consumption by 8.5 times. Switching the slower stream from soft writes to hard writes, on the other hand, will only reduce their energy consumption by 1.4 times. As a result, for particularly large values of overwrite time, the number of writes is fewer, and the returns are diminishing. Hence, changing the optimization objective from endurance to total energy only reduces the energy consumption for the ideal oracle by 7.5%. SRTP's energy savings are also very similar to endurance objective at 2.45x.

6.2.2 Endurance Savings

Figure 6.9, plots the effective SWA_{end} for SPEC CPU2017 benchmarks, when total energy optimization objective is used. The endurance results are also quite comparable to the endurance optimization objective for the same reasons discussed in the last section. The writes in the range where energy objective differs from endurance objective are much smaller with diminishing impact on endurance. The effective SWA_{end} for RRM-base, RRM-aggr, SRTP, and Oracle are 1.46, 2.5, 6.23, and 7.62, respectively.

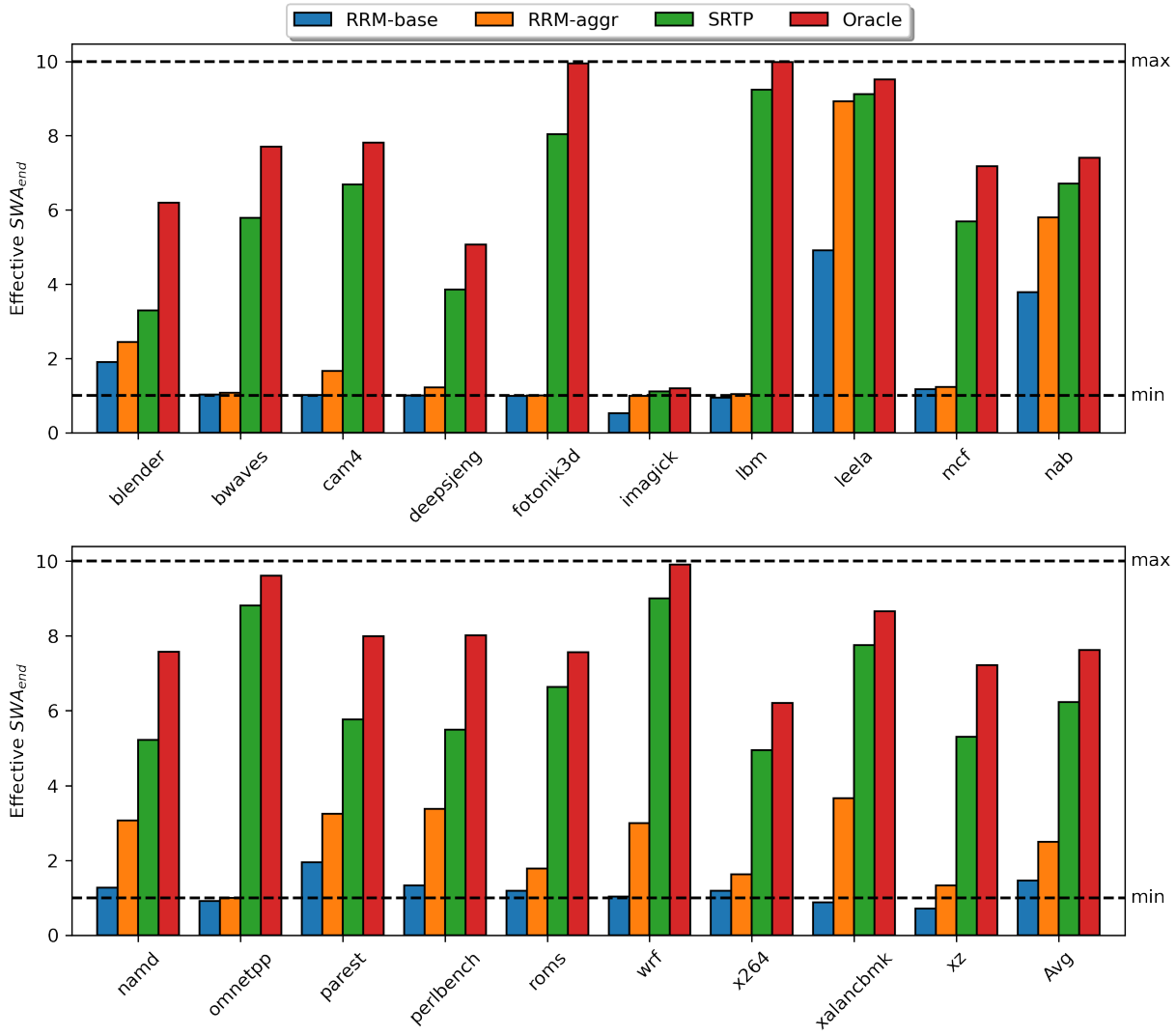

Figure 6.9: Effective SWA_{end} for soft write techniques with energy optimization objective.

6.3 Sensitivity Studies

The previous two sections presented the energy and endurance advantage of SRTP at a minimal performance cost. We demonstrated substantial improvements over the state of the art and proximity to the Oracle policy's results. In this section, we conduct sensitivity studies for various parameters associated with SRTP, study their impact on its performance, and discuss the underlying reasons for any changes.

6.3.1 Impact of Sampling in Cache

SRTP needs additional hardware on the CPU die to detect the overwrite time for last-touch store instructions and associate correct soft write decisions to the data written by them. The majority of processors use writeback caches at the lower levels, where writes are not sent to the main-memory immediately when a store instruction is executed. The modified data lingers in the cache and is sent to the main memory only when evicted from LLC by another piece of incoming data. When the written data is being sent to main-memory, RTD needs its address and last-touch store instruction to determine the overwrite time which in turn guides the soft write decisions (Section 4.2). While the address is available at the time of eviction, the last-touch store PC values are only accessible when executing the store instructions, not during the writebacks from LLC at a later point. Hence, this necessitates storing the last-touch store PC values along with the modified data. As discussed in Section 4.1, the overhead of storing the last-store PC could account for up to 10% of the LLC cache capacity. SRTP employs dynamic set sampling (DSS) to make this overhead more reasonable. Here, a small fraction of cache sets are chosen, and last-touch PC is tracked only for these sampled sets. This reduces the tracking overhead by the sampling factor.

Sampling fraction	PC tracking overhead (KB)	SRTP on die overhead (KB)
1	701.4	749.3
1/8	87.7	135.6
1/16	43.8	91.7
1/32	21.9	69.8
1/64	11	58.9
1/128	5.5	53.4

Table 6.2: Variation of SRTP overhead on CPU die with sampling fraction.

For instance, the overhead is 32 times lower if we sample one out of every 32 sets. If a particular behavior is stable across cache sets, sampling a few sets is adequate to get the complete picture.

DSS has previously been used with cache replacement policies, dead block prediction techniques etc. To the best of our knowledge, this is the first time it is being used while ascertaining the overwrite time values. The impact of DSS on the accuracy of our technique would depend upon the stability of the behavior it is trying to capture. Section 3.3 demonstrated the overwrite time value has a single dominant peak for an individual static store instruction of a program. Hence, sampling only a few instances of each last-touch store instruction is sufficient to capture this behavior, making DSS a good fit to reduce overhead for SRTP without seriously compromising the accuracy of the scheme.

This section studies the impact of sampling on SRTP's performance. *Sampling fraction* is the ratio of sample size to the population size. We modify the sampling fraction by changing the number of sampled cache sets and observe the variation in the endurance savings for SRTP. Figure 6.10 plots effective SWA_{end} for SRTP as sampling fraction is varied from 1 to $\frac{1}{128}$, which represent cases where all sets have PC tags to one in every 128 sets has a PC tag, respectively. The tracking overhead for last-touch store PCs goes by the sampling fraction and is shown in Table 6.2. Effectively, we are trading off overhead by choosing to sample frequently in the cache

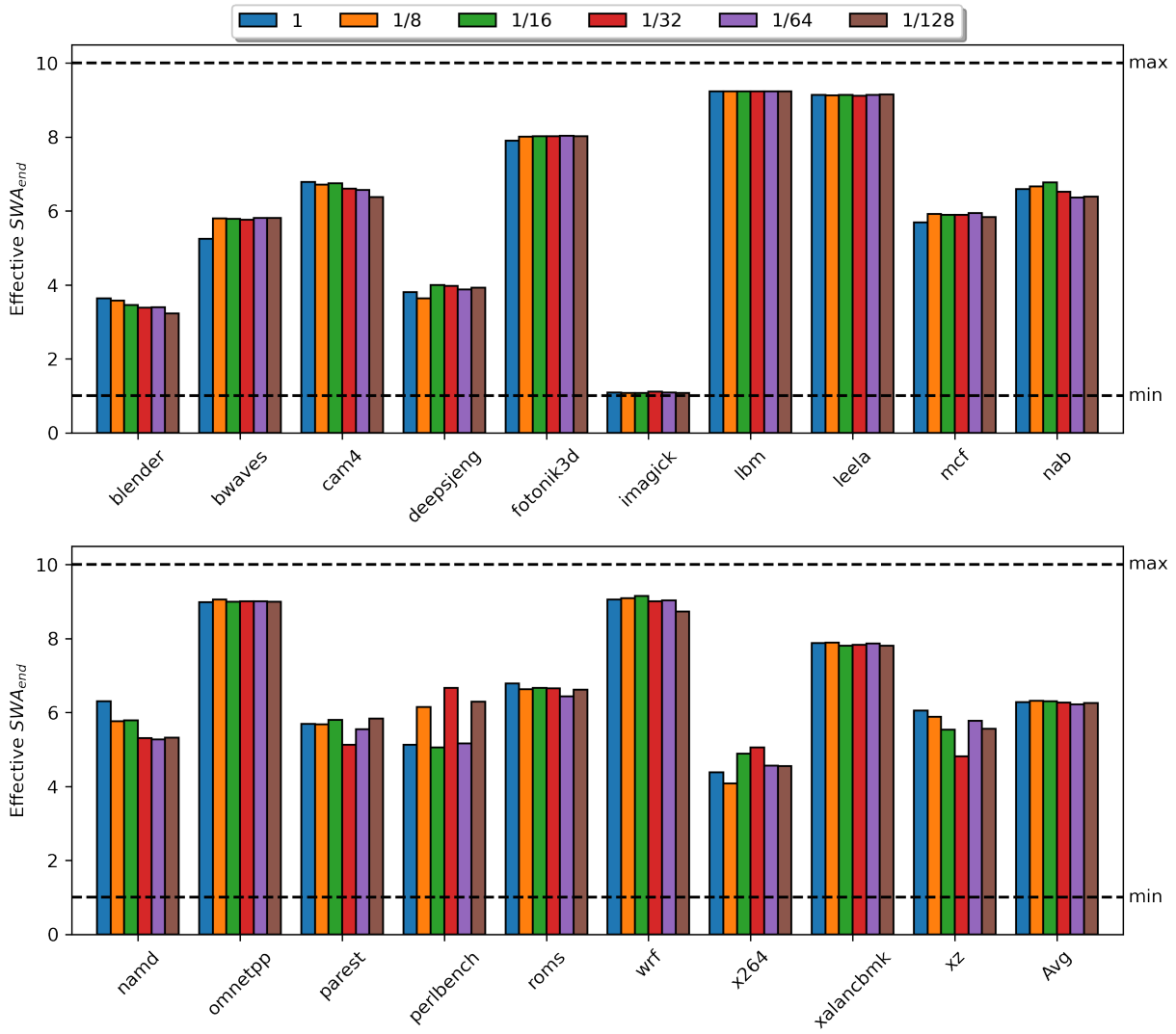

Figure 6.10: Sensitivity of effective SWA_{end} for homogenous benchmarks to the sampling fraction for last-touch PC tags in cache.

hierarchy. It also shows the overall on die SRAM overhead of SRTP. The NVM overhead on the memory side is omitted as it remains constant at 512KB.

The no sampling case with sampling fraction of 1 has effective SWA_{end} of 6.28 which is 0.3% more than the result for baseline SRTP with sampling fraction of $\frac{1}{32}$. This is reasonable price to pay for 10.8 times reduction in the table overhead on the CPU die. The results also demonstrate that the sampling fraction can be reduced all the way down to $\frac{1}{128}$ without further

degrading the performance. This makes the current SRTP overhead conservative with a further scope for 23.5% reduction. Overall, very small changes in performance with sampling further underscores the stability of the store overwrite time values as well as SRTP's ability to ascertain it with only a few instances.

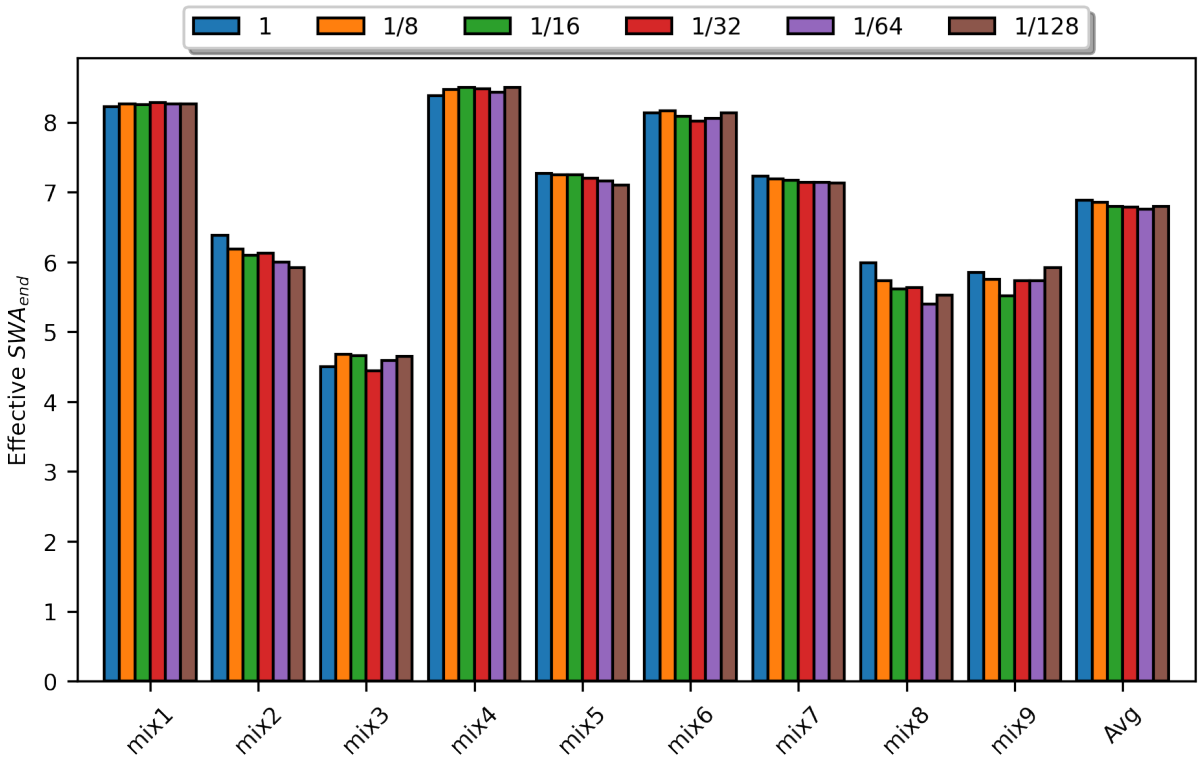

Figure 6.11: Sensitivity of effective SWA_{end} for mixed workloads to the sampling fraction for last-touch PC tags in cache.

Very similar behavior is observed for mixed workloads with sampling shown in Figure 6.11. Sampling has almost negligible impact on the performance of SRTP with effective SWA_{end} increasing by only 1.4% when sampling fraction was changed from $\frac{1}{32}$ to 1. When the sampling fraction is reduced beyond $\frac{1}{32}$, the results don't change. Therefore, earlier observation about sampling and its effect on endurance savings holds true whether the workloads are homogenous or mixed.

6.3.2 Impact of RTD Entries

The reuse time detector is responsible for identifying the overwrite time peaks associated with each last-touch store PC, as described in Section 4.2. The SWP modules are then trained using this data to generate soft write predictions for those PC values. In contrast to the typical microarchitectural components like branch predictors and prefetchers, which cover events at the scale of nanoseconds, we are interested in detecting overwrite time values that are in the range of seconds. This means the entries in RTD need to persist for seconds to make a prediction. Consequently, it is crucial to have enough entries in RTD to accommodate the majority of static last-touch store instructions associated with the programs executing on the cores. We have noted that there are typically not a lot of these instructions in a program; hence, RTD does not require a lot of storage. For the SPEC CPU2017 benchmarks, the top 76 last-touch store instructions account for over 90% of all program writes, though not all of them are active simultaneously.

This section examines the effects of RTD's capacity to track different PC values by varying the number of entries associated with it. The more entries the RTD is assigned, the more last-touch PC values it can track without displacing other entries. But, this also means that on-chip storage overhead will increase. Therefore, striking a balance between the two is very important where we get optimal performance out of SRTP without taking up a lot of space on the CPU die. Figure 6.12 plots the effective SWA_{end} for SRTP while varying the RTD entry count. Here, we increase the number of sets in RTD from 2 to 128 while keeping the associativity constant at 16. This increases the RTD entries from 32 to 2048, and the overhead is increased from 1.5 KB to 46.8 KB.

The rate of eviction for last-touch PC entries in RTD goes down as more space is allocated

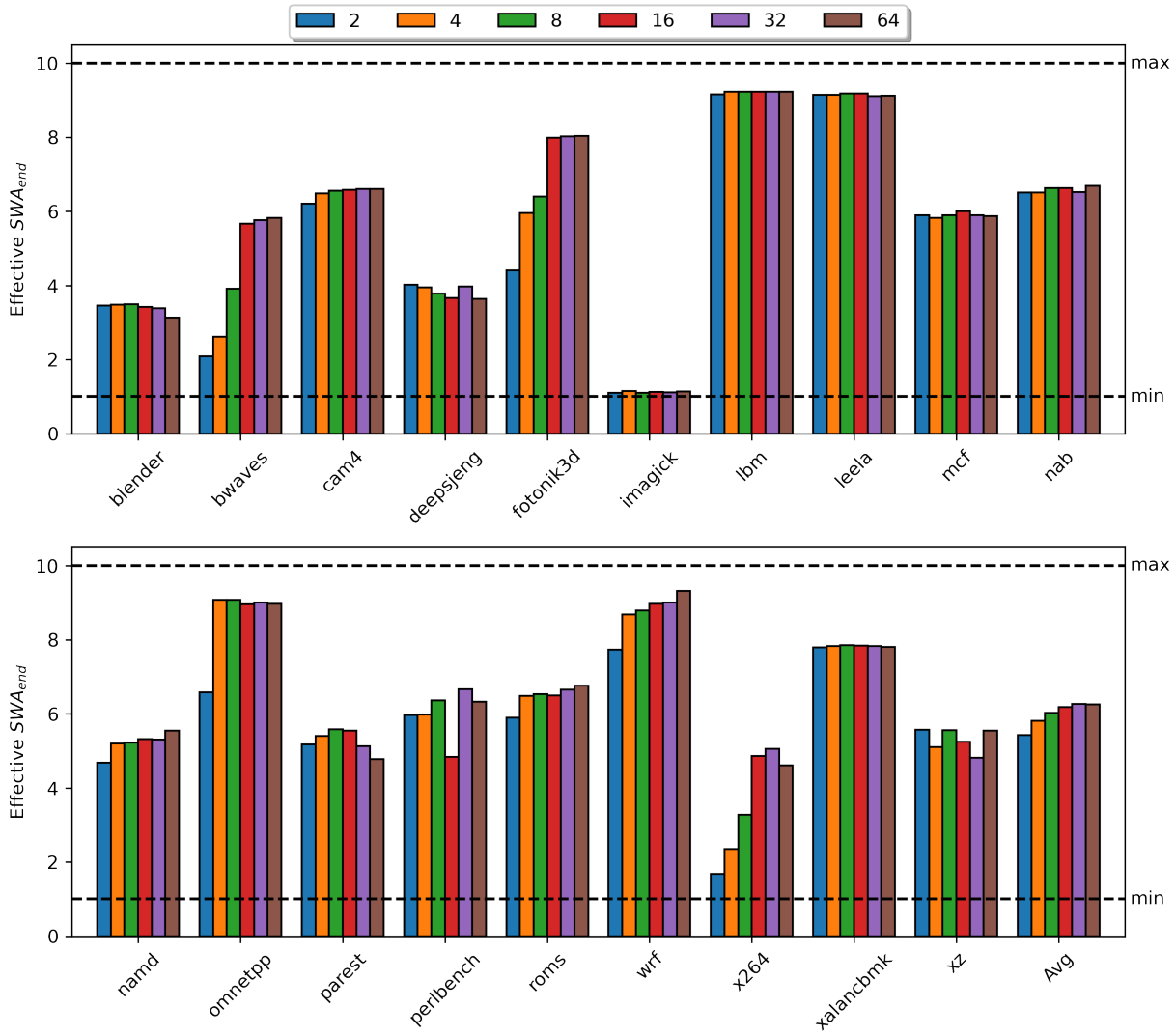

Figure 6.12: Sensitivity of effective SWA_{end} for homogenous benchmarks to the number of entries in SRTP.

for them. Figure 6.13 shows the average rate of evictions (number of evictions divided by total accesses) in RTD for SPEC CPU2017 benchmarks as the number of RTD sets is altered. Beyond 32 sets, the eviction rate is almost zero. The hit rate increases with more entries as fewer evictions suggest RTD can better accommodate the working set of last-touch PCs. Figure 6.14 plots the average last-touch PC hit rate in RTD for SPEC CPU2017 benchmarks. The RTD can keep the entries for longer as the hit rate increases, allowing it to learn the large-scale overwrite times.

