INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

DOCTORAL THESIS

Content and Coherence Based Strategies for Optimizing Refreshes in Volatile Last Level Caches

Sheel Sindhu Manohar
Roll No: 156101027

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy of the Indian Institute of Technology Guwahati.

Supervisor: Prof.Hemangee K. Kapoor

Department of Computer Science Engineering
Indian Institute of Technology Guwahati
Guwahati (Assam), India, Pin- 781039

December 2021

I would like to dedicate this thesis to my loving parents . . .

Declaration

Sheel Sindhu Manohar
Roll No: 156101027
Department of Computer Science Engineering
Indian Institute of Technology Guwahati
Guwahati (Assam), India, Pin- 781039
Email: sheel.manohar@iitg.ac.in

I certify that

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisor(s).
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, grpahs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their and giving their details in the references. Elaborate sentences used verbatim from published work have been clearly identified and quoted.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.

- I am fully aware that my thesis supervisor(s) are not in position to check for any possible instance of plagiarism within this submitted work.

Sheel Sindhu Manohar
December 2021

Certificate

Prof.Hemangee K. Kapoor
Department of Computer Science Engineering
Indian Institute of Technology Guwahati
Guwahati (Assam), India, Pin- 781039
Email: hemangee@iitg.ac.in

This is to certify that the work contained in the thesis entitled **‘Content and Coherence Based Strategies for Optimizing Refreshes in Volatile Last Level Caches’** by **Sheel Sindhu Manohar** (Roll No. 156101027), a student of Department of Computer Science Engineering, Indian Institute of Technology Guwahati, for the award of degree of Doctor of Philosophy, has been carried out under my supervision.

The present thesis or any part thereof has not been submitted elsewhere for award of any other degree, diploma or associateship.

Prof.Hemangee K. Kapoor
December 2021

Acknowledgements

Today, where I stand is possible because of a great deal of support and assistance. Firstly, I would like to present my sincere gratitude towards my supervisor Prof. Hemangee K. Kapoor, for the continuous support and guidance throughout my Ph.D. tenure. I have received invaluable advice, tutelage, and motivation from her. She taught me how to think critically and the importance of 'how' and 'why' in research fields. She made me a more confident and growing person over these years. I admire him for her incredible attitude towards computer science. She always showed confidence in my work and kept patience for results. I never felt pressured to produce results under her guidance. I found myself being more creative after joining her group. I liked the way she celebrated every achievement with us and made the lab environment happen all the time. One of the best things I liked after joining her lab is that she prioritizes her time with her scholars over any other important stuff she deals with. I could not have imagined having a better advisor and mentor other than ma'am for my Ph.D. Her insightful feedback pushed me to work at a higher level and honed my thinking and observation skills.

I am thankful to my Doctoral Committee Members: Prof. Diganta Goswami, Dr. Aryabartta Sahu, Dr. Arnab Sarkar, and Dr. Chandan Karfa, for their constructive suggestions and comments for my thesis work. In addition, my sincere thanks to Prof. Jatindra Deka, the Head of the Department of Computer Science Engineering, and other department members for their constant support throughout my research tenure.

I wish to thank my lab mates Dr. Shounak Chakraborty, Dr. Sukarn Agarwal, Dr. Khushboo Rani, Palash Das, and Arijit Nath, for their help in carrying out lab work. They helped me out in developing the ideas. They always have a special place for the bond we forged and the good times we enjoyed together. They were indeed my stress buster. I especially thank my seniors Dr. Shirshendu Das, Dr. Shounak Chakraborty, and Dr. Sukarn Agarwal, for their initial support in my research in the world of computer architecture. They also created a great environment and provided me with good company and support during the maturing stage of my research work.

I would also like to thank Dr. Sonal Yadav, Manik Bhosale, Rodney Stephen Rodrigues, Neelkamal, T. Susma Devi, Imlijungla Longchar and Ashwathi for helping and coordinating during the TA duties.

I wish to thank my batchmates Debendra, Subrata, Shreshta, Bikramjit, Kangkan, Anasua, Abhijit, Arunangshu, Sisir, Dr. Pranav, Manjari, Shikha, Dr. Anirban, Dr. Shubhlakshmi, and many uncountable names for creating the great environment and providing me good company and support. They made IIT Guwahati another home for me. I have made several good memories with them.

I wish to extend my thanks to Prof. Shekhar Verma, who motivated and supported my growth. I wish to thank Dr. Neha Jain, who introduced me to this programming world. I would like to thank Dr. Reena for keeping patience and faith in me at my critical phase.

Thanks to the technical and staff members of the department, Mr. Raktajit Pathak, Mr. Bhriguraj Borah, Mr. Nanu Alan Kachari, Mrs. Gauri Deori, Mr. Monojit Bhattacharjee, and all other department members for providing help throughout. I would like to thank the student affairs section for helping me by providing me with on-campus facilities. I am expressing appreciation to security guards, Janitors, hostel mess and workers, and canteen staff for hospitality in the IITG campus.