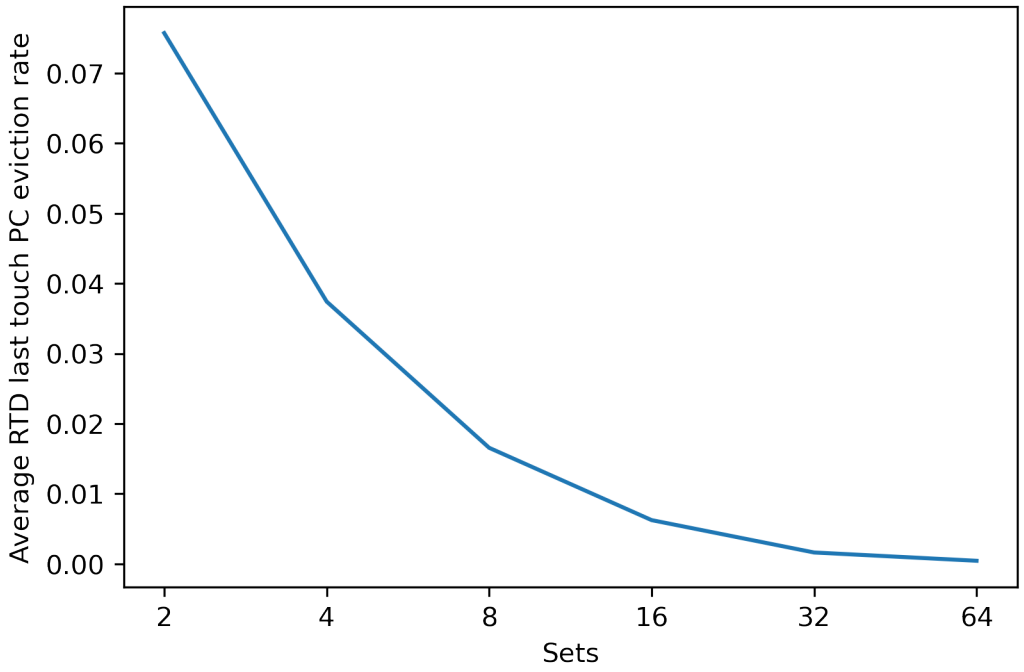

Figure 6.13: The average last-touch PC entry eviction rate in RTD for SPEC CPU2017 benchmarks for varying RTD entries.

Consequently, we observe improvement in effective SWA_{end}, which went from 5.42x for 2 sets to 6.27x for 32 sets. With gains plateauing at 32 sets, the tracking overhead is not too high for homogenous workloads. In fact, RTD is almost four times over provisioned because, with the exception of three benchmarks, the endurance gains for the 8 sets closely follow those for the 32 sets. On average, the effective SWA_{end} for 8 sets is 6.02x which is within 96% of the maximum achieved at 32 sets. It is possible to cut the RTD overhead by 4 times at a small cost of 4% reduction in endurance savings.

Mixed workloads are more sensitive to the number of RTD entries since they have a more diverse set of static store instructions to track. Figure 6.15 plots the effective SWA_{end} for mixed workloads from Table 6.1 at varying number of RTD sets while maintaining associativity constant at 16. Similar to the homogenous case, as more entries become available, the store PC hit rate

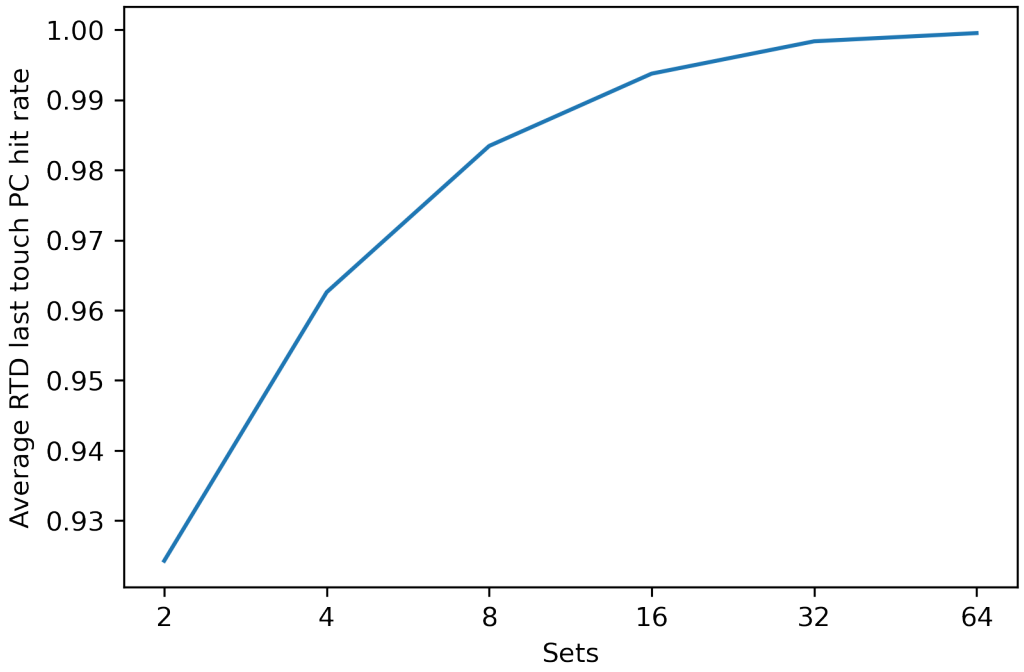

Figure 6.14: The average last-touch PC hit rate in RTD for SPEC CPU2017 benchmarks for varying RTD entries.

increases and the eviction rate goes down. Once more, the performance peaks at 32 sets, but this time, RTD is at most two times over provisioned. The endurance savings start to decline more quickly, as the number of sets is decreased below 16. Effective SWA_{end} drops by almost 23% from baseline to 5.23x when number of RTD sets is reduced to 2.

6.3.3 Impact of RTD Associativity

Hardware caches use a small set of bits from the address to index into the sets. This often results in situations where many accesses pathologically map to a small number of sets. If direct mapped cache was used with just one entry per set, then these conflicts would cause frequent evictions from cache even though space was available in other sets. RTD is very similar to cache with last-touch PC used as the tags instead of data addresses. This section examines RTD's

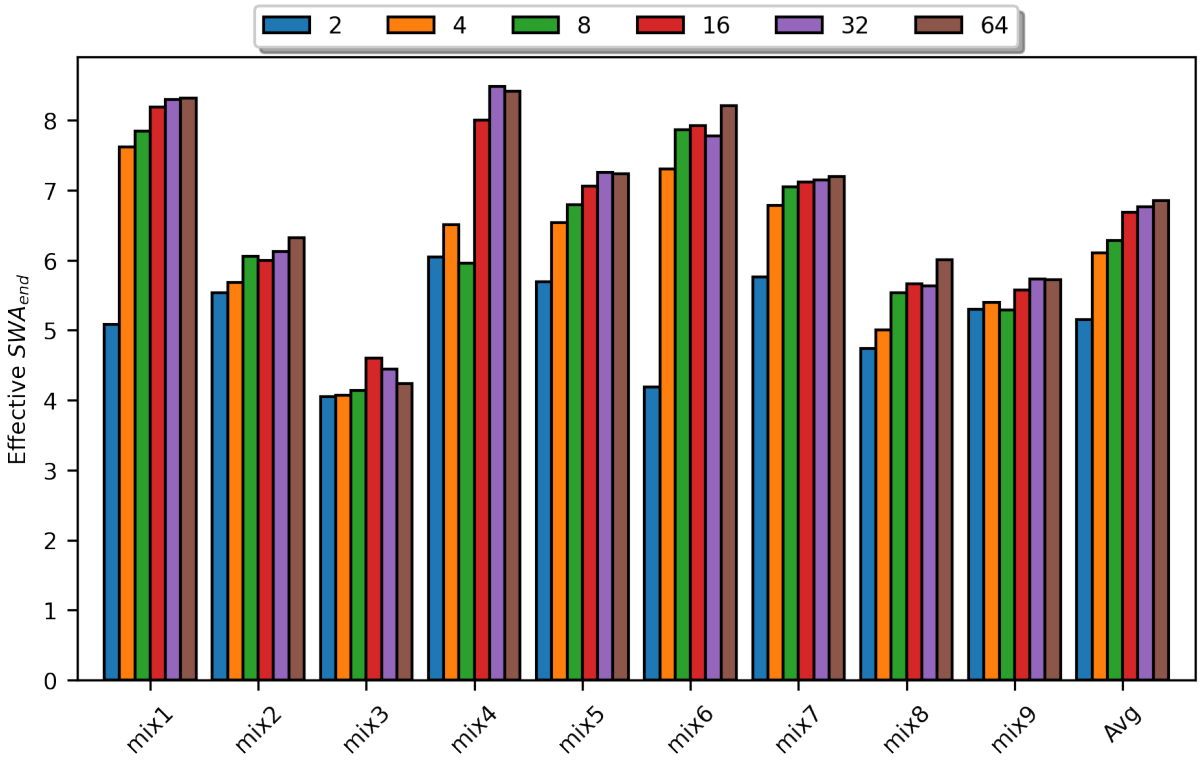

Figure 6.15: Sensitivity of effective SWA_{end} for mixed workloads to the number of entries in SRTP.

susceptibility to conflicts within its sets and impact of associativity. Making the RTD N-way set associative assigns N entries to each set where the data mapping to that set might be found. With N places to go for each last-touch PC mapping to a set, conflict pressure is reduced significantly. But this slows down the cache/RTD access rate due to added complexity of checking multiple entries. RTD accesses, however, are parallel to main memory accesses, keeping it out of the critical path, as we saw in Section 4.2. As a result, its access latency is not as important as the endurance/energy savings that it provides.

In this section, we assess how sensitive SRTP is to RTD's associativity. In order to correlate variations in SRTP performance with shifts in RTD associativity, we maintain a fixed number of entries in RTD. To achieve this, whenever associativity changes, the number of sets is also

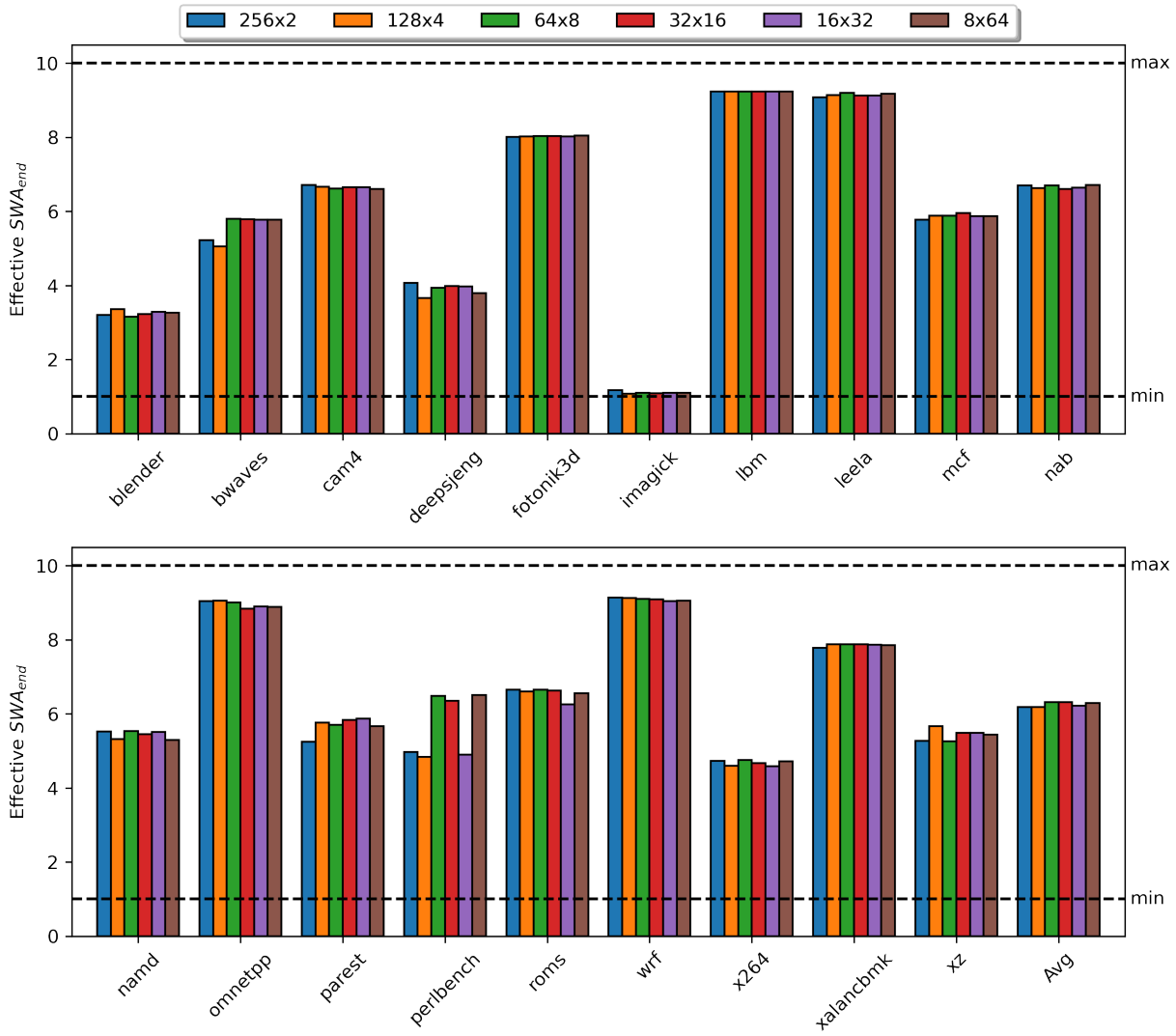

Figure 6.16: Sensitivity of effective SWA_{end} for SPEC CPU2017 benchmarks to associativity of SRTP sets.

changed in the opposite direction so that their product (number of RTD entries) remains constant. We vary the associativity for the base case with 512 RTD entries (32 sets x 16 way associative). Impact of changing the RTD associativity on endurance savings or effective SWA_{end} for SPEC CPU2017 benchmarks is plotted in Figure 6.16. The RTD (sets x associativity) for each configuration is displayed in the legend, with the product's value always set to 512.

The findings indicate that RTD associativity is not a major factor in SRTP's performance.

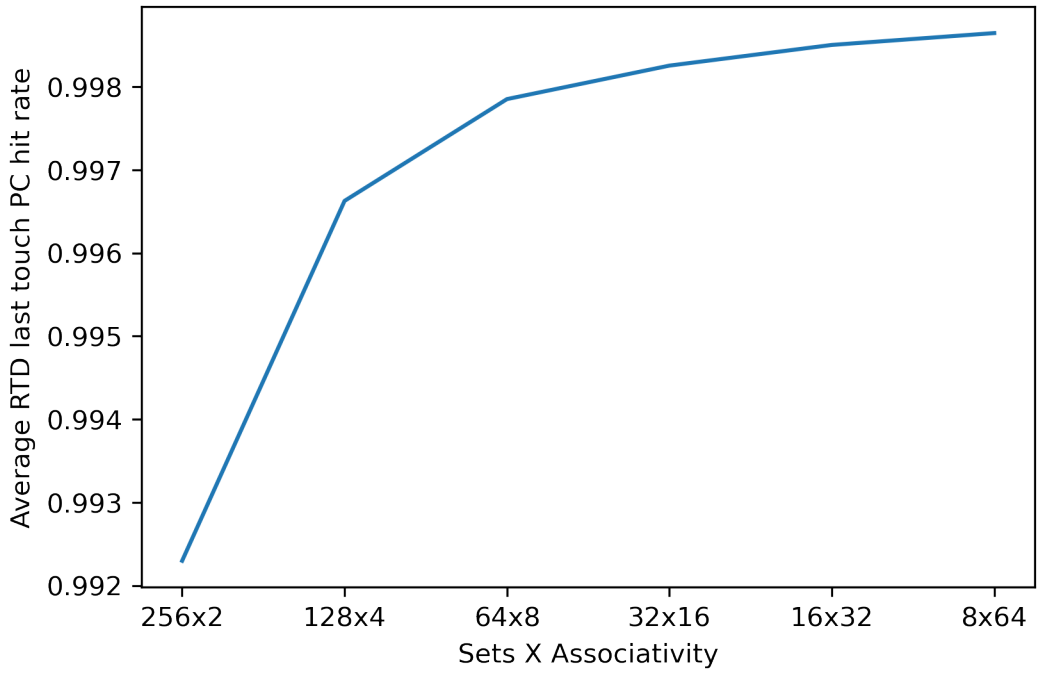

Figure 6.17: The average last touch PC hit rate in RTD for SPEC CPU2017 benchmarks for varying RTD associativity.

The effective SWA_{end} increases from 6.18 to 6.27, when associativity is changed from 2 to 64. Overall, the endurance savings only improve by 1.4% with a 32x increase in associativity. Figure 6.17 plots the average RTD hit rate for various associativity data points. The hit rate is over 99% in all cases emphasizing the observation that high associativity is not a requirement for RTD. This provides insights into the way last-touch store instruction addresses are distributed into RTD sets. The store PCs are spread much more uniformly unlike the data addresses used in regular caches, which are prone to pathological access patterns that put uneven pressure on different sets. Hence, RTD associativity can be reduced to really low values to further bring down design complexity as well as its access latency.

These same observations carry over to the heterogeneous workloads too. As demonstrated in Figure 6.18, SRTP is not very sensitive to RTD associativity for mixed workloads too. In

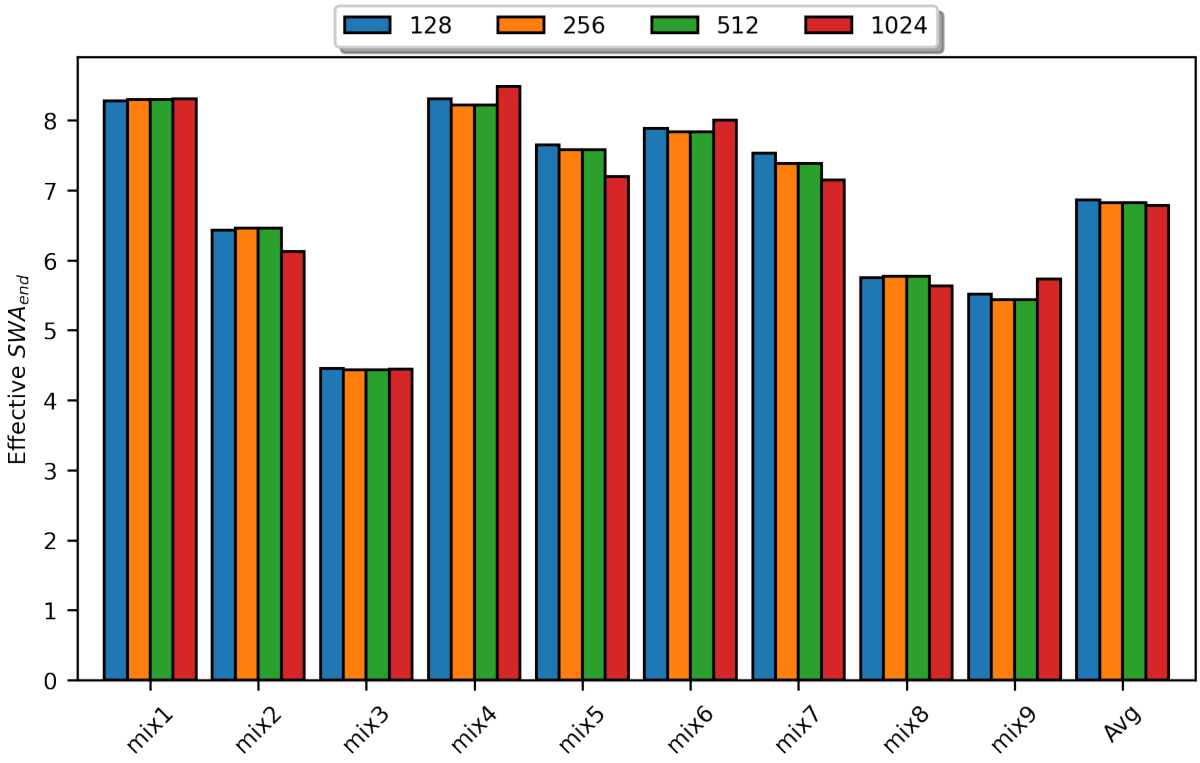

Figure 6.18: Sensitivity of effective SWA_{end} for mixed workloads to RTD associativity.

fact, the variations are even smaller when multiple workloads are present. This makes sense, as the chances are high that different workloads put uneven pressure on different sets in RTD. The resulting conflicts will be more uniformly spread out compared to copies of the same benchmark running together.

6.3.4 Impact of Address Tags in RTD

Another important factor in determining the efficacy of RTD to determine overwrite time values and train SWP modules is the number of data address and timestamp pairs used for each PC entry. As discussed in Section 4.2, an RTD entry is allocated for each last-touch store PC. These entries track two writeback addresses, along with the timestamp of the most recent write operation to those addresses in main memory. When the same addresses are written again, a match occurs

in the RTD. At that point, the overwrite time for the last-touch store PC is calculated using the difference between the current time and previously stored timestamp. Furthermore, the overwrite time is used to evaluate if soft writes are profitable using Inequality 2.9 or 2.15. The soft write decision, in turn, is used to train the SWP modules to decide the type of write that should be used for the store instruction under consideration. Multiple address and timestamp pairs in each RTD entry would allow it to simultaneously evaluate multiple overwrite time candidates, thus increasing the speed with which SWPs are trained. From this point forward, we will refer to the write address and timestamp combination as a 'writeback sample'.

The balance between the overall number of entries and the writeback samples per entry is crucial to the RTD design. For a fixed hardware budget, increasing the writeback samples tracked per entry would result in fewer RTD entries overall, and vice-versa. Having more entries in RTD increases coverage by enabling the tracking of more last-touch PCs. Conversely, increasing the writeback samples for each RTD entry enables SWP to be trained more quickly for a specific static store instruction. In this section, we study the impact of this trade-off between coverage and speed for the SPEC CPU2017 benchmarks. The experiments involve increasing writeback samples per RTD entry from 1 to 8. An increase in the writeback samples is accompanied by a proportional reduction in the number of RTD entries. For example, when the writeback samples are increased from 1 to 2 per entry, the total number of RTD entries is reduced from 1024 (64x16) to 512 (32x16). The number of entries is changed by modifying the number of sets while maintaining the same associativity. The 512 entry case with 2 writeback samples per entry corresponds to the baseline SRTP configuration used throughout this work.

Figure 6.19 plots the effective SWA_{end} for the RTD configurations with varying writeback sample storage described earlier. The effective SWA_{end} improves initially from 5.98 to 6.27,

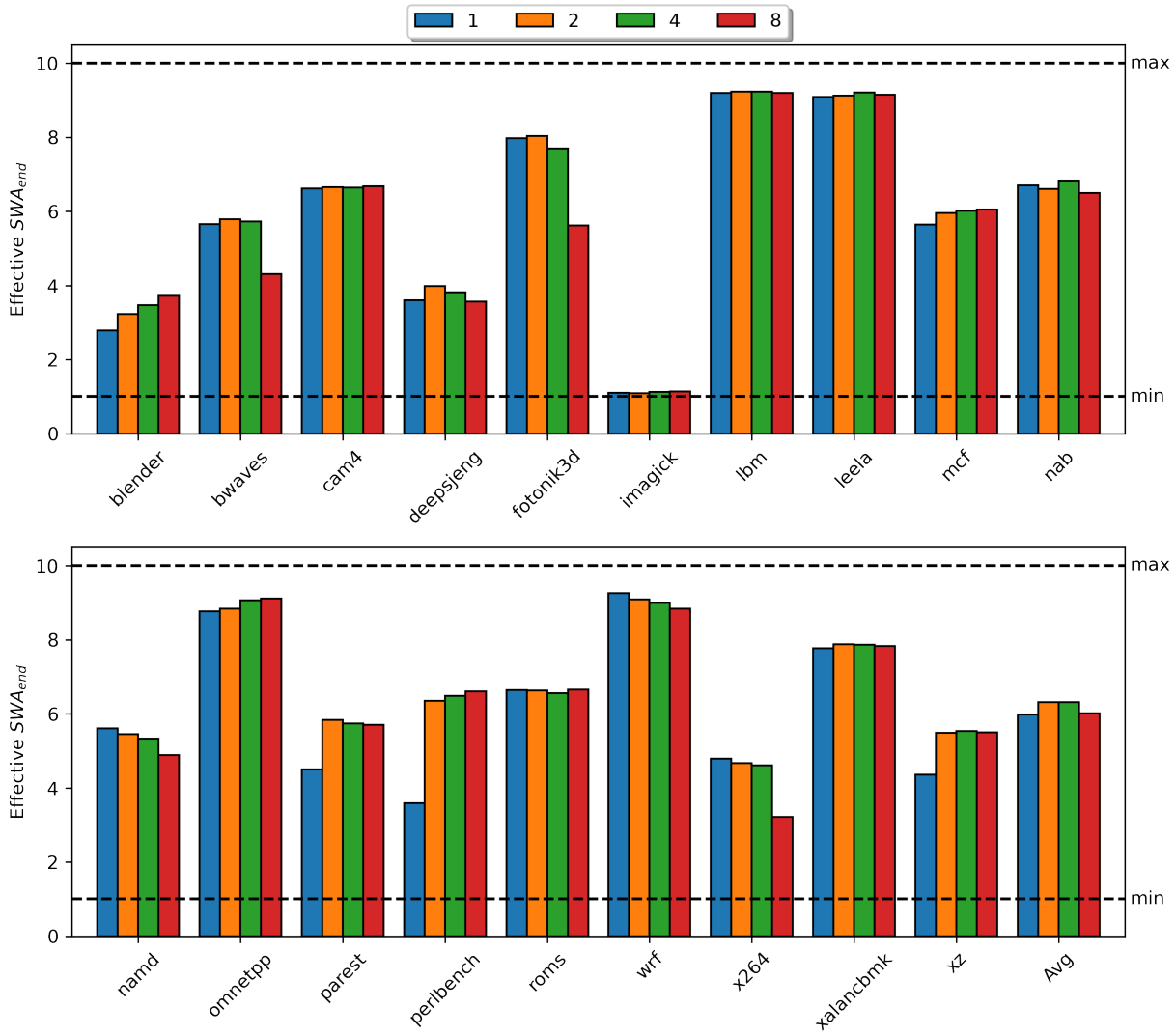

Figure 6.19: Sensitivity of effective SWA_{end} for SPEC CPU2017 benchmarks to the number of writeback samples in each RTD entry.

when the writeback samples in RTD entries are increased from 1 to 2. This improvement can be attributed to RTD providing more SWP training updates. We note that the write address and timestamp pairs per entry, at least when they are few in number, have a greater influence on the SWP training speed than the number of RTD entries. This is evident from Figure 6.20, which plots the average number of SWP training updates from RTD for SPEC CPU2017 benchmarks for the configurations being studied in this section. The SWP training updates rise almost linearly

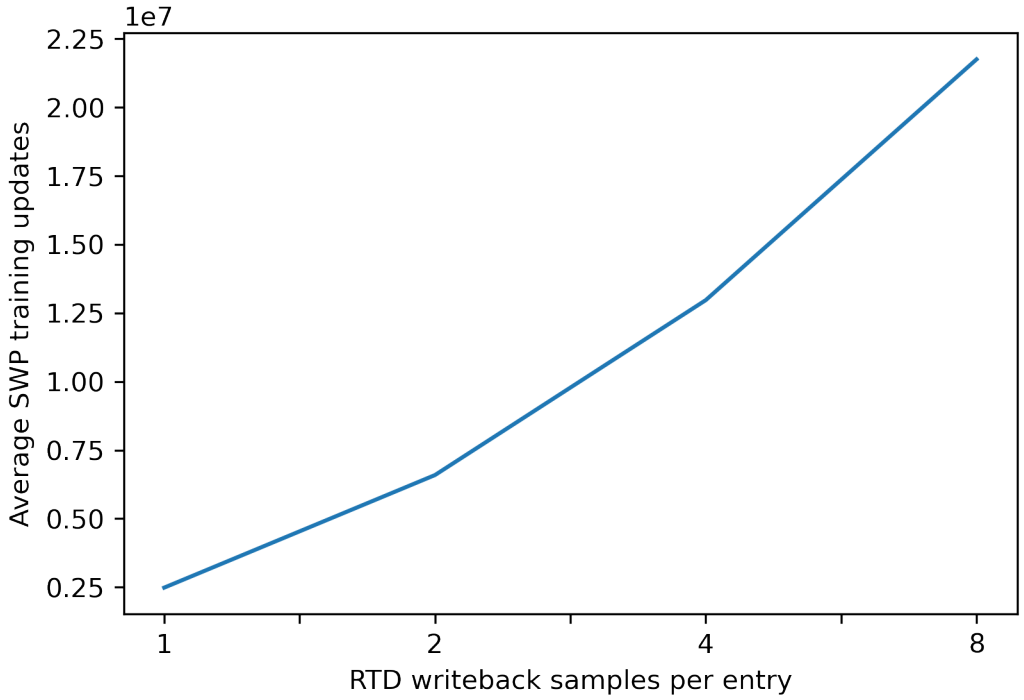

Figure 6.20: The average SWP module training updates from RTD for SPEC CPU2017 benchmarks while varying writeback samples and RTD entries.

as the writeback samples are increased, despite a reduction in the total number of RTD entries. Therefore, sampling more addresses for each RTD entry enables SWP modules to be trained faster for every last-touch PC. However, as the number of writeback samples is increased beyond 2, the endurance gains begin to level off, as shown in Figure 6.19. The speed of SWP training for each PC has diminishing returns beyond a certain point.

At the same time, the impact of decreased last-touch PC coverage caused by reduction in the RTD entries starts to show after a certain threshold. This is demonstrated in Figure 6.21, which plots the average last-touch PC hit rate in RTD for the benchmarks and the configurations under study in this section. Beyond 4 writeback samples per entry, there is a sharp drop in the hit rate due to limited coverage of last-touch PCs in the RTD. This results in reduction of endurance results when 8 writeback samples are taken per RTD entry, while the gains from faster SWP

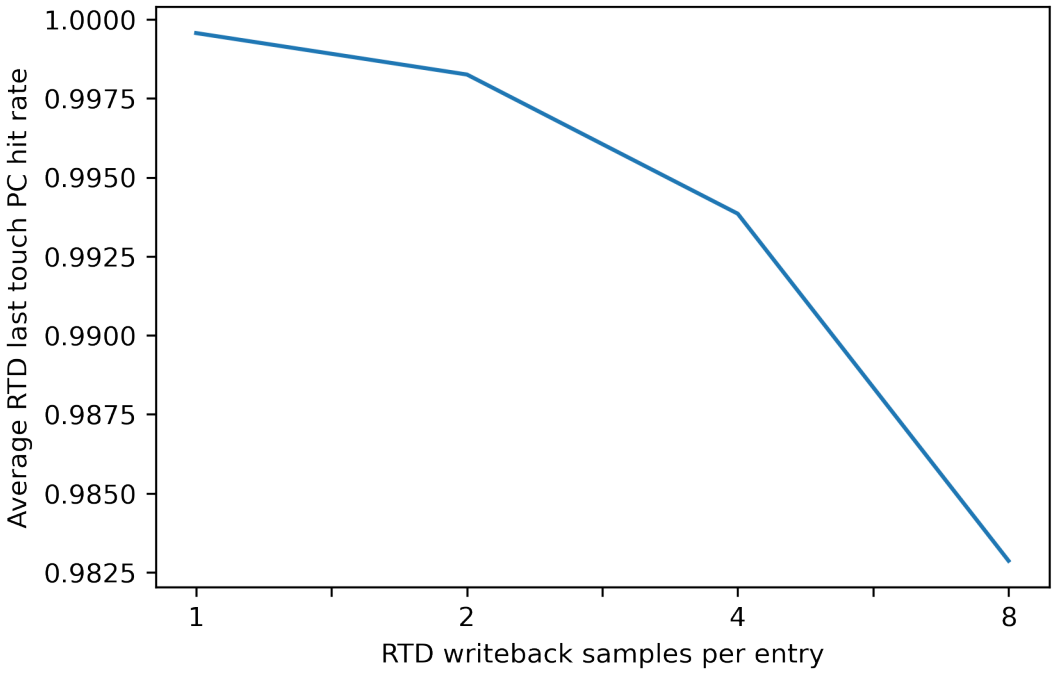

Figure 6.21: The average last-touch PC hit rate in RTD for SPEC CPU2017 benchmarks while varying writeback samples and RTD entries.

training stagnate. Therefore, for a given hardware budget, striking a balance between coverage and speed in RTD modules is crucial. For the individual benchmarks, having two writeback samples in each RTD entry provides SWP training speed and last-touch PC coverage in RTD that maximizes the endurance gains.

The tipping point between speed and coverage is a bit different for heterogeneous benchmarks. Figure 6.22 illustrates the impact of varying the writeback samples per RTD entry on effective SWA_{end} achieved by SRTP for mixed workloads. As discussed in Section 6.3.2, mixed workloads are more sensitive to the number of RTD entries because they have a more diverse set of last store PC values. Therefore, results for mixed workloads do not suffer as much when we slow down SWP module training by lowering the writeback samples per RTD entry from 2 to 1. While slower training does reduce the gains, the increased coverage from additional RTD

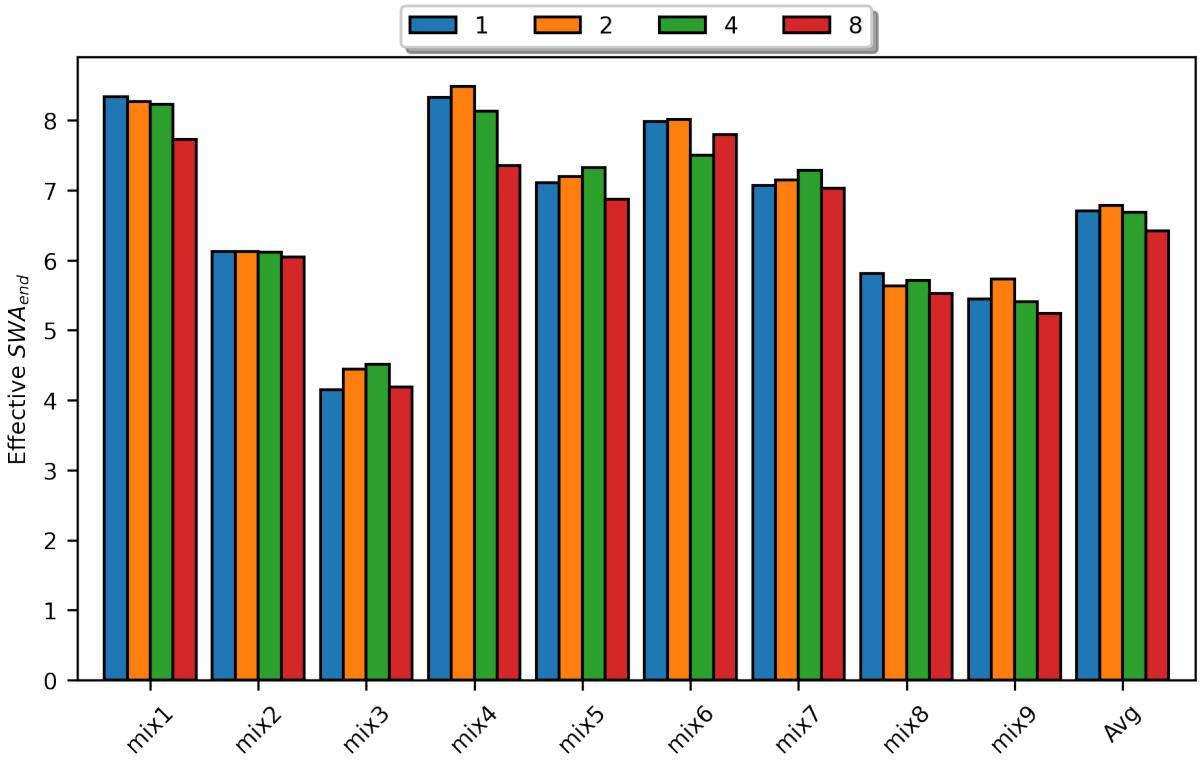

Figure 6.22: Sensitivity of effective SWA_{end} for mixed workloads to the number of writeback samples in each RTD entry.

entries offsets most of it. At the same time, the endurance savings drop rapidly once the write-back samples are increased beyond 2 per RTD entry. This is unlike the homogenous case, where the outcomes didn't start dipping again until 4 writeback samples were added to each RTD entry. As more benchmarks are added to the mix, the balance shifts in favor of slower speed and greater coverage. In both single and mixed benchmarks cases, having two writeback samples per RTD entry provides the best results on average. We use that as the default SRTP configuration because we believe it to be a good equilibrium point in general. If the soft write policy could dynamically modify the speed and coverage ratio to better match the workloads, performance could be enhanced. The average effective SWA_{end} would rise from 6.27 and 6.78 for single and mixed cases to 6.43 and 6.83, respectively, if we let SRTP choose the best configuration for each

benchmark.

6.3.5 Impact of SWP Table Size

Once trained, SWP modules, which are modelled after skewed branch predictors, are used to forecast soft write decisions for store instructions. They have very small overhead, contributing to merely 1.57% of the on die SRAM overhead of SRTP (refer to Table 5.4). As a result, we can allocate oversized tables in SWP modules to prevent aliasing. This section examines how the size of SWP tables affects SRTP performance. Figure 6.23 plots effective SWA_{end} for SRTP while varying the SWP table size from 128 to 1024. Given the small number of primary last-touch store instructions, the number of entries in SWP tables have a minimal impact on endurance results of SRTP. As a result, even when the SWP entries are scaled down by a factor of 8 from 1024 to 128, the average performance stays the same.

When the impact does happen, it is very probabilistic because it depends upon the type of aliasing that takes place. We can divide the aliasing into two categories: constructive and destructive aliasing. The constructive or destructive nature depends upon the soft write decision associated with the store instructions aliasing into a table. If the soft write decisions were the same, then aliasing actually helps by speeding up the training, and we call it constructive. On the contrary, colliding store instructions with opposite soft write decisions will interfere in one another's training, resulting in destructive aliasing. As the majority vote is taken into account, having multiple tables further lessens the impact of collisions in one table. However, they do occasionally happen as evidenced by a 1.3% average drop in performance of the 512 SWP entry case.

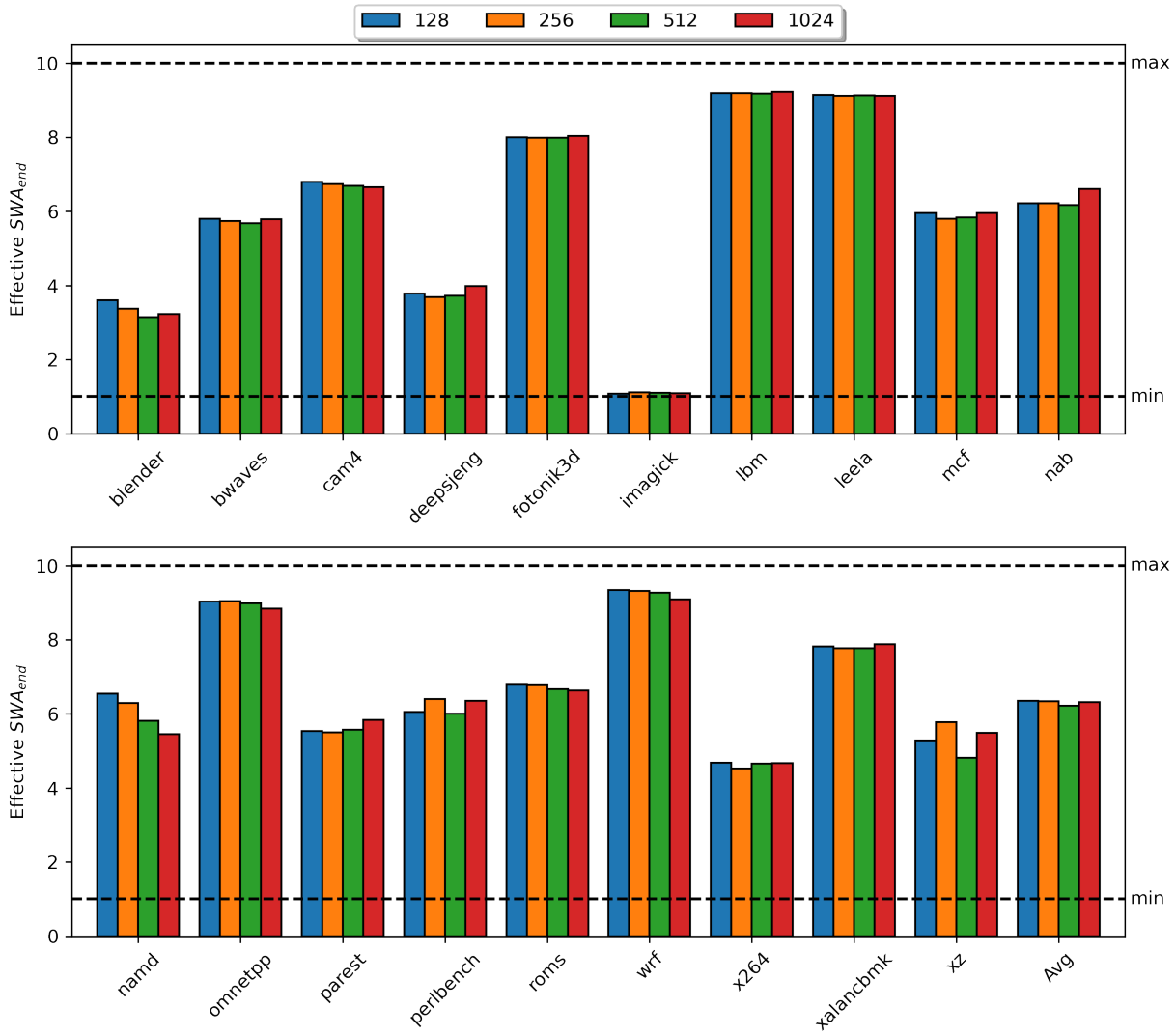

Figure 6.23: Sensitivity of effective SWA_{end} for SPEC CPU2017 benchmarks to the number of entries in each SWP table.

When dealing with heterogeneous workloads, the behavior hardly changes. The results are not significantly affected by SWP table sizes, as shown in Figure 6.24. Destructive aliasing causes a slight dip in performance for 1024 entries instead of 512 entries.

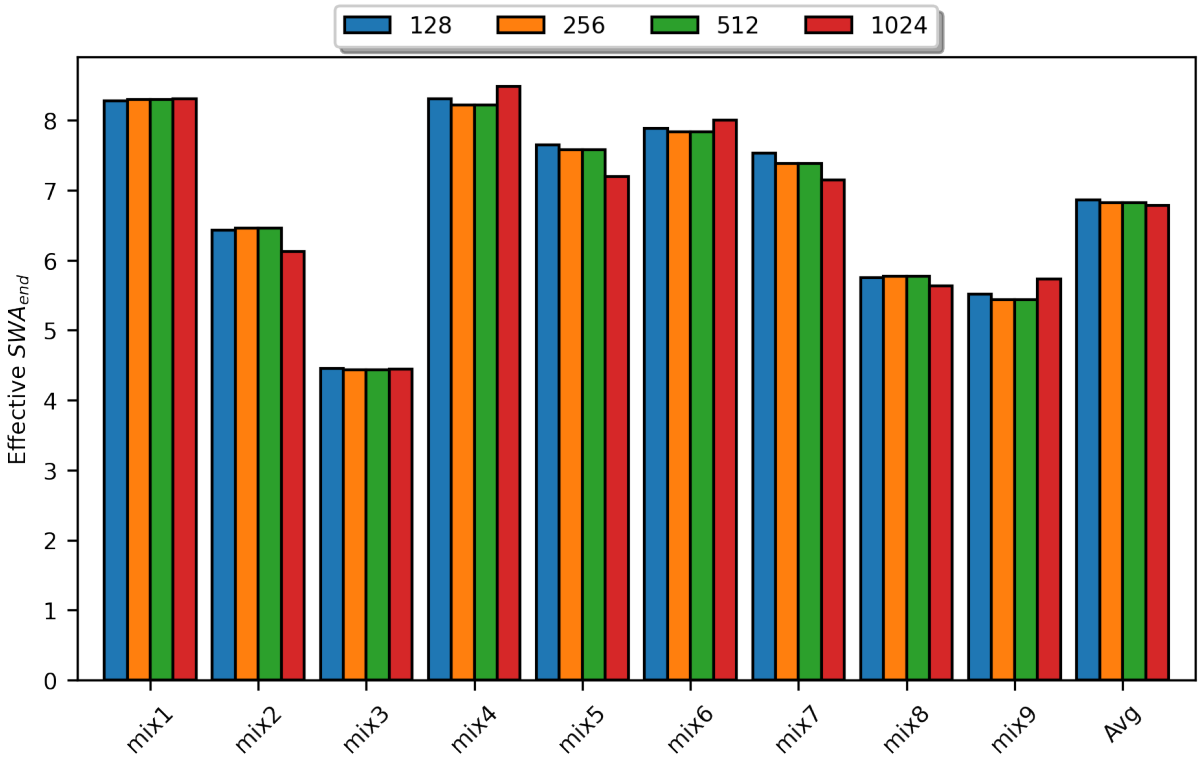

Figure 6.24: Sensitivity of effective SWA_{end} for mixed workloads to the soft write threshold of SWP.