This thesis is dedicated to my family members who have kept faith and belief in me. A special feeling of gratitude to my father, Late. Mridul Manohar and Ramola Devi, whose hard work and motivation pay me off. My sister Madhulika Manohar for continuous encouragement and motivation throughout. Their patience and trust made me more confident, hardworking, and energetic throughout the tenure.

Finally, I would like to gratefully acknowledge Dr. Bhimrao Ramji Ambedkar for giving me the motivation to learn throughout my life. I would also like to thank the Ministry of Human Resource Development (MHRD) and Equal Opportunity Cum Special Reservation Cell, IIT Guwahati, for providing me with a fellowship to carry out my research work.

Thank you, everyone.

Sheel Sindhu Manohar

Abstract

With each process generation, Moore's law offers us an exponential growth in the transistor budget on the chip. Technically, these extra transistors were used to improve processor architecture speed by adding more complicated and simple pipelines and better arithmetic and floating-point units. The futher advances included multi-core systems which demanded larger on-chip caches to support the data demands. Larger last-level caches are deployed across the chip to meet the increasing need for higher cache capacity due to included CMPs in processing cores. LLCs play an important function in the cache hierarchy by giving necessary data to hungry CMPs. SRAMs are not scalable and require advancements in power, performance, and scalability. In order to deploy massive LLCs, researchers are focusing on the construction of caches using alternative technologies that have advantages over traditional SRAM. High scalability, lower leakage power, and higher capacity in the same area footprint as SRAM are among the benefits of these technologies. However, we must investigate the best of these alternatives because they are not without flaws.

In order to control the leakage generated by SRAM based caches, researchers have explored avenues in emerging eDRAM and STTRAM based designs. These new memories come with their own challenges, in that eDRAM requires refreshing the data at regular intervals while low retention STTRAM needs restores and refreshes. This thesis aims to reduce the number of refreshes required for such type of last level caches. Our goal is to address the reliability of data over the volatile caches by ensuring the retention of data blocks. Here, we deal with volatile memory technologies' challenges and make them suitable candidates for cache hierarchy. Similarly, in STTRAM, the asymmetrical read and write with the costly write operation. Thus, it can be addressed by the relaxed versions of STTRAM.

Towards this aim, the thesis makes the following contributions: (1) In the first contribution we note that eDRAM based caches need to be refreshed at regular intervals due to their volatile nature. Here we use content based information to decide which blocks can be avoided from refreshing. In particular, blocks has zero values need not be refreshed. (2) Second contribution uses Coherence based messages to identify the cache blocks which get loaded for getting written/updation by upper level caches. These exclusively hold block get stale eventually Copies of such private blocks in the LLC need not be refreshed as they will get updated in due course. (3) In our third contribution we deal with volatile STTRAM caches. At deep sub-micron technology these caches have read and write current values close to each other leading to read disturbance error (RDE). This makes each read destructive forcing us to restore the data after every read. We handle this by identifying read intensive blocks and moving them to a SRAM buffer. Future reads will be served from the buffer thus saving read related restores. (4) Our fourth contriution is deals with issues arising out from multi-retention STTRAM caches. STTRAM caches can be optimised to reduce their write energy by reducing their retention interval. This has disadvantage of blocks expiring at the end of retention interval. To avoid this, we use coherence based messages to identify the best partition to place a block. The idea behind this is that if the cache block is accessed before it expires we can avoid refreshing that block. All proposals have saved considerable refreshes, saved total energy as well as improved system performance.

Table of contents

List of figures

List of tables

CHAPTER 1

Introduction

Introduction

With each process generation, Moore's law [1] offers us an exponential growth in the transistor budget on the chip. Technically, these extra transistors were used to improve processor architecture speed by adding more complicated and simple pipelines and better arithmetic and floating-point units. All this setup created the environment for faster CPUs with enhanced performance. However, if the clock frequency increases at the same rate, managing the processor's heat dissipation will be difficult. When combined with the end of voltage scaling, the electronics industry is approaching a new phase in which transistors become a limited commodity for which a new technological generation cannot be expected to double. In order to stay up with Moore's law, researchers turned away from enhancing frequency and toward numerous cores on the chip. Single-core systems had attained the upper limit of maximum achievable performance, i.e., the most significant operational frequency, over the last decade (with the end of Dennard Scaling), owing to their higher power consumption.

This motivated architects and designers to develop processor chips with multiple cores that can function at lower voltages and frequencies than single-core counterparts while delivering the same performance using less power. Furthermore, the ever-increasing demand for processing capacity to execute modern demanding programs necessitates a significant degree of parallel processing, which single-core computers can no longer provide.