6.3.6 Impact of SWP Aggressiveness

Another important factor is aggressiveness of the SWP modules when assigning soft write labels to static store instructions. Hard writes are the default for all store instructions in SWP. The RTD ascertains overwrite time peaks for last-touch store instructions and relays a soft or hard write decision based on that information to the SWP for training. These training inputs are used by SWP to increase or decrease the saturation counters linked to the store instruction. As each saturation counter exceeds soft write threshold, SWP's confidence increases. When at least two out of the three tables are voting in favor of a specific store instruction, SWP begins setting the soft write bit in cache for that instruction. The soft write bit is consulted to determine the type of write that needs to be performed in main memory on eviction of a dirty block from LLC. The

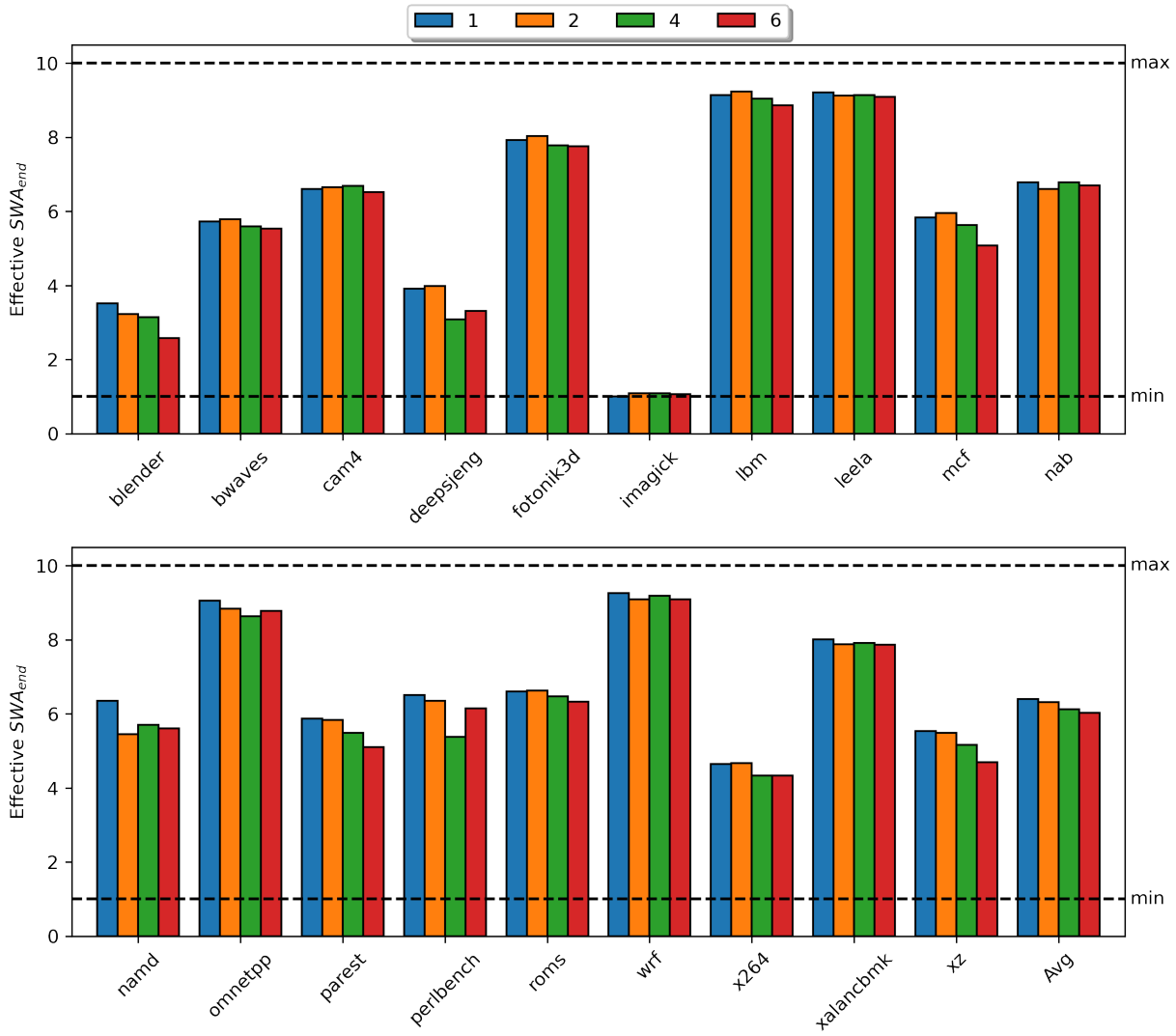

Figure 6.25: Sensitivity of effective SWA_{end} for SPEC CPU2017 benchmarks to the soft write threshold of SWP.

soft write threshold determines how many training samples from RTD must be provided in order for SWP to learn how to make soft write decisions for a specific store instruction. The soft write threshold could be lowered to speed up how quickly SWP modules are trained for each store instruction. But by completing the training on fewer writeback samples, it could also make soft write decisions more inaccurate.

Figure 6.25 plots effective SWA_{end} as a function of the soft write threshold of SWP mod-

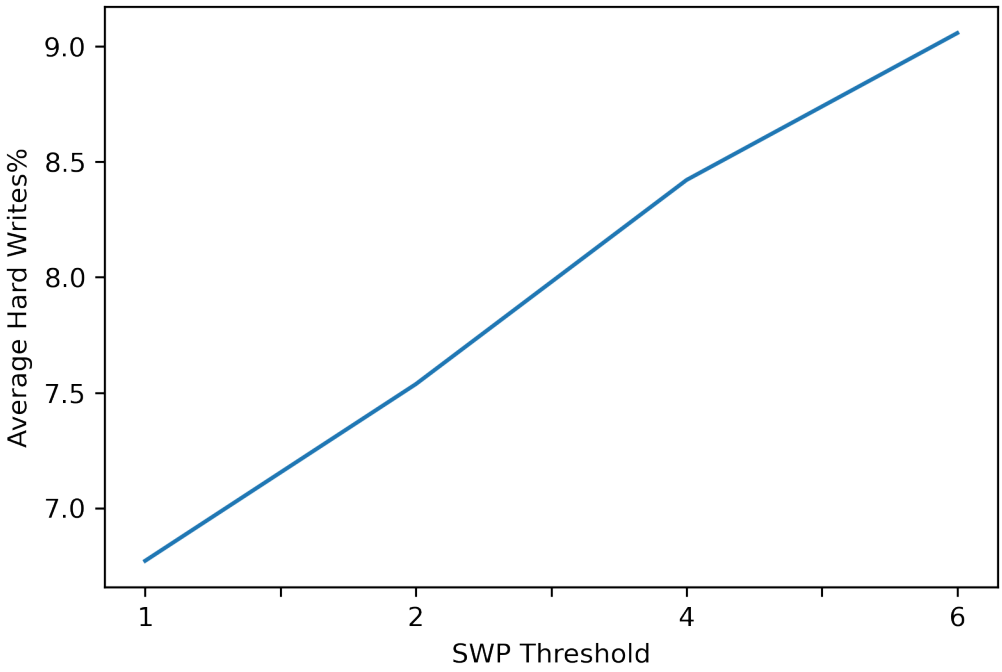

Figure 6.26: Average hard writes allowed by SWP for SPEC CPU2017 benchmarks normalized by the writes of baseline case with only hard writes.

ules. The SWP tables entries are 3 bit saturation counters, which caps their values at 7. Therefore, we only evaluate the soft write threshold between 1 and 6. As mentioned earlier, SWP aggressiveness decreases as soft write threshold value increases. In turn, this causes SWP modules to take longer to learn the soft write decisions for each static store instruction. As a result, SRTP performs more hard writes, as demonstrated in Figure 6.26. It plots the average number of hard writes as a function of soft write threshold, normalized by the total writes from baseline case without any soft write technique. Since baseline writes never change, they are used for normalization instead of write count from soft write techniques which fluctuates with their parameters. The auxiliary hard writes from resets are not included here. Higher threshold values also means that the SWP modules are more certain of the choices they do make regarding soft writes. This results in reduction in the fraction of reset hard writes caused by wrong soft write decisions. The

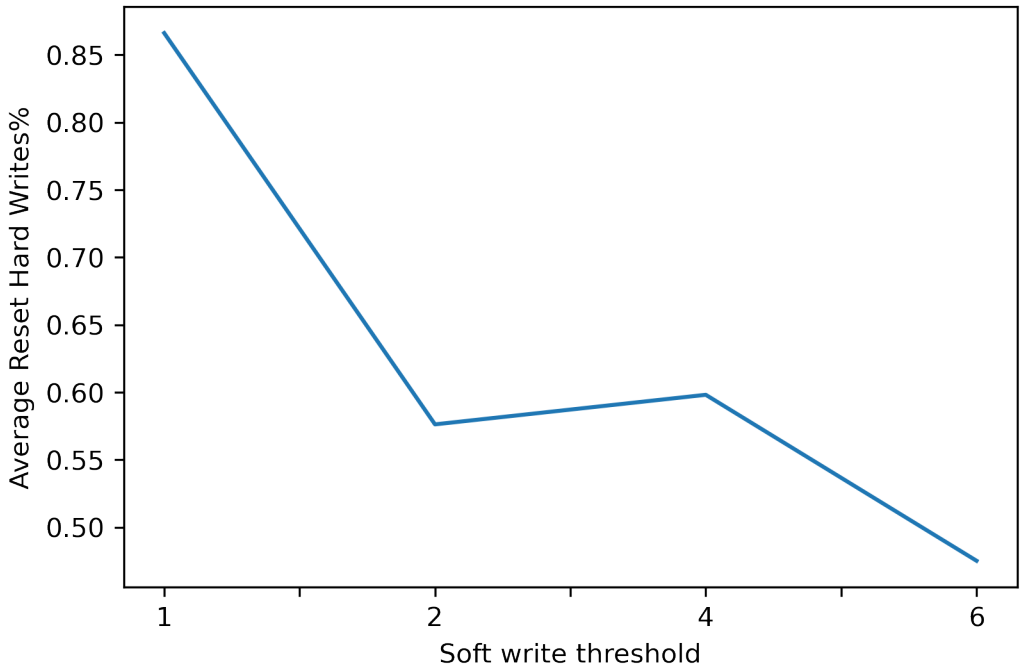

Figure 6.27: Average reset hard writes allowed by SWP for SPEC CPU2017 benchmarks normalized by the writes of baseline case with only hard writes.

average reset hard writes for the SPEC CPU2017 benchmarks as a function of SWP threshold are plotted in Figure 6.27. The values are again normalized by the baseline write count. The rise in hard writes always outpaces the decline in reset hard writes. Therefore, the lower threshold values result in higher effective SWA_{end} in Figure 6.25. The effective SWA_{end} increases from 6.27 for the baseline with a threshold value of 2 to 6.43 when the threshold value is set to 1. The results decline to 6.11 for the threshold value of 6.

Mixed workloads exhibit a very similar sensitivity to the soft write threshold as shown in Figure 6.28. Highest effective SWA_{end} of 6.88 is attained at a threshold of 1, and it decreases as the threshold's value rises.

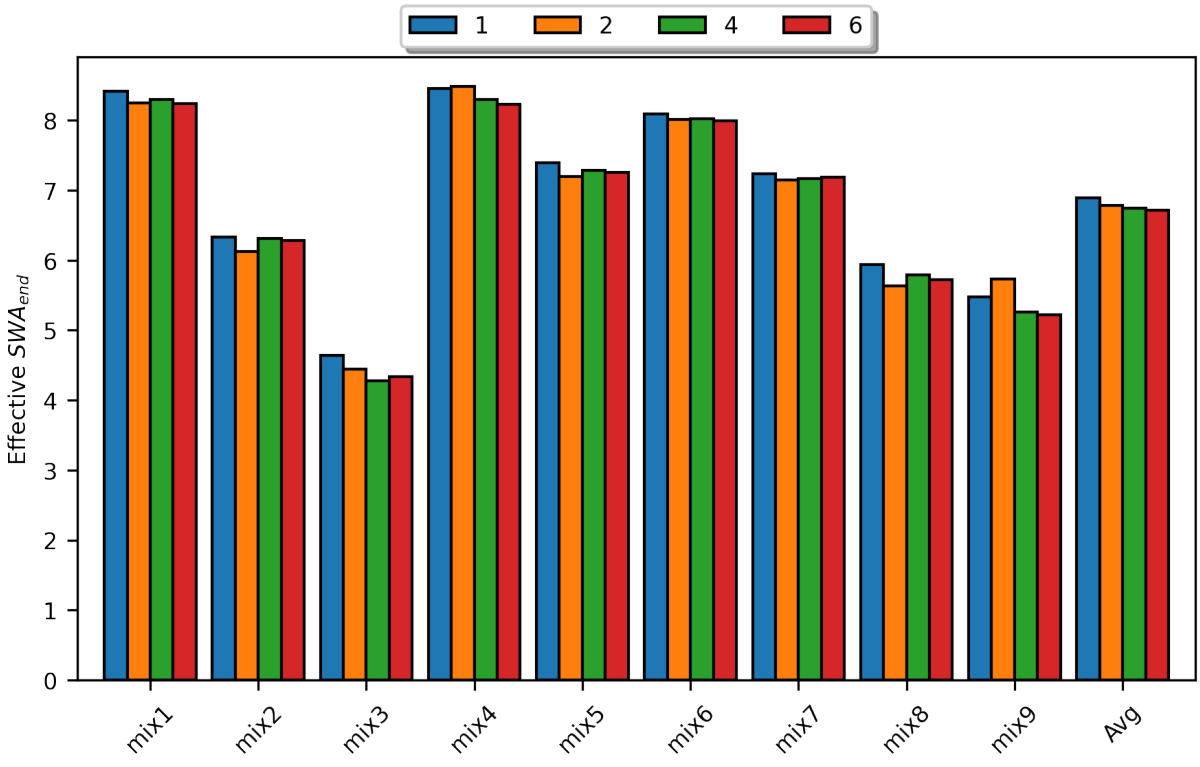

Figure 6.28: Sensitivity of effective SWA_{end} for mixed workloads to the soft write threshold of SWP.

6.3.7 Impact of Retention Time

The sensitivity studies so far considered the impact of SRTP hardware parameters on endurance savings. Now we evaluate the effectiveness of our soft write technique compared to the Oracle when ReRAM cell parameters are altered. ReRAM cells can be fabricated in a variety of ways with different materials. Cells exhibit different properties depending on the materials used to create the oxide layers. It is very likely that retention time will differ for these too. In this section, we study the effects of varying the retention time while keeping the SWA_{end} constant. This will alter the overwrite time threshold until which soft writes are beneficial. Results from this study will provide insights into the effectiveness of SRTP at different retention time values as well as guide any future device-level work in designing ReRAM cells for soft writes.

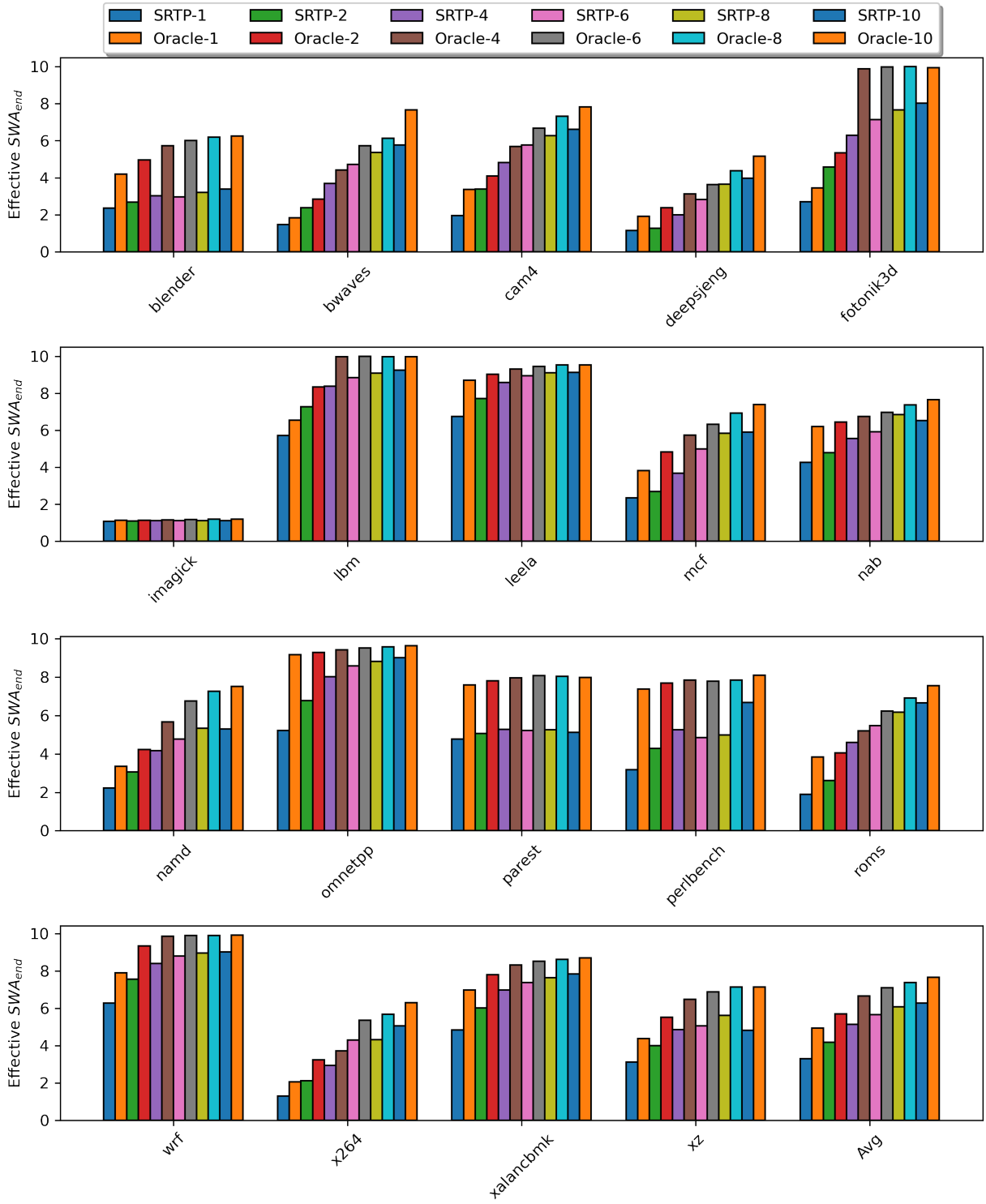

Figure 6.29: Sensitivity of effective SWA_{end} for SPEC CPU2017 benchmarks to the retention time of soft writes. Suffix to the technique represents the retention time value in seconds.

Figure 6.29 plots the effective SWA_{end} for soft write techniques as the retention time is changed from 1 second to 10 seconds. We show results for both SRTP and Oracle at each retention time value. The numerical suffix to the technique name in the legend represents the retention time in seconds; *e.g.*, 'SRTP-1' stands for SRTP with the retention time of 1 second. The effective SWA_{end} decreases for all policies in every case, as we reduce the retention time. It is primarily because of two reasons. First, as retention time decreases, fewer soft write opportunities become available. This is because the threshold overwrite time up to which soft writes are beneficial is directly proportional to retention time, as per Equation 2.3. Hence, as we reduce the retention time, the number of beneficial soft writes also decreases. The second reason is because of the additional cost of refreshes. Reduced retention time means the softly written data needs to be refreshed more frequently. Quantifying the impact of refreshes as retention time decreases is not as straightforward because even if the number of refreshes for individual soft write operations increase, we have a relatively less number of soft writes too. Consequently, depending on how many soft write opportunities were lost, the overall number of refreshes may also decrease.

Figure 6.30 plots the average difference in effective SWA_{end} from Oracle for SRTP at varying retention time values. The gap between Oracle and SRTP widens with decreasing retention time, going from 18% at 10 seconds to 33.2% at 1 second. We plot the average accuracy of SRTP's soft write decisions for the SPEC benchmarks in Figure 6.31. Each soft or hard write decision made by SRTP is compared to an Oracle decision to determine its accuracy. The number of times they concur divided by the total writes gives the accuracy of SRTP. While the accuracy does drop with retention time, it only reduces by 5.1% from 96.5% to 91.4% for 10 and 1 second retention time, respectively. So, inaccuracy in soft write decision-making is not the main reason for the larger difference from Oracle at a smaller retention time.

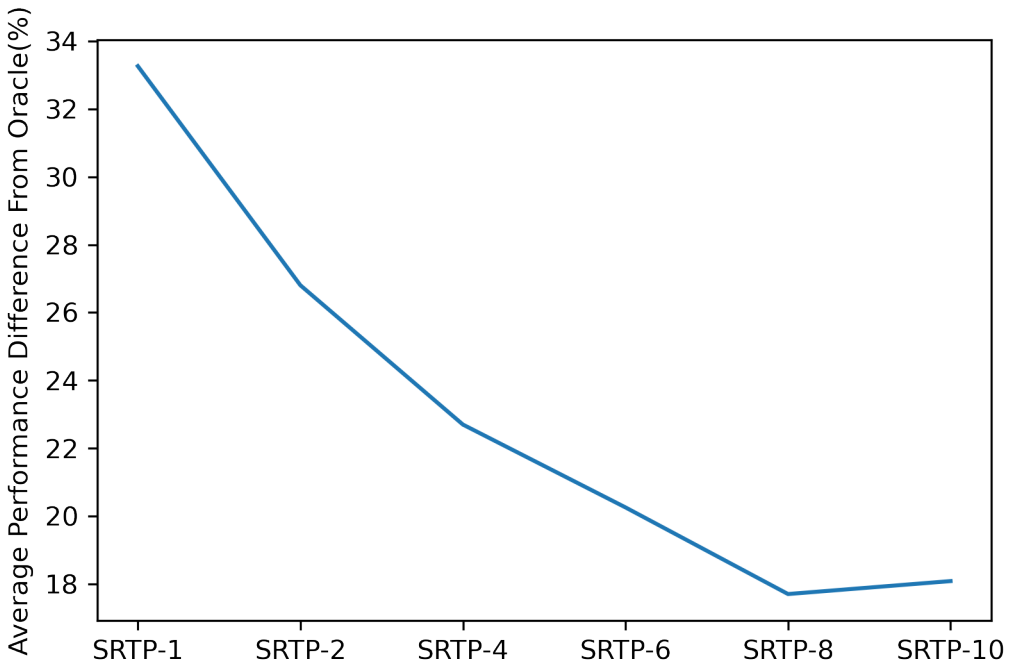

Figure 6.30: Average performance difference of SRTP from Oracle at different retention times.

Figure 6.32 illustrates the effect of more frequent refreshes at a shorter retention time. It plots the average refresh count for SPEC CPU2017 benchmarks at different retention time values, normalized by baseline write count with hard writes only. While the total number of writes varies due to the varying number of auxiliary writes incurred in terms of refreshes and resets, the baseline write count remains constant across all configurations. This makes the baseline write count a good candidate for normalizing refresh count. The average refresh count rises much more sharply for SRTP compared to Oracle at lower retention time values, as shown in Figure 6.32. For instance, SRTP's average refreshes increased by 4.67x when retention time is reduced from 10 seconds to 1 second. On the other hand, the average number of refreshes for Oracle increased by 3.3x times for the same retention time range. As a result, even though the accuracy of soft write decisions by SRTP does not change significantly, its performance gap from Oracle increases at a lower retention time due to a higher number of refresh operations.

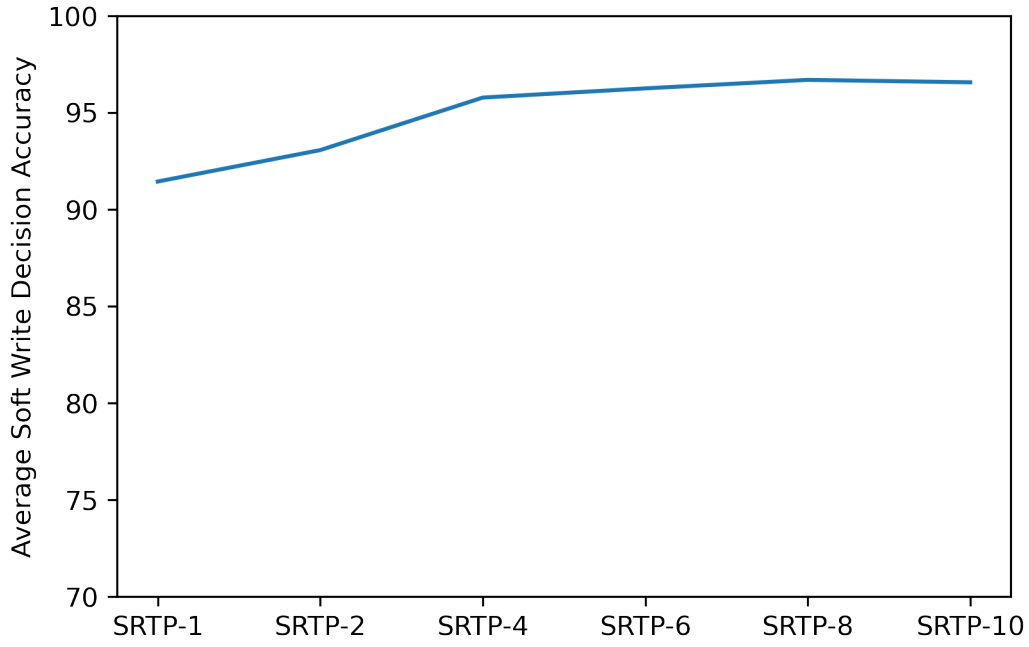

Figure 6.31: Average SRTP accuracy at varying retention time values for SPEC CPU2017.

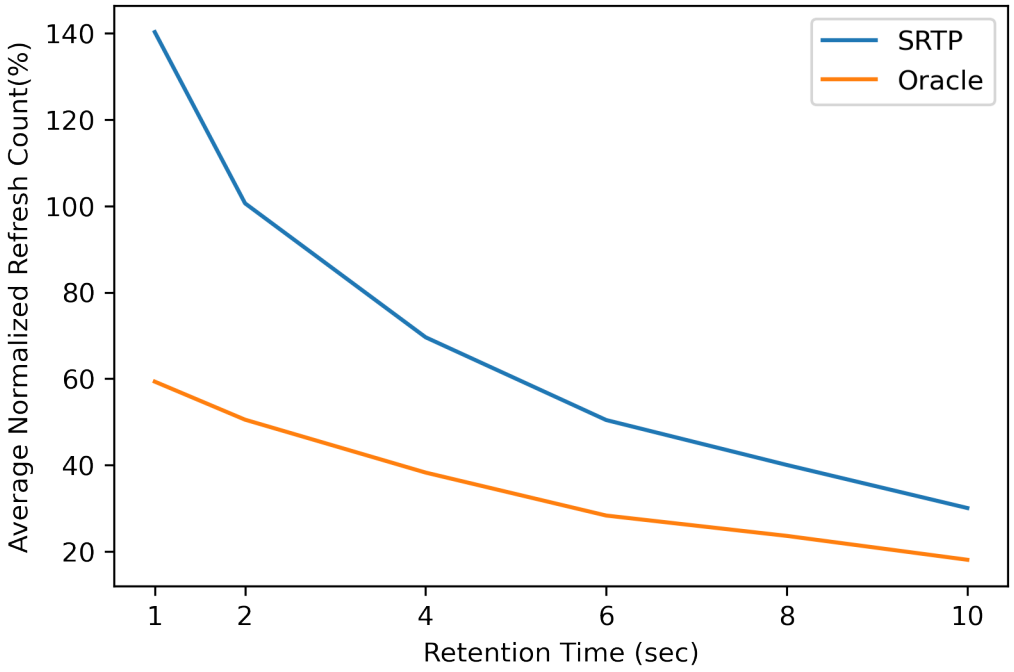

Figure 6.32: Average number of refreshes incurred by SRTP at different retention times normalized against the writes from baseline case without any soft writes.

Chapter 7: Wear-Leveling Integration

In this chapter, we analyze the efficacy of SRTP when integrated with the Ouroboros wear-leveling technique from Section 4.5, and report the actual lifetime achieved. SRTP and wear-leveling policies have a symbiotic relationship. Section 4.5 covered one part of this relationship: SRTP using wear-leveling indirection tables to track softly written data with minimal overhead. The other part is making soft write decisions available to the wear-leveling scheme to improve its accuracy. We first demonstrate the need for updating the wear-leveling techniques to take soft writes into account and then go over the improvements in their performance by doing so. To analyze the performance of wear-leveling techniques, we use the wear-leveling efficacy as the yardstick, which is described next.

7.1 Wear-Leveling Efficacy

Before defining the wear-leveling efficacy, another term, *endurance wear* (EnWr), needs to be introduced. It refers to the memory cell endurance consumed by a write operation. We assign the wear caused by a single soft write as its unit. By that definition, EnWr for hard writes is SWA_{end}. The target of wear-leveling policies is to distribute EnWr evenly across all physical memory locations. Hence, wear-leveling efficacy can be defined as the ratio of total EnWr accrued before memory failure and the maximum EnWr the memory system can take, shown in Equation 7.1.

$$Wear\ Leveling\ Efficacy(\%) = \frac{Total\ EnWr\ before\ memory\ failure}{Maximum\ EnWr} * 100 \qquad (7.1)$$

For a given program and memory size, the total EnWr before memory failure is obtained by running the program repeatedly until the number of defective lines is greater than the spare lines. Given the requirement for at least a few years of lifetime before failure and slow simulation speeds, running the program until system failure is not practical. We instead use extrapolation to determine the total EnWr caused by the benchmarks. Since Start-Gap has been demonstrated to be highly effective in regions much larger than our page size, we assume perfect wear-leveling within a page for simplicity and do the extrapolation to examine wear-leveling across pages by Ouroboros.

As discussed in Section 2.3.3, Ouroboros performs migrations at the end of a global epoch (100 million writes). Hence, simulating the ordering of writes within an epoch is not necessary for extrapolation as the policy does not care about that. It only cares about simulating the global epochs in order. We collect per logical page write profiles for the benchmarks at every global migration epoch of Ouroboros. This is a list of the number of writes each logical page receives within a global epoch. Then during extrapolation, these profiles are run repeatedly in order until all spare frames (128) are worn out. The extrapolation code simulates an Ouroboros translation table. For each global epoch profile, it looks up the entry for logical pages that received any writes. Then, the demand and usage counters for those entries are updated. After that, it simulates the migration policy of Ouroboros to remap a few high-demand logical pages to new physical pages. The translation table is also updated to reflect these migrations. If the usage counter of

an entry exceeds the maximum value, the corresponding physical page is marked dead and is substituted with a spare frame. When the policy runs out of spare frames, the next physical page failure is where memory failure occurs. That's when the extrapolation code stops, and the EnWr accumulated until then is the numerator in Equation 7.1. The maximum EnWr only depends on the memory size and per line endurance and is given by (Number of lines * Per line endurance * SWA_{end}). Wear leveling efficacy is estimated by plugging this information into Equation 7.1.

7.2 Write Count vs Wear Out

The main goal of wear-leveling techniques is to uniformly distribute the wear out caused by writes. Under normal circumstances, all writes have the same impact on endurance wear out. Evenly distributing the write count is equivalent to evenly distributing endurance wear out. As a result, all wear-leveling policies aim to evenly distribute the write count across physical memory locations. When soft writes are added to the mix, the relationship between write count and endurance wear out breaks because hard and soft writes have a different impact on the endurance wear of a cell. This is illustrated in Figure 7.1, which plots the fraction of endurance wear different types of writes contribute to the benchmarks with SRTP. Hard writes account for only 6% of the write count(see Figure 6.2) but cause 23% of endurance wear. Consider two pages that have been written the same number of times, but one has received only hard writes and the other only soft writes. Judging by the number of writes, both pages will appear to have the same wear out, but in reality, the page with hard writes has suffered 10 times (or SWA_{end} times) more wear. So, if the wear-leveling policy is not made aware of the type of writes, it will impact its ability to uniformly spread out the endurance wear.

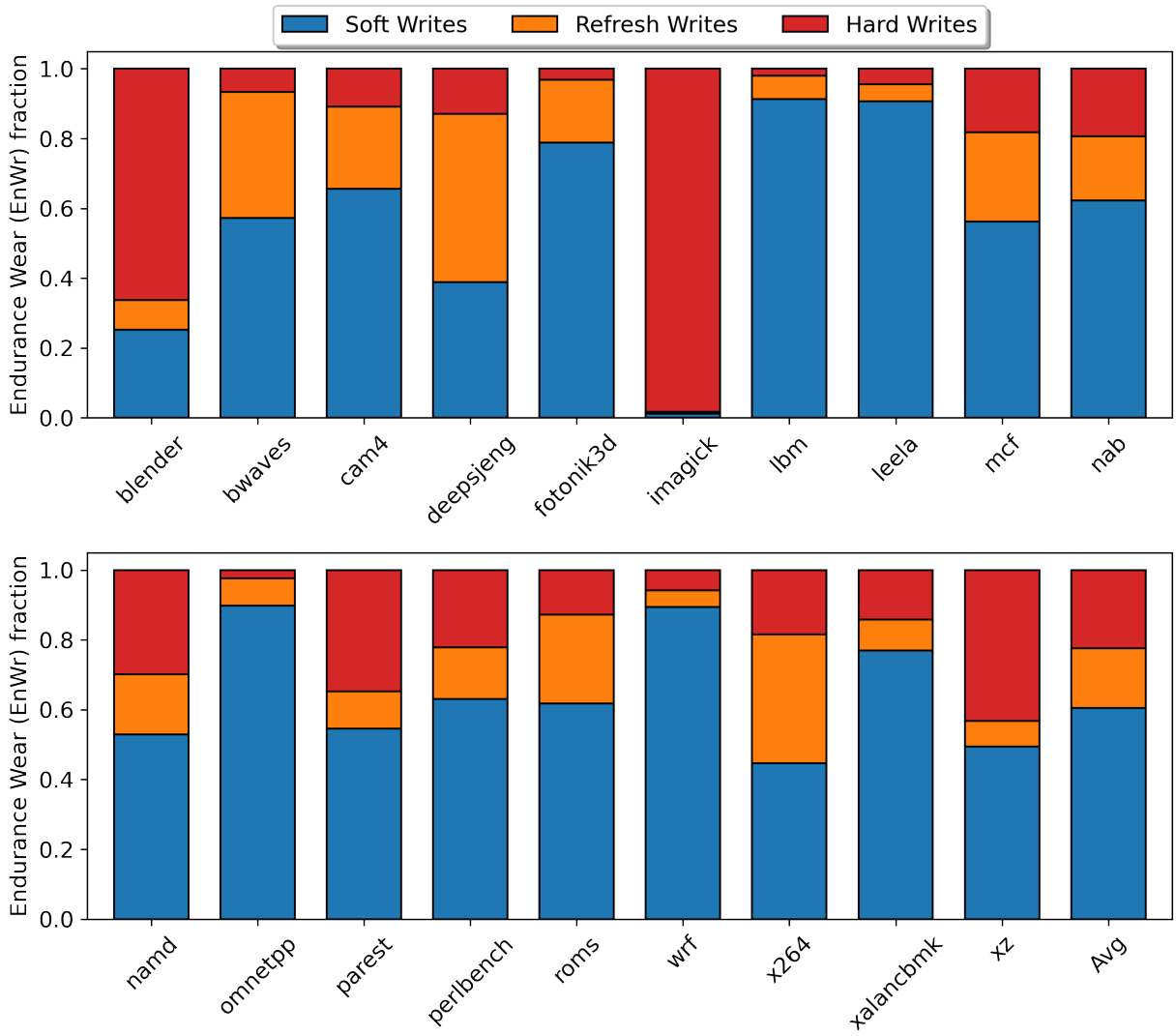

Figure 7.1: Normalized Endurance Wear (EnWr) by different types of writes for SPEC CPU2017 benchmarks with SRTP.

When integrating with wear-leveling techniques, we propose to use endurance wear as a guiding metric instead of write count, given that soft and hard writes cause uneven wear out. The traditional policy of just considering write count will negatively impact the lifetime a wear leveling technique can achieve. To make Ouroboros consider wear out, demand and usage counters are incremented by the EnWr of the writes instead of their counts. So, soft writes and refreshes cause increments of 1, and hard writes cause increments of SWA_{end} (10). This allows Ouroboros

to make migration decisions based on the endurance wear of pages instead of write count (WC). To ascertain the improvement in wear leveling efficacy because of this, we simulate two versions of Ouroboros for each soft-write policy. The first one uses write count as the guiding metric and is represented by the suffix "-WC" to the policy name. The second one uses endurance wear and is represented by the suffix "-EnWr". Figure 7.2 plots the wear-leveling efficacy for both of these techniques with RRM-base, RRM-aggr, and SRTP.

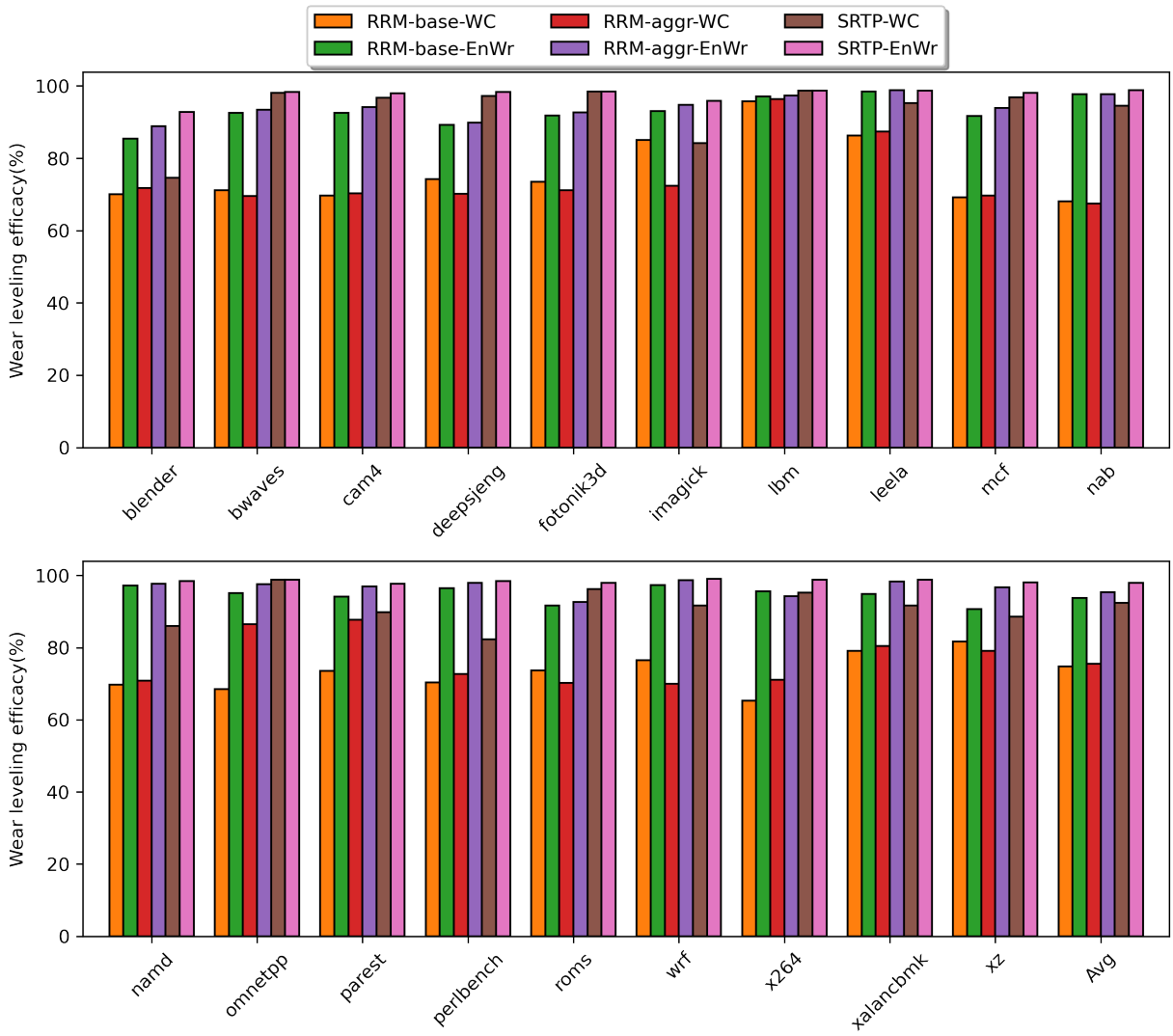

Figure 7.2: Wear-leveling efficacy of Ouroboros using write count (WC) and Endurance Wear (EnWr) as guiding metrics for different soft write techniques.

The wear-leveling efficacy for SRTP improved from 92.3% to 98% when write count was substituted with endurance wear as the guiding metric. This demonstrates that only using write count in wear -eveling techniques leaves some gains in the lifetime improvement on the table. This is especially true for the benchmarks with a significant portion of hard writes in Figure 6.2. Since larger the fraction of hard writes with disproportionate endurance wear, the more inaccurate write count becomes as a metric to make wear-leveling decisions. So, not only do those benchmarks achieve lesser gains from soft write techniques, but also, the wear-leveling performance takes a hit. For example, blender has an effective SWA_{end} of 3.1x with SRTP, and the wear-leveling efficacy with write count is also just 74.6%. By adjusting the wear-leveling metrics to take soft writes into account, we are able to increase the wear-leveling performance back up to 92.8%.

The difference between the two wear-leveling policies is even larger for RRM since it has more hard writes in Figure 6.2. On average, RRM-base-wc and RRM-base-EnWr have wear-leveling efficacy of 75.2% and 93.8%, respectively. RRM-aggr is also very similar, with the efficacy of 75.9% and 95.3% for write count and endurance wear based wear-leveling, respectively. Hence, depending upon the soft write technique employed, the endurance wear metric can improve the average wear-leveling performance by 19.4%. This further corroborates our claim regarding the inability of traditional wear-leveling techniques to perform well in the presence of soft write methodologies. But, using the new guiding metrics proposed in this thesis, the wear-leveling performance can be bought back to be close to ideal. To the best of our knowledge, this is the first time the impact of writes with different wear out is considered on wear-leveling techniques.

7.3 Lifetime

The lifespan of non-volatile memory systems is an important measure due to constrained per-cell endurance. The technology must have a practical lifespan before it can be used in the real world. The extrapolation runs can be used to estimate memory lifetime with various benchmarks. The number of runs of the benchmark from extrapolation and the runtime for a single simulated run gives us the lifetime of the memory system if the program was run repeatedly until failure.

Figure 7.3 compares the lifetime achieved by the soft write policies against the baseline system with just hard writes. For each soft write technique, we report the lifetime when using the write count metric versus the endurance wear metric to guide wear-leveling, labeled with suffixes "-WC" and "-EnWr", respectively. The geometric mean is used instead of the arithmetic mean to report the average results. This is done to reduce the excessive skew caused by results for a few benchmarks like imagick, whose low write intensity results in really long lifetimes.

The lifetime is directly proportional to both effective SWA_{end} achieved and the wear-leveling efficacy. It all boils down to how many writes the memory system can take before all the spare lines are consumed since more writes mean more runs of the program resulting in longer lifetimes. Soft writes reduce endurance wear of individual writes, allowing the memory system to accommodate more writes. Wear-leveling allows the memory system to come close to the maximum number of writes it can take.