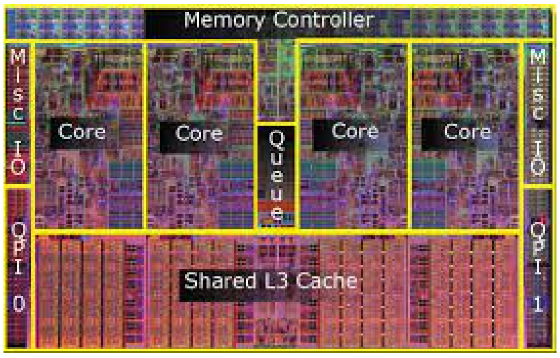


Fig. 1.1 Nehalem

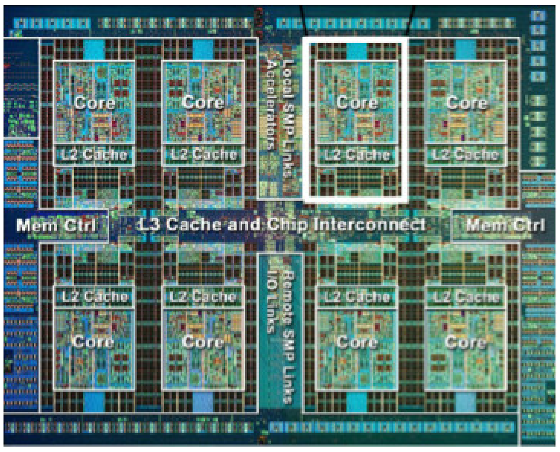

Core
L2 Cache
Local SMP Links Accelerators
Mem Ctrl
L3 Cache and Chip Interconnect
Mem Ctrl
Remote SMP Links I/O Links

Fig. 1.2 IBM POWER7 CPU with eDRAM

1.1 Chip Multi-Processors(CMP)

In CMP, multiple cores are combined on the same die. Workloads and applications will require more throughput and parallelism to keep up with the increased data demands with each new generation. Furthermore, it is impossible to simply increase the clock speed of CPUs since doing so would make water-cooled systems difficult to use. It is too expensive to develop larger processors every year with enormous number of transistors present on microprocessor chips. CMPs circumvent these issues by packing a processing die with numerous, relatively basic CPU cores rather than a single massive core. The multi-level on-chip caches are used to meet the data requirement. In order to achieve parallelism, this setup of cores and caches requires a strong communication system provided by the so-called Network-on-chip (NoC). Figure 1.1 and 1.2 [2] presents the design and the floor-plan outlines of two modern CMPs architectures: Nehlam and IMB POWER7+.

The exact size of a CMPs cores can vary from very simple pipelines to moderately complex superscalar processors, but once a core has been selected the CMPs performance can easily scale across silicon process generations simply by stamping down more copies of the hard-to-design, high-speed processor core in each successive chip generation.

Components in CMPs

Presently, the modern CMP are built by the following components:

- **CPU cores:** On chip computational elements. In terms of processing, these cores could be homogeneous or heterogeneous. We get enhanced performance by deploying more identical cores over the chip.
- **On-chip caches:** to store most recently used data related to present running application
- **Network-on-Chip:** A crucial element of chip multiprocessors (CMPs) is network-on-chip (NoC), which enables communication across several cores. NoCs for current processors, however, can make up to 30% of the chip's entire power budget. Furthermore, it is anticipated that NoC power will contribute more as chips continue to get compact. For chips to be capable of being able to keep up with the rising demand, it is necessary to cut NoC power consumption. Power-gating is a good strategy for minimizing the static power of idle resources in various kinds of circuits.

The performance is quantified in terms of Instructions per second(IPS), speed-up factor, Total execution time, Instruction per cycle(IPC) to a base system, etc. The time needed to execute an application and the parallelism achieved by the applications during execution are two factors that influence system performance. The application's parallelism is achieved by using multi-threading, which divides the application into numerous threads.

There are different type of Applications executes over CMPs. They can be classified into two categories based on the nature of the Instruction: Memory Bound and Compute Bound. The great majority of the instructions in compute-bound applications are for arithmetic and logical operations. Memory-bound programs have the majority of the instructions for memory operations, such as load and store. As a result, system performance has been directly influenced by memory access latency and core clock frequency in this type of application. Network-on-Chip is dependent on the performance of communication, whereas overall execution time of the system can be represented as:

$TotalExecTime = ComputationTime + MemoryCycles + NoCLatency$

1.2 Cache Memory

Cache memory is essential in bridging the gap between the slowest memory component and the fastest processing element. [3] Thus, performance advances throughout processor generations. It is the smallest and fastest memory near the computing element used over the chip, which provides the requested data to the running application on CMPs. It could be divided