The dependence on wear-leveling efficacy is demonstrated by the similarity in the lifetime and efficacy results for the two wear -leveling policies simulated with the same soft write technique. The average lifetime for SRTP-WC and SRTP-EnWr is 13.72 and 14.59 years, respectively. It is the same 6.3% improvement as the wear-leveling efficacy from Figure 7.2. Similarly,

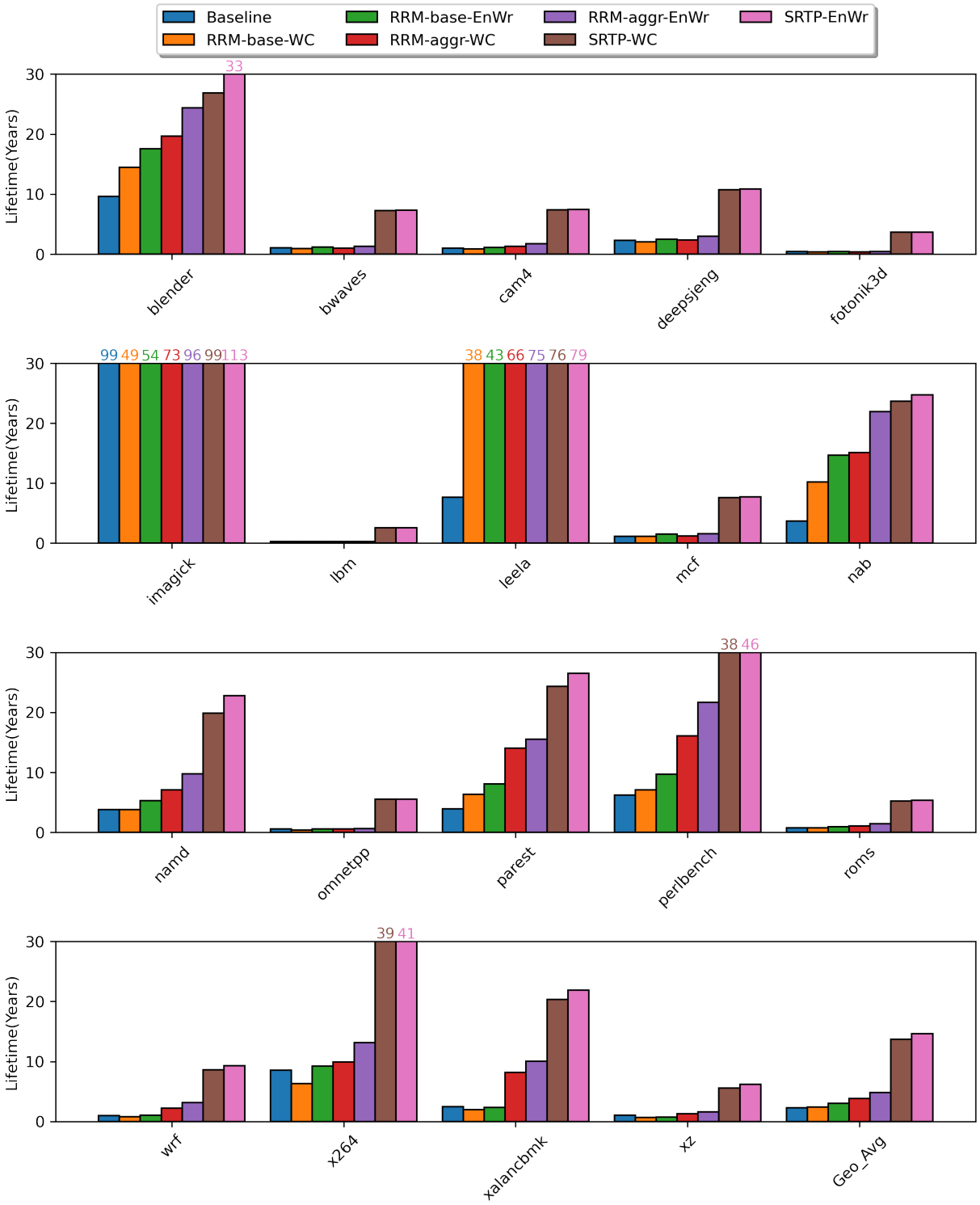

Figure 7.3: Extrapolated lifetime

for RRM-base, the endurance wear metric improves the lifetime from 2.38 years to 3 years. RRM-aggr sees the lifetime increase from 3.8 years to 4.83 years with the new metric. These improvements are also very close to the wear-leveling efficacy gains obtained by using endurance wear for these techniques.

Overall, SRTP improves the average lifetime of ReRAM from 2.26 years of the baseline to 14.59 years. With wear-leveling efficacy close to 100%, these results can be primarily attributed to endurance savings in terms of effective SWA_{end}. The 6.2x effective SWA_{end} translates to a 6.4x improvement in memory lifetime. The memory lifetime improvements over baseline for RRM-base, and RRM-aggr also agree with their effective SWA_{end} at 3 years (1.3x) and 4.83 years (2.2x), respectively. Any slight differences could be attributed to the variations in wear leveling efficacy. These results tie the figure of merit proposed in this thesis for endurance improvement to a real-time metric important for non-volatile memories. Any gains in effective SWA_{end} increase the lifetime of the memory system by the same factor, assuming wear leveling efficacy remains constant.

So far, we have discussed average results, which are crucial to understanding the expected lifetime of non-volatile memory chips. However, in the real world, the memory system will be exposed to all sorts of write patterns, and it will be expected to function reliably for a reasonable duration. Therefore, in order to assess the viability of NVMs, the worst-case lifetime must also be taken into account. RRM and baseline are able to achieve a few years of lifetime on average. But when they are exposed to write intensive programs (higher WPKI in Table 5.2), their lifespan shrinks to much less than a year. In the case of lbm, the best they can do is 0.29 years by RRM-agg-WC. SRTP, on the other hand, still manages to get 2.6 years of lifetime in the worst case. SRTP outperforms the state of the art while achieving a reasonable lifetime in all cases, enabling

more robust implementations of non-volatile memories.

7.4 Wear-Leveling Sensitivity Studies

This section further dives into the sensitivity of the wear leveling and lifetime results to multiple workloads, as well as the spare lines present in the memory system.

7.4.1 Mixed Workloads

Thus far, the evaluations only considered the effectiveness of wear-leveling schemes with a single workload running continuously. In this section, we assess a more realistic system with multiple concurrent workloads. This study uses the multiprogrammed workloads from Section 6.1.4. To determine the wear-leveling efficacy, write profiles were gathered and extrapolated as described in Section 7.1. Figure 7.4 plots the wear-leveling efficacy for the mixed workloads. The results for each benchmark are divided into the three soft write techniques, namely RRM-base, RRM-aggr, and SRTP. These are further split into the different versions of Ouroboros simulated in this work based on the guiding metric, denoted by the suffixes 'WC' and 'EnWr', which stand for write count and endurance wear, respectively.

We find the conclusions from Section 7.1 still hold true in the presence of multiple workloads. As more workloads are added, the effectiveness of wear-leveling for soft write techniques does not change all that much. SRTP-EnWr improves wear-leveling results from 93.3% of SRTP-WC to 98.3%, showcasing that endurance wear is a superior guiding metric for wear-leveling. Due to a higher percentage of hard writes, RRM-base and RRM-aggr still lag behind significantly with write count metric at 72.2% and 69.9%, respectively. The majority of their wear-leveling

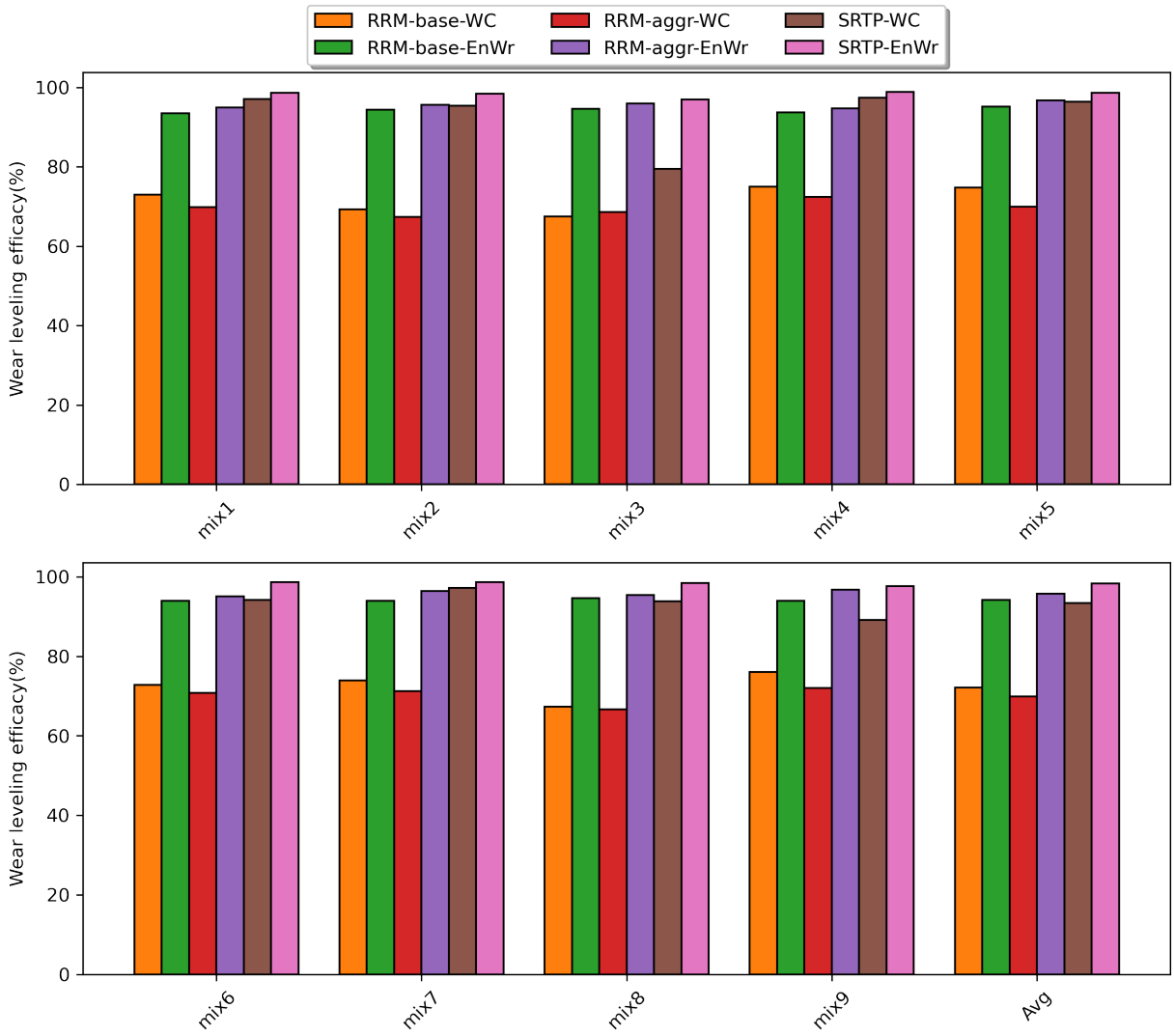

Figure 7.4: Wear-leveling efficacy of Ouroboros using write count(WC) and Endurance Wear(EnWr) for mixed workloads and various soft write techniques.

performance can be recuperated using the endurance wear metric. It raises the wear-leveling efficacy of RRM-base and RRM-aggr up to 94.2% and 95.7%, respectively.

While there is little difference in the wear-leveling efficacy between single and mixed workloads, lifetime results are another matter. The memory system lifespan for multiprogrammed workloads with different soft write policies and wear-leveling techniques is plotted in Figure 7.5. For a given benchmark and soft write technique, the difference between the lifetime results of

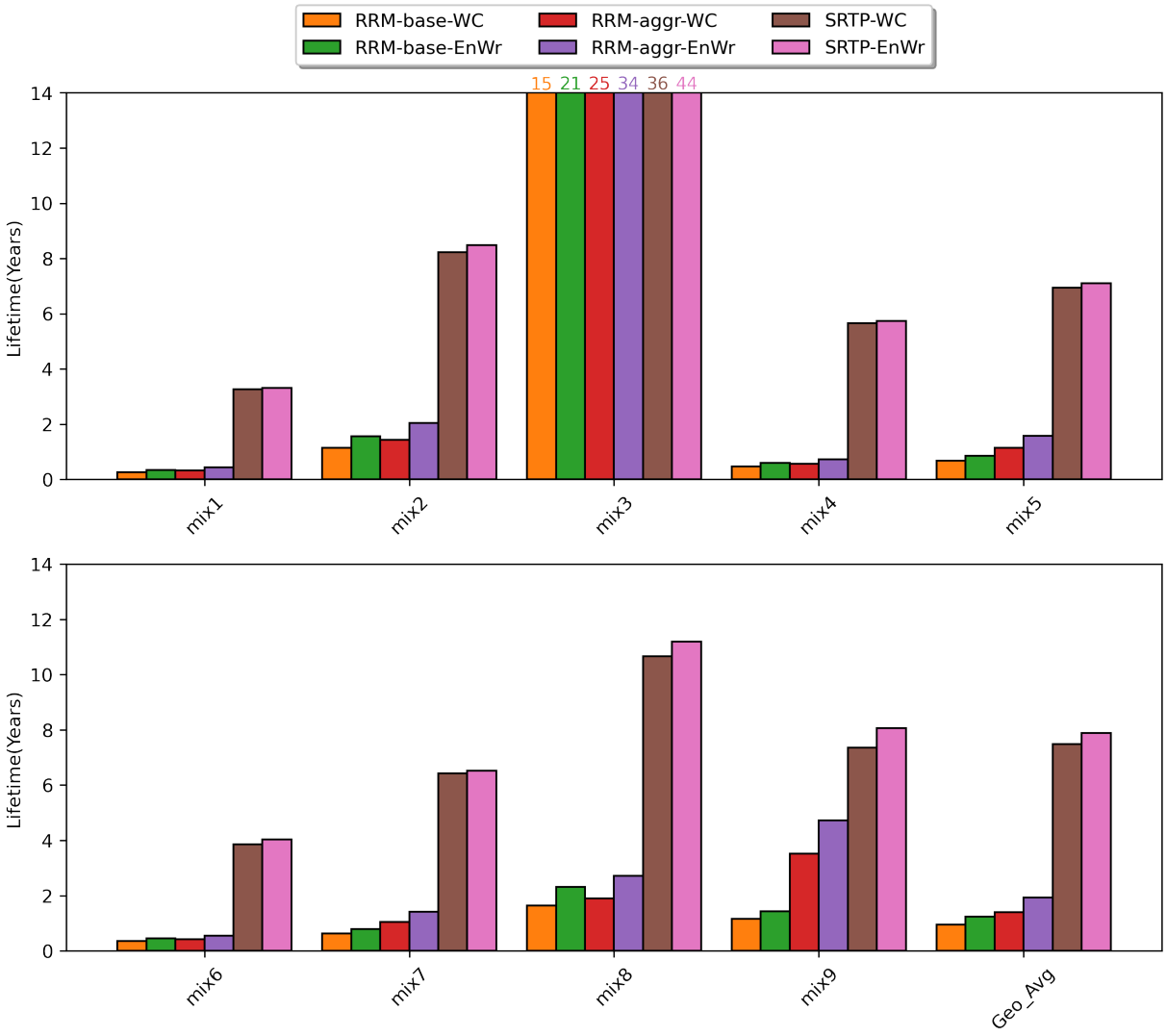

Figure 7.5: Extrapolated lifetime for mixed workloads.

two wear leveling policies is the same as the difference in their efficacies from Figure 7.4. Since the differences in the two wear-leveling techniques don't have anything new to add, the rest of this discussion focuses only on the lifetime results corresponding to endurance wear based wear-leveling for brewity.

Mixed workloads see a reduction in the average lifetime of all techniques. This is particularly true for RRM, with the lifespan of its baseline and aggressive versions falling below two years, at 1.24 and 1.92 years, respectively. SRTP's lifetime also reduces from 14.59 years for in-

dividual benchmarks to 7.9 years for mixed workloads. The lifetime, in general, is shorter when multiple programs are running, even though the effective SWA_{end} and wear-leveling efficacy are similar to the single workload case. This is because, on average, mixed workloads experience higher write intensity. Frequent writes lead to a shorter lifetime. The mixed workloads were created by combining benchmarks with different write intensities (refer to Table 6.1 and Table 5.2). We observe that the presence of a few or all write-intensive benchmarks in a mix (mix1, mix4, mix5, mix6, mix7) ends up dominating the write intensity and skews the lifetime on the shorter side. The only exception is the case where all the benchmarks have low write intensity (mix3).

Assuming benchmarks of all write intensities are equally likely to run, this can be broken down into a balls and bins problem where our 4 cores are the bins, and the programs are the balls. For each bin, we choose one ball randomly and throw it in. The chances of having all 4 programs with low write intensity is $\frac{1}{3^4}$ or 0.012, whereas the chance of having at least one write-intensive program out of four are $1 - (\frac{2}{3})^4$ or 0.8. Therefore, chances are, some write-intensive programs will appear in the mix. Running only a few write-intensive programs skews the lifespan of the memory system to the worst-case lifetime value from Section 7.3, making the worst-case results even more important. The lifespan of RRM-base when write-intensive programs are present in the mix is consistently below 1 year. This leaves much to be desired from the lifetime achieved by the state-of-the-art. SRTP bridges that gap successfully with a worst-case lifetime of 3.3 years.

7.4.2 Sensitivity to spare frames

Non-volatile memories with limited write endurance are typically equipped with some spare cells. These spare cells keep the memory operational when a few lines are corrupted. Significant work

has been done in bad block management for flash memory to identify the blocks about to fail and replace them with fresh ones [102–105]. The slack that spare lines afford wear-leveling by allowing the memory to function normally in case of some failures makes them an integral part of non-volatile memory systems.

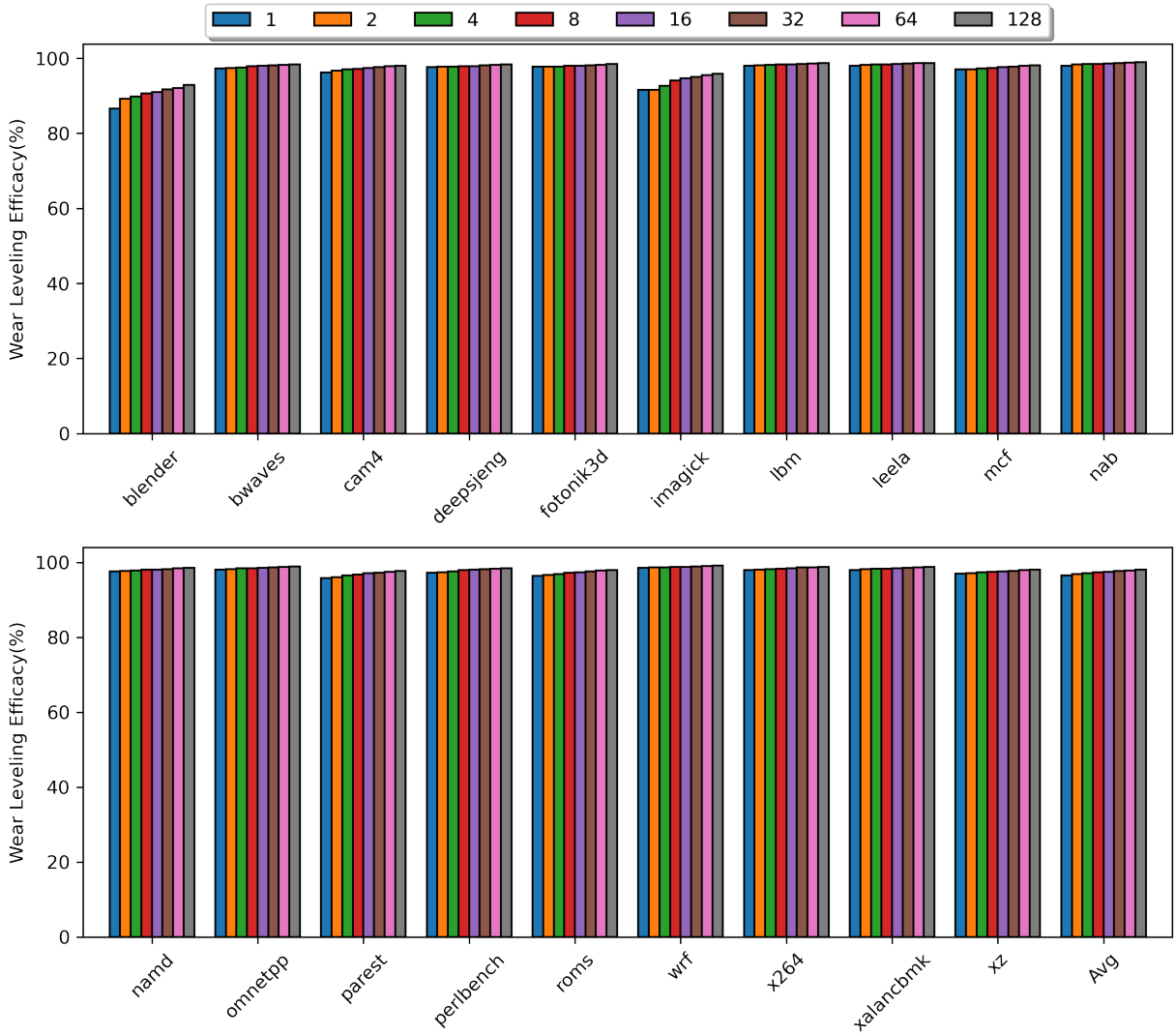

Figure 7.6: Wear-leveling efficacy variations with spare lines for endurance wear directed Ouroboros running with SRTP.

Ouroboros has spare frames the same size as its pages (8 KB). The wear-leveling efficacy and lifetime results so far assumed 128 spare frames. In this section, we examine the impact

of spare frames on wear-leveling techniques. Figure 7.6 plots the wear-leveling efficacy for endurance wear guided Ouroboros running with SRTP (SRTP-EnWr) as we change the number of spare frames from 1 to 128. Ouroboros is not very sensitive to spare frames for the majority of benchmarks, as demonstrated by a 1.46% gain in average wear-leveling efficacy as spare frames go from 1 to 128. But, a few benchmarks like blender and imagick show higher sensitivity to spare frames. Wear-leveling efficacy for blender goes from 86.5% to 92.8% as the spare frames are increased from 1 to 128. Spare frames are needed to handle outliers like these instead of the average case.

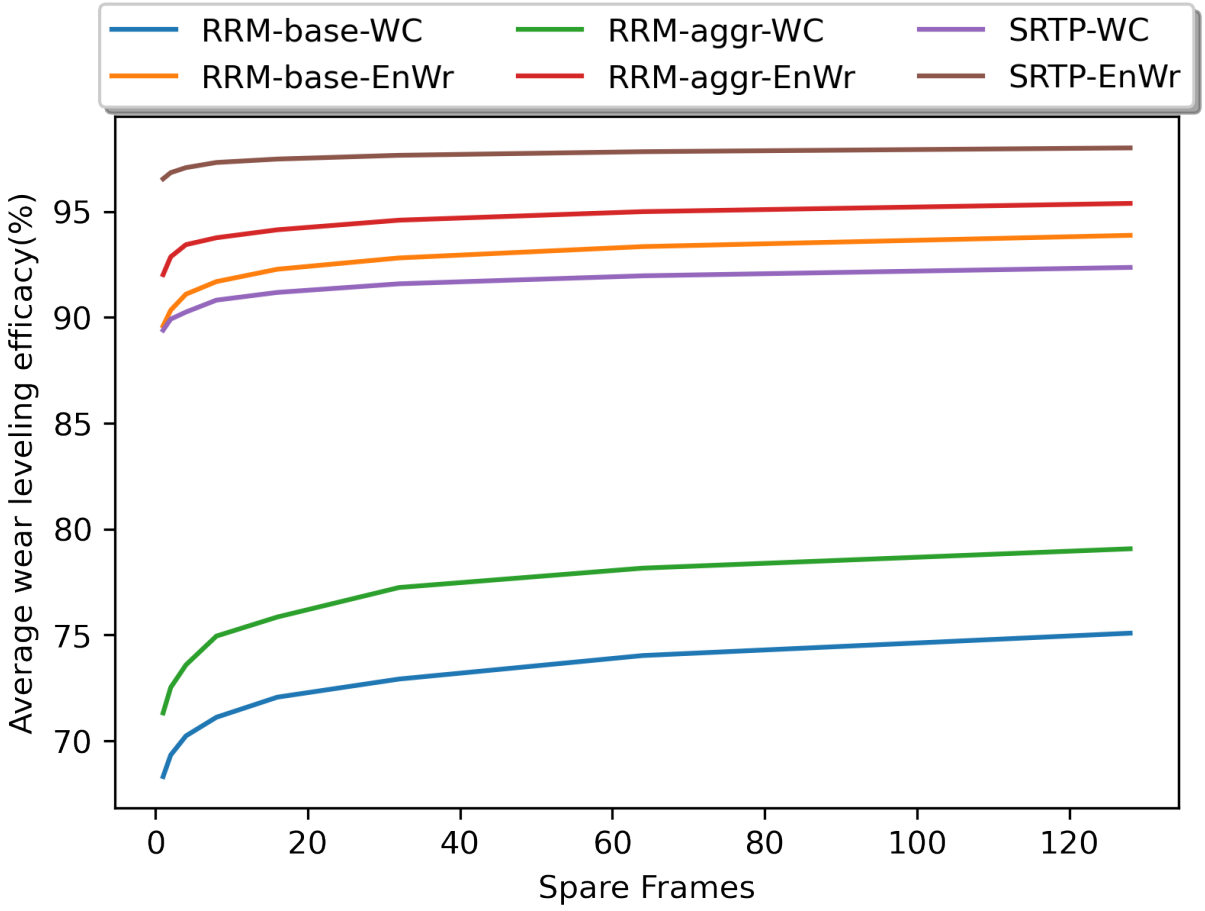

Figure 7.7: Average wear-leveling efficacy variations with spare frames for different soft write technique and wear-leveling combinations.

The trends for the remaining combinations of soft write techniques and wear-leveling techniques are very similar. Hence, in the interest of space, we only plot variations of the average bars with spare lines in Figure 7.7. The plots show that the wear leveling efficacy improves for at least a few initial spares but then plateaus after 40 spare frames. Also, write count directed Ouroboros is more sensitive to spare frames as its performance drops more in all cases when spare frames are reduced.

Chapter 8: Variable Retention

The tradeoffs between a ReRAM cell's endurance and retention were covered in Section 2.2. This includes the exponential relationship between retention time and conductive filament diameter in Equation 2.3. Additionally, the conductive filament area was inversely proportional to SWA_{end}. Keep in mind that SWA_{end} denotes the factor of reduction in endurance wear caused by a soft write for a given retention time. Therefore, decreasing the diameter of the conductive filament results in an exponential reduction in retention time. At the same time, soft writes with lower retention cause proportionally lower endurance wear too. Figure 8.1 plots the retention time and its corresponding SWA_{end}. These parameters, along with overwrite time values, determine when to perform soft writes, how frequently the softly written cells are refreshed, and the overall endurance savings they provide.

In order to assess the soft write techniques, discussions and experiments up to this point have only used one data point from the endurance-retention tradeoff. Depending upon the prominent overwrite time distances in a program, this one retention fits all approach leaves some endurance gains on the table. For instance, the retention time of 10 seconds might be excessive if the overwrite time for the majority of writes is much shorter than that. These writes might benefit from a more aggressive reduction in the conductive filament area, which would lead to less retention that is still adequate to cover those overwrite times and provide higher SWA_{end}. Similarly,

programs with large overwrite time values will benefit from larger retention time values. In this chapter, we explore the improvement in SRTP endurance gains by using a retention-endurance data point customized for each program.

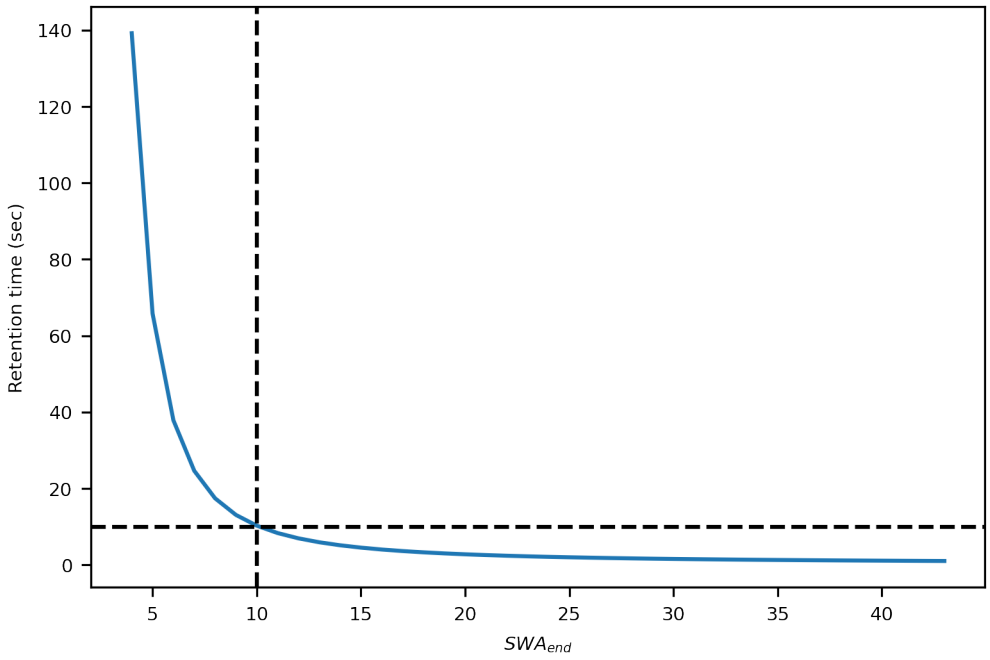

Figure 8.1: Retention vs soft write advantage for endurance.

8.1 Best Retention Time Per Program

To identify the retention time value that yields the best endurance results for a program, we collect the overwrite time profile for each benchmark. For this, the programs are simulated in the oracle mode, and we register the frequency of different overwrite time values encountered during its execution. For each write to main-memory, there is an entry in the overwrite time profile. It is recorded when the memory location modified by the write is written to again in the future. At that point, the time difference between the two writes is the overwrite time value associated with the former write. For the very last write to each memory location, including the writes for locations

written once, we assign an infinite overwrite time value. The oracle already calculates this value using a dedicated timestamp for each memory location, as described in Section 3.2. We combine these into a histogram called the overwrite time profile of a benchmark.

Once the overwrite time is known for a write, we can determine the ideal soft or hard write decision for any given retention time and SWA_{end} pair using Inequality 1.1 (or Inequality 2.15, depending upon the optimization objective). If it was a soft write, the number of refreshes incurred could also be calculated using Equation 2.6. With the write breakdown, *i.e.*, number of soft writes, refreshes, and hard writes, known Equation 2.17 gives the effective SWA_{end}. In this way, the overwrite time profiles can be used to determine the effective SWA_{end} that the oracle can achieve for various retention time values and their corresponding SWA_{end} without having to simulate them.

We sweep the retention time from 1 to 100 seconds and use the overwrite time profile to calculate the effective SWA_{end} for each case. This process is repeated for every benchmark to find the retention time value that maximizes effective SWA_{end}. This is the optimal retention time for the benchmark with the Oracle scheme. Table 8.1 displays the best case retention time value for each benchmark and the corresponding SWA_{end}. Since all benchmarks, with the exception of one, have optimal retention time values much lower than 100 seconds, we decided to limit the maximum value of the retention time sweep to 100 seconds. At 100 seconds retention and its associated SWA_{end}, the majority of soft write candidates are already covered. Therefore, retention time values above it won't be able to take advantage of any new soft write opportunities. However, their corresponding SWA_{end} will continue to drop, effectively reducing the benefit of each soft write. As a result, except imagick, the effective SWA_{end} drops monotonically for all benchmarks beyond 100 seconds.

Benchmark	Optimal retention time (seconds)	SWA_{end}
lbm	1	43.13
fotonik3d	3	18.96
omnetpp	1	43.13
roms	1	43.13
mcf	1	43.13
wrf	1	43.13
bwaves	11	9.6
cam4	1	43.13
xz	1	43.13
xalancbmk	1	43.13
deepsjeng	20	7.57
nab	1	43.13
parest	1	43.13
namd	1	43.13
perlbench	1	43.13
leela	1	43.13
blender	1	43.13
x264	8	11.2
imagick	100	4.4

Table 8.1: Best case retention time for each benchmark and the corresponding soft write advantage in endurance.

The lower limits of the sweep are more constrained by the operational integrity of ReRAM cells. Reduction in retention time also means reduced conductive filament area. Beyond a certain limit, the filament would cease to be continuous, making the ReRAM cell non-operational. These edge cases, where the assumption of a continuous filament will be violated, are not considered in the retention-endurance tradeoff model. The cell will act like an insulator even in the low resistance state, if there is not a continuous filament bridging the top and bottom portions of the cell. The equations used to derive the relationship between retention time and endurance do not hold because the conducting behavior is implicit in their derivation, which is why the ReRAM cell is not functional in such circumstances.

Experimental data is available supporting ReRAM operation at 10x reduced filament area,

which is the base case for soft writes in this work. Unfortunately, it has not been investigated to what extent the filament area can be decreased before the ReRAM cell ceases to function as a conductor, even in the SET state. We use 1 second as the lower limit for retention time sweep. At this point, the conductive filament area has been shrunk by 43 times. Figure 8.1 shows the retention time almost plateaus beyond SWA_{end} of 43.12, which corresponds to 1 second. It is reasonable to assume that beyond that point, the retention-endurance model does not hold either.

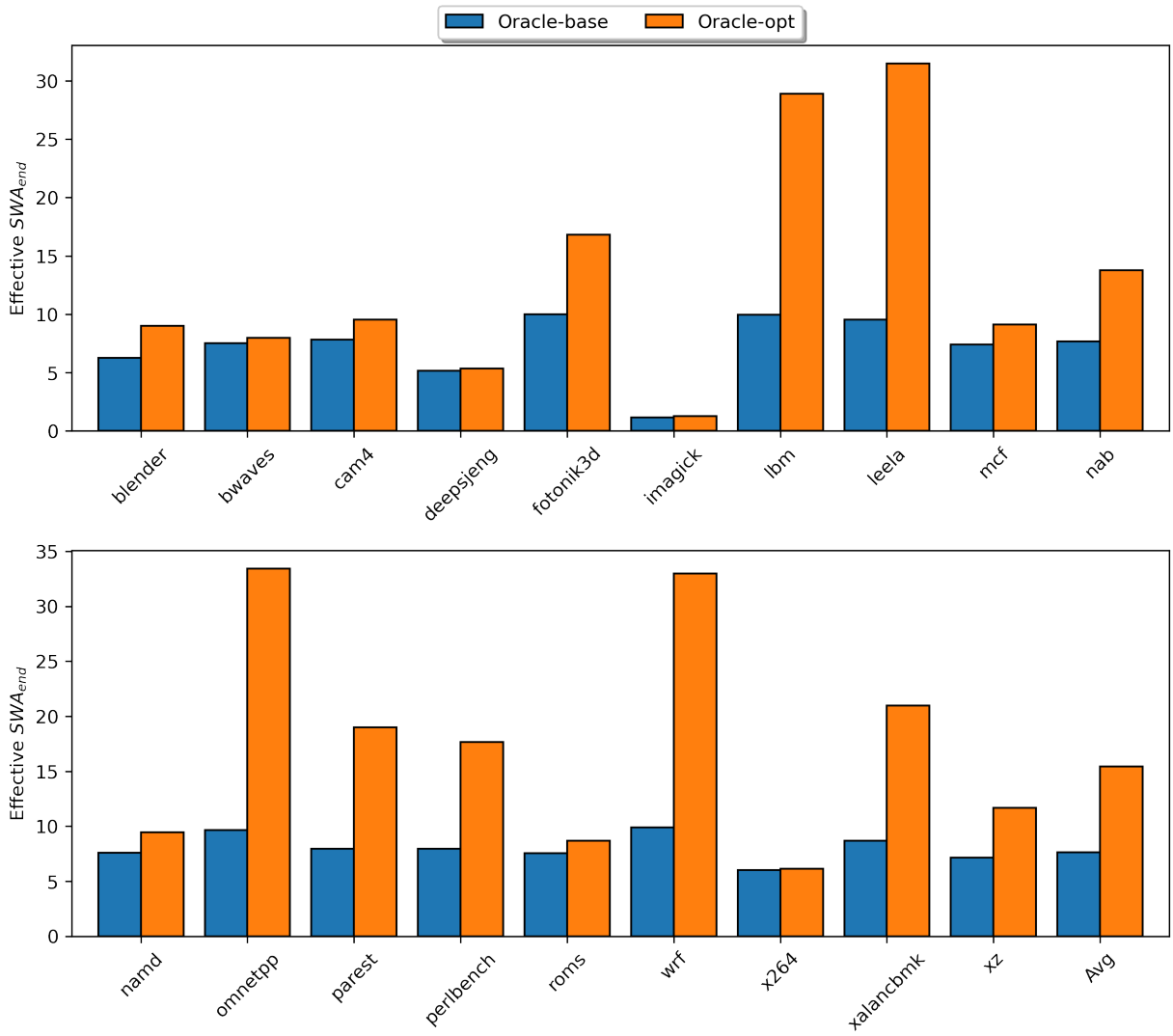

Figure 8.2: Effective SWA_{end} for Oracle-base with 10 seconds retention time and Oracle-opt with optimal retention for each benchmark.

The effective $SWAend$ improvement for the Oracle scheme with the best retention time is shown in Figure 8.2. The baseline Oracle scheme used up to this point in this thesis, with a 10-second retention time, is represented by the 'Oracle-base' plot. 'Oracle-opt' plots the endurance results when using retention times that maximize each benchmark's results. The average effective SWA_{end} for oracle-base and oracle-opt are 7.63 and 15.39, respectively. Therefore, there is still room to improve the endurance results by 201% if the best retention time for each benchmark is used.

8.2 SRTP with optimal retention

The previous section illustrated the best retention time for each program as well as potential gains in endurance results for the oracular scheme. Now, we evaluate how well our soft write scheme would function if each benchmark were to be used with the ideal retention time. 'SRTP-opt' refers to the SRTP scheme that is aware of the ideal retention time for each benchmark. We simulate SRTP-opt and compare the results to the baseline scheme, called 'SRTP-base' in this section, which was used in the earlier chapters. Figure 8.3 plots the effective SWA_{end} for these two versions of SRTP.

When using the best case retention time for each benchmark, SRTP-opt is able to raise effective $SWAend$ from 6.31 for SRTP-base to 10.48, an increase of 166%. As we saw in Chapter 7, effective SWA_{end} is directly related to the lifetime of the memory system. Therefore, for SRTP-opt, the lifetime would also increase by about 166 percent. This is a significant advancement, but it necessitates profiling the overwrite time for a program ahead of time. The overwrite time profile would be sensitive to numerous run-time factors like program inputs, ma-

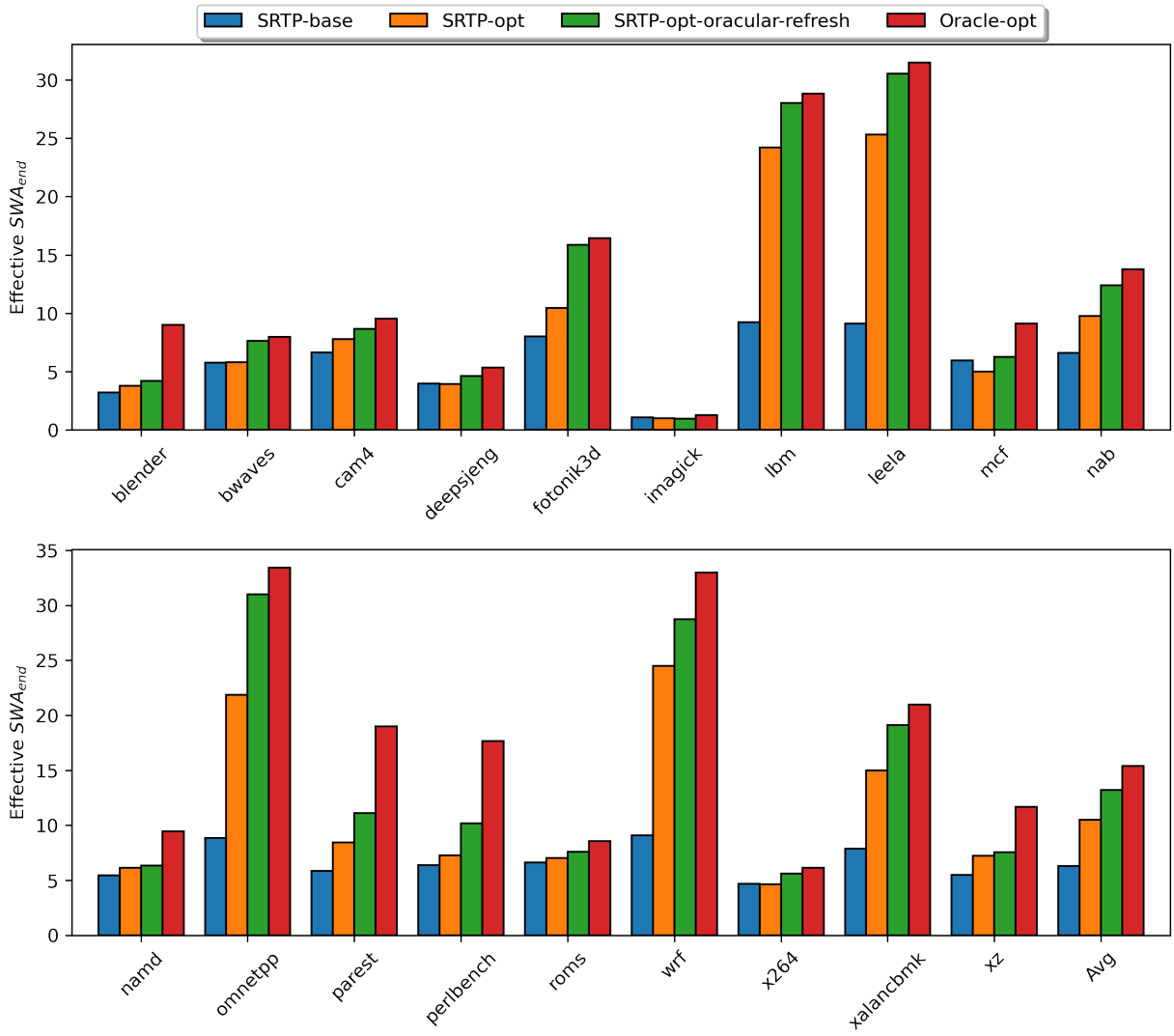

Figure 8.3: Effective SWA_{end} for SRTP-base with 10 seconds retention time and SRTP-opt with optimal retention for each benchmark.

chine configuration, etc. Hence, profiling a program just once and using the same information in all circumstances would be ineffective. The ideal retention period would need to be determined at run-time by the soft write policy to accomplish these results. Here, we demonstrate the advantages of doing so while leaving implementation to future research.

While SRTP-opt is able to get higher endurance savings compared to the baseline SRTP, the gap between SRTP and the Oracle scheme grows wider for optimal retention compared to

baseline retention. As discussed in Section 6.1, SRTP-base is within 18.5% of Oracle-base. Results for SRTP-opt, on the other hand, are 31.9% less than Oracle-opt. Simply looking at the effective SWA_{end} numbers is misleading, as Oracle's performance improvement depends on two factors. The accuracy of soft write decision-making as well as the number of refreshes incurred for those soft write decisions.

As shown in Table 8.1, the optimal retention for benchmarks is usually much lower than the baseline retention of ten seconds. While doing so lessens the endurance cost associated with a single soft write, it also increases the number of refreshes needed for a given overwrite time. Hence, we observe a much larger number of refresh operations for optimal retention time. Oracle has a highly unfair advantage when it comes to reducing the number of refreshes. It has a timer associated with each memory line that not only does the bare minimal refresh at a line granularity, but also eliminates refresh all together when the next write to a line arrives before retention time expires. With the memory overhead of one timestamp per line, this is not feasible for any practical soft write technique. As a result, practical soft write techniques have to resort to using low overhead techniques like tracking and refreshing the whole physical pages.

We combine SRTP's soft write decisions and Oracle's refreshes to estimate the performance loss brought on by an excessive amount of refreshing in a practical soft write technique. Results for this scheme are represented graphically as 'SRTP-opt-oracular-refresh' in Figure 8.3. This hybrid version of SRTP-opt achieves an effective SWA_{end} of 13.2, closing the gap between SRTP-opt and Oracle-opt from 31.9% to 14.2%. The remaining difference in the results can be attributed to the accuracy and speed of the overwrite time detection by SRTP. Although this is not a realistic implementation of SRTP, the results highlight crucial differences in the performance of SRTP and Oracle for optimum retention time. With variations in refresh accounted for less

than half of the performance difference can be attributed to soft write decision-making. So SRTP is still competitive in identifying soft write opportunities, but the refresh starts to incur higher overhead at lower retention times.

Chapter 9: Related Work

Multi-level cell PCM exhibits a tradeoff between write latency and retention. Writes can be made faster by sacrificing some retention time. Previous work leveraged this to improve performance at the expense of retention and incurring the requirement to refresh. Such techniques include compiler based approaches [106–108], NVM as a cache for SSDs [109], persistent programming frameworks [110], and architectural methodologies [8, 111]. Out of these, RRM is the best-performing technique that (like our work) targets systems with CPUs and non-volatile main memory [8]. We adapted RRM to control soft writes for single-level cell ReRAM main memory to optimize energy and endurance. Our findings demonstrate that SRTP significantly outperforms RRM for ReRAM cell properties. Another possible direction might be to optimize SRTP for MLC PCM tradeoffs and then compare it against RRM.

Mellow Writes [84, 112] improve ReRAM endurance by spreading the same (or slightly greater) write energy over a longer write cycle during idle periods in the memory system, thus minimally impacting contention and performance. Because a similar write energy is used, Mellow Writes do not impact the CF size, so retention remains roughly the same. While Mellow Writes only helps endurance, SRTP improves both endurance and energy since it employs "true" soft writes that dissipate less energy (though at the expense of reduced retention time and the need to refresh). Notice, SRTP and Mellow Writes are orthogonal. SRTP could make its soft or

hard writes mellow, and further improve lifetime.

All of the above works try to reduce the impact of *individual writes* to non-volatile memory. Another direction of work tries to reduce the *number of writes*. This includes techniques that read the data first to only write the modified bits [113–117], encode the data [118–125], compress the data [126,127], or utilize asymmetries in set and reset operations [113,114,128,129]. All of these techniques are orthogonal to our work and can be applied on top of soft writes. In fact, there are some co-optimization opportunities like refresh aware encoding where data is encoded keeping not just soft writes but future refreshes in mind.

Another write reduction technique is LLC management policies to reduce the number of writebacks [130–134]. These policies can only target writebacks with short overwrite times, leaving a lot of soft write opportunities. Since SRTP is agnostic to LLC management, it can adjust to these techniques as long as the RTD gets the needed information. Memory cocktail therapy [135] looks at different techniques for optimizing aspects of non-volatile memory and uses a learning based framework to adaptively choose the best techniques for an application. SRTP can be added to its collection of techniques to take soft writes into account.

Chapter 10: Conclusion

Conventional memory technologies are experiencing scaling issues while the requirements of the modern workloads grow at never seen before rates. Emerging non-volatile memory technologies, such as ReRAM, offer a lucrative alternative because of their high density, scalability, 3-D integration, and non-volatility. These technologies do, however, have some drawbacks of their own. The main obstacle preventing these memories from becoming more widely used are problems with their write operation. In addition to having limited write endurance, their cells also require much higher write energy. These limitations cause reliability issues where the memory cells stop working after a certain number of writes and excessive energy consumption in memory for programs that write frequently. Addressing these challenges is the subject of this thesis.

We note that the non-volatility or persistence is not without cost. The writes require a significant amount of energy to make the data last for 10 years in memory, which also decreases cell endurance. In fact, the write energy and endurance are inversely related. Reducing the energy pumped into the ReRAM cell can increase its endurance. However, doing so makes the final state of the device less stable, which leads to lower retention time. So, non-volatility is not absolute and ReRAM exhibits multiple levels of volatility depending upon the intensity of the write. This device characteristic allows trading off retention for energy and endurance, depending upon the application.

The average lifespan of a program is much less than 10 years. Furthermore, a sizable portion of the modified memory locations is overwritten in a matter of seconds. Therefore, the 10-year retention time is not necessary for a significant portion of writes to the main memory. In this thesis, we exploit the retention versus energy and endurance tradeoff that exists in ReRAM. Our work identifies the relationship between store overwrite time, retention time, and soft write advantage that governs the profitability of using soft writes for each LLC writeback. We develop an architectural technique, called SRTP, that learns the dominant overwrite time on a per-store PC basis, and uses the store locality information to predict a soft versus hard write decision at every dynamic store instruction. We conduct a simulation-based evaluation of SRTP, and show that it improves endurance by 2.6x to 4.1x compared to a state-of-the-art soft write technique, RRM. SRTP also comes within 18.5% of the Oracle's endurance. At the same time, SRTP reduces the main-memory energy consumption by 2.12x. This is 1.67 to 2.05 times better than the energy savings by RRM. We integrate SRTP with an existing wear-leveling technique and demonstrate that SRTP can improve actual lifetime by 6.4 times.

Additionally, we showed that traditional wear-leveling methods were ineffective when there were writers with various levels of endurance wear. This thesis's novel wear-leveling policy guiding metric aids in recovering the loss in the efficacy of the wear-leveling techniques by 5.7% to 19.4%.

Bibliography

[1] Guangyu Sun, Jishen Zhao, Matt Poremba, Cong Xu, and Yuan Xie. Memory that never forgets: emerging nonvolatile memory and the implication for architecture design. *National Science Review*, 5(4):577–592, 2018.

[2] Microsoft. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the Worldś Largest and Most Powerful Generative Language Model. `https://www.microsoft.com/en-us/research/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/`, 2021.

[3] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. Architecture design with stt-ram: Opportunities and challenges. In *2016 21st Asia and South Pacific design automation conference (ASP-DAC)*, pages 109–114. IEEE, 2016.

[4] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[5] Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Understanding the trade-offs in multi-level cell reram memory design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2013.

[6] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 14–23. IEEE, 2009.

[7] Qingyue Liu and Peter Varman. Ouroboros wear leveling for nvram using hierarchical block migration. *ACM Transactions on Storage (TOS)*, 13(4):1–31, 2017.

[8] Mingzhe Zhang, Lunkai Zhang, Lei Jiang, Zhiyong Liu, and Frederic T. Chong. Balancing Performance and Lifetime of MLC PCM by Using a Region Retention Monitor. In *International Symposium on High Performance Computer Architecture*, 2017.

[9] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[10] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[11] IEEE. International Roadmap for Devices and Systems (IRDS™) 2021 Edition. "`https://irds.ieee.org/editions/2021`", 2017.

[12] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *ACM SIGARCH Computer Architecture News*, 40(3):1–12, 2012.

[13] Crossbar. Personal communication. 2022.

[14] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–488. IEEE, 2015.

[15] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[16] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143–143, 2010.

[17] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH computer architecture news*, 37(3):14–23, 2009.

[18] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112. IEEE, 2009.

[19] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the International Symposium on Computer ARchitecture*, Austin, TX, 2009.

[20] Dongliang Xue, Chao Li, Linpeng Huang, Chentao Wu, and Tianyou Li. Adaptive memory fusion: Towards transparent, agile integration of persistent memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 324–335. IEEE, 2018.

[21] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayanan. Nonvolatile processor architectures: Efficient, reliable progress with unstable power. *IEEE Micro*, 36(3):72–83, 2016.

[22] Mohammad Arjomand, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. Boosting access parallelism to pcm-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 695–706. IEEE, 2016.

[23] Intel. Intel Optane Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2017.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[25] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. *ACM SIGARCH computer architecture news*, 38(3):383–394, 2010.

[26] Andre Seznec. A phase change memory as a secure main memory. *IEEE Computer Architecture Letters*, 9(1):5–8, 2010.

[27] Neha Agarawal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[28] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 2016 EuroSys Conference*, 2016.

[29] Takahiro Hirofuchi and Ryousei Takano. RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems. In *Proceedings of the ACM Symposium on Cloud Computing*, Santa Clara, CA, October 2016.

[30] Dmitrii Ustiugov, Mark Suterhland, Alexandros Daglis, Edouard Bugnion, Dionisios Pnevmatikatos, Javier Picorel, and Babak Falsafi. Design Guidelines for High-Performance SCM Hierarchies. In *Proceedings of the 4th International Symposium on Memory Systems*, National Harbor, D.C., 2018.

[31] Tae Jun Ham, Bharath K Chelepalli, Neng Xue, and Benjamin C Lee. Disintegrated control for energy-efficient and heterogeneous memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 424–435. IEEE, 2013.

[32] Majed Valad Beigi, Bahareh Pourshirazi, Gokhan Memik, and Zhichun Zhu. Deepswapper: A deep learning based page swap management scheme for hybrid memory systems. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 353–354, 2020.

[33] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. Adaptive page migration policy with huge pages in tiered memory systems. *IEEE Transactions on Computers*, 71(1):53–68, 2020.

[34] Hao Wang, Jie Zhang, Sharmila Shridhar, Gieseo Park, Myoungsoo Jung, and Nam Sung Kim. Duang: Fast and lightweight page migration in asymmetric memory systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–493. IEEE, 2016.

[35] Crossbar. Crossbar ReRAM Overview. 2020.

[36] Corey Lammie, Mostafa Rahimi Azghadi, and Daniele Ielmini. Empirical Metal-Oxide RRAM Device Endurance and Retention Model for Deep Learning Simulations. *Semiconductor Science and Technology*, 36, 2021.

[37] D. Ielmini, Federico Nardi, Carlo Cagli, and Andrea L. Lacaita. Trade-off Between Data Retention and Reset in NIO RRAMs. In *Proceedings of the IEEE International Reliability Physics Symposium*, May 2010.

[38] Shimeng Yu, Yang Yin Chen, Ximeng Guan, and H.-S. Philip Wong. A Monte Carlo Study of the Low Resistance State Retention of HfOx Based Resistive Switching Memory. *Applied Physics Letters*, 2012.

[39] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the International Symposium on Microarcitecture*, Porto Alegre, Brazil, December 2011.

[40] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture*, Saint Malo, France, June 2010.

[41] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, September 2012.

[42] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–257, 2003.

[43] Samira Mirbagher-Ajorpaz, Elba Garza, Gilles Pokam, and Daniel A Jiménez. Chirp: Control-flow history reuse prediction. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 131–145. IEEE, 2020.

[44] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*, pages 245–250. IEEE, 2007.

[45] Wanli Liu and Donald Yeung. Enhancing ltp-driven cache management using reuse distance information. *J. Instr. Level Parallelism*, 11, 2009.

[46] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572. IEEE, 2022.

[47] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the International Symposium on Microarchitecture*, 2006.

[48] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, Seattle, WA, June 2007.

[49] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2002.

[50] Qingyue Liu and Peter Varman. Ouroboros Wear-Levling: A Two-Level Hierarchical Wear-Leveling Model for NVRAM. In *Proceedings of the International Conference on Massive Storage Systems and Technology*, Santa Clara, CA, May 2017.

[51] Prashant J Nair, Dae-Hyun Kim, and Moinuddin K Qureshi. Archshield: Architectural framework for assisting dram scaling by tolerating high error rates. *ACM SIGARCH Computer Architecture News*, 41(3):72–83, 2013.

[52] Namhyung Kim and Kiyoung Choi. Exploration of trade-offs in the design of volatile stt–ram cache. *Journal of Systems Architecture*, 71:23–31, 2016.

[53] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, 28(2):223–262, 2010.

[54] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. Resistive random access memory (rram): an overview of materials, switching mechanism, performance, multilevel cell (mlc) storage, modeling, and applications. *Nanoscale research letters*, 15(1):1–26, 2020.

[55] Ilia Valov, Rainer Waser, John R Jameson, and Michael N Kozicki. Electrochemical metallization memories—fundamentals, applications, prospects. *Nanotechnology*, 22(25):254003, 2011.

[56] Federico Nardi, Daniele Ielmini, Carlo Cagli, S Spiga, M Fanciulli, Ludovic Goux, and DJ Wouters. Control of filament size and reduction of reset current below 10 μa in nio resistance switching memories. *Solid-State Electronics*, 58(1):42–47, 2011.

[57] C Nail, G Molas, P Blaise, G Piccolboni, B Sklenard, C Cagli, M Bernard, A Roule, M Azzaz, E Vianello, et al. Understanding rram endurance, retention and window margin trade-off using experimental results and simulations. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 4–5. IEEE, 2016.

[58] Meenatchi Jagasivamani. *Resistive RAM Based Main-Memory Systems: Understanding the Opportunities, Limitations, and Tradeoffs*. PhD thesis, University of Maryland, College Park, 2020.

[59] Meiran Zhao, Huaqiang Wu, Bin Gao, Xiaoyu Sun, Yuyi Liu, Peng Yao, Yue Xi, Xinyi Li, Qingtian Zhang, Kanwen Wang, et al. Characterizing endurance degradation of incremental switching in analog rram for neuromorphic systems. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 20–2. IEEE, 2018.

[60] Sergiu Clima, YY Chen, Andrea Fantini, Ludovic Goux, Robin Degraeve, Bogdan Govoreanu, Geoffrey Pourtois, and Malgorzata Jurczak. Intrinsic tailing of resistive states distributions in amorphous hfo x and tao x based resistive random access memories. *IEEE Electron Device Letters*, 36(8):769–771, 2015.

[61] Haiyu Mao, Xian Zhang, Guangyu Sun, and Jiwu Shu. Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1623–1626. IEEE, 2017.

[62] Sunwoong Kim, Hyunmin Jung, Woojae Shin, Hyokeun Lee, and Hyuk-Jae Lee. Had-twl: Hot address detection-based wear leveling for phase-change memory systems with low latency. *IEEE Computer Architecture Letters*, 18(2):107–110, 2019.

[63] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 914–919, 2010.

[64] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: Improving {NAND} flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 47–59, 2014.

[65] Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. An endurance-enhanced flash translation layer via reuse for nand flash memory storage systems. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[66] Hongliang Yu and Yuyang Du. Increasing endurance and security of phase-change memory with multi-way wear-leveling. *IEEE Transactions on Computers*, 63(5):1157–1168, 2014.

[67] Duo Liu, Tianzheng Wang, Yi Wang, Zili Shao, Qingfeng Zhuge, and Edwin Sha. Curling-pcm: Application-specific wear leveling for phase change memory based embedded systems. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 279–284, 2013.

[68] Jianming Huang, Yu Hua, Pengfei Zuo, Wen Zhou, and Fangting Huang. An efficient wear-level architecture using self-adaptive wear leveling. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Guan Wang, Fei Peng, Lei Ju, Lei Zhang, and Zhiping Jia. Double circulation wear leveling for pcm-based embedded systems. In *Advanced Computer Architecture*, pages 190–200. Springer, 2014.

[70] SPEC CPU2017 Benchmarks. . 2017.

[71] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178. IEEE, 2006.

[72] David A Patterson Hennessy. Computer architecture: A quantitative approach by john l. *Hennessy, David A. Patterson*, 2017.

[73] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[74] André Seznec. A case for two-way skewed-associative caches. *ACM SIGARCH computer architecture news*, 21(2):169–178, 1993.

[75] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[76] Matthew Poremba, Tao Zhang, and Yuan Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, 2015.

[77] Candace Walden, Devesh Singh, Meenatchi Jagasivamani, Shang Li, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. Monolithically integrating non-volatile main memory over the last-level cache. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–26, 2021.

[78] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Design of cross-point metal-oxide reram emphasizing reliability and cost. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 17–23. IEEE, 2013.

[79] Yang Zhang, Dan Feng, Wei Tong, Jingning Liu, Chengning Wang, and Jie Xu. Tiered-reram: A low latency and energy efficient tlc crossbar reram architecture. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 92–102. IEEE, 2019.

[80] Ying-Chen Chen. *Selector-less resistive random access memory (RRAM) with intrinsic nonlinearity for crossbar array applications*. PhD thesis, 2019.

[81] Albert Lee, Chieh-Pu Lo, Chien-Chen Lin, Wei-Hao Chen, Kuo-Hsiang Hsu, Zhibo Wang, Fang Su, Zhe Yuan, Qi Wei, Ya-Chin King, et al. A reram-based nonvolatile flip-flop with self-write-termination scheme for frequent-off fast-wake-up nonvolatile processors. *IEEE Journal of Solid-State Circuits*, 52(8):2194–2207, 2017.

[82] Yukio Hayakawa, Atsushi Himeno, Ryutaro Yasuhara, W Boullart, E Vecchio, T Vandeweyer, T Witters, D Crotti, M Jurczak, S Fujii, et al. Highly reliable tao x reram with centralized filament for 28-nm embedded application. In *2015 Symposium on VLSI Technology (VLSI Technology)*, pages T14–T15. IEEE, 2015.

[83] Alessandro Grossi, Elisa Vianello, Mohamed M Sabry, Marios Barlas, Laurent Grenouillet, Jean Coignus, Edith Beigne, Tony Wu, Binh Q Le, Mary K Wootters, et al. Resistive ram endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Transactions on Electron Devices*, 66(3):1281–1288, 2019.

[84] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 519–531. IEEE, 2016.

[85] Zixuan Chen, Huaqiang Wu, Bin Gao, Dong Wu, Ning Deng, He Qian, Zhichao Lu, Brent Haukness, M Kellam, and Gary Bronner. Performance improvements by sl-current limiter and novel programming methods on 16mb rram chip. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017.

[86] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14. IEEE, 2007.

[87] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *ICCAD: International Conference on Computer-Aided Design*, pages 694–701, 2011.

[88] Norman P Jouppi, Andrew B Kahng, Naveen Muralimanohar, and Vaishnav Srinivas. Cacti-io: Cacti with off-chip power-area-timing models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1254–1267, 2014.

[89] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, pages 90–95, 1998.

[90] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[91] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158. IEEE, 2018.

[92] Pawel Gepner, David L Fraser, and Victor Gamayunov. Evaluation of the 3rd generation intel core processor focusing on hpc applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[93] Yiping Zhang, Canyan Zhu, Lijun Zhang, and Ziou Wang. A write-verification method for non-volatile memory. In *2019 International Conference on IC Design and Technology (ICICDT)*, pages 1–3, 2019.

[94] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 67–76, 2000.

[95] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying multicore processor scaling via reuse distance analysis. *ACM SIGARCH Computer Architecture News*, 41(3):499–510, 2013.

[96] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 367–375, 2012.

[97] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, 2011.

[98] David Lo and Christos Kozyrakis. Dynamic management of turbomode in modern multicore chips. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–613, 2014.

[99] Jiacheng Zhao, Xiaobing Feng, Huimin Cui, Youliang Yan, Jingling Xue, and Wensen Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212, 2013.

[100] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-core mapping policies to reduce memory system interference in multicore systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 107–118, 2013.

[101] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1443–1456, 2016.

[102] Wei Debao, Qiao Liyan, Zhang Peng, and Peng Xiyuan. Bpm: A bad page management strategy for the lifetime extension of flash memory. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 57199, page V009T07A010. American Society of Mechanical Engineers, 2015.

[103] Duwon Hong, Myungsuk Kim, Geonhee Cho, Dusol Lee, and Jihong Kim. {GuardedErase}: Extending {SSD} lifetimes by protecting weak wordlines. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 133–146, 2022.

[104] Micron. TN-29-59: Bad block management. `https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2959_bbm_in_nand_flash.pdf`, 2011.

[105] Mahesh Sreekandath and Saugata Das Purkayastha. Asynchronous bad block management in nand flash memory, November 1 2016. US Patent 9,483,395.

[106] Qingan Li, Lei Jiang, Youtao Zhang, Yanxiang He, and Chun Jason Xue. Compiler directed write-mode selection for high performance low power volatile pcm. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pages 101–110, 2013.

[107] Keni Qiu, Qingan Li, and Chun Jason Xue. Write mode aware loop tiling for high performance low power volatile pcm. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[108] Chen Pan, Mimi Xie, Jingtong Hu, Yiran Chen, and Chengmo Yang. 3m-pcm: Exploiting multiple write modes mlc phase change main memory in embedded systems. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2014.

[109] Dongwoo Kang, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H Noh, and Onur Mutlu. Amnesic cache management for non-volatile memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2015.

[110] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: Unified working memory and persistent store architecture. *ACM SIGARCH Computer Architecture News*, 42(1):455–470, 2014.

[111] Mingzhe Zhang, Lunkai Zhang, Lei Jiang, Frederic T Chong, and Zhiyong Liu. Quick-and-dirty: An architecture for high-performance temporary short writes in mlc pcm. *IEEE Transactions on Computers*, 68(9):1365–1375, 2019.

[112] Mohammad Reza Jokar, Lunkai Zhang, and Frederic T Chong. Cooperative nv-numa: prolonging non-volatile memory lifetime through bandwidth sharing. In *Proceedings of the International Symposium on Memory Systems*, pages 67–78, 2018.

[113] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison

write scheme. In *2007 IEEE International Symposium on Circuits and Systems*, pages 3014–3017. IEEE, 2007.

[114] 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, 2009.

[115] Jie Chen, Ron C Chiang, H Howie Huang, and Guru Venkataramani. Energy-aware writes to non-volatile main memory. *ACM SIGOPS Operating Systems Review*, 45(3):48–52, 2012.

[116] 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, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 290–295. IEEE, 2015.

[117] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H Loh, and Doug Burger. Preventing pcm banks from seizing too much power. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 186–195. IEEE, 2011.

[118] Adam N Jacobvitz, Robert Calderbank, and Daniel J Sorin. Coset coding to extend the lifetime of memory. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 222–233. IEEE, 2013.

[119] Kazuteru Namba and Fabrizio Lombardi. A coding scheme for write time improvement of phase change memory (pcm) systems. *IEEE Transactions on Multi-Scale Computing Systems*, 2(4):291–296, 2016.

[120] Dan Feng, Jie Xu, Yu Hua, Wei Tong, Jingning Liu, Chunyan Li, and Yiran Chen. A low-overhead encoding scheme to extend the lifetime of nonvolatile memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2516–2529, 2019.

[121] Huizhang Luo, Liang Shi, Qiao Li, Chun Jason Xue, and Edwin H-M Sha. Energy, latency, and lifetime improvements in mlc nvm with enhanced wom code. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 554–559. IEEE, 2018.

[122] Jue Wang, Xiangyu Dong, Guangyu Sun, Dimin Niu, and Yuan Xie. Energy-efficient multi-level cell phase-change memory system with data encoding. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pages 175–182. IEEE, 2011.

[123] Jie Xu, Dan Feng, Yu Hua, Wei Tong, Jingning Liu, Chunyan Li, and Wen Zhou. Improving performance of tlc rram with compression-ratio-aware data encoding. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 573–580. IEEE, 2017.

[124] Poovaiah M Palangappa and Kartik Mohanram. Rapid: read acceleration for improved performance and endurance in mlc/tlc nvms. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2018.

[125] Yang Zhang, Dan Feng, Wei Tong, Jingning Liu, Chengning Wang, and Jie Xu. Tiered-reram: A low latency and energy efficient tlc crossbar reram architecture. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 92–102, 2019.

[126] Poovaiah M. Palangappa and Kartik Mohanram. Compex: Compression-expansion coding for energy, latency, and lifetime improvements in mlc/tlc nvm. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 90–101, 2016.

[127] Jie Xu, Dan Feng, Yu Hua, Wei Tong, Jingning Liu, and Chunyan Li. Extending the lifetime of nvms with compression. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1604–1609. IEEE, 2018.

[128] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting pcm asymmetries. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 282–293. IEEE, 2013.

[129] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. Improving phase change memory performance with data content aware access. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 30–47, 2020.

[130] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A Jiménez. Wade: Writeback-aware dynamic cache management for nvm-based main memory system. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):1–21, 2013.

[131] Bahareh Pourshirazi, Majed Valad Beigi, Zhichun Zhu, and Gokhan Memik. Wall: A writeback-aware llc management for pcm-based main memory systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 449–454. IEEE, 2018.

[132] Deshan Zhang, Lei Ju, Mengying Zhao, Xiang Gao, and Zhiping Jia. Write-back aware shared last-level cache management for hybrid main memory. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[133] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[134] Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Reducing writebacks through in-cache displacement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(2):1–21, 2019.

[135] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–244, 2017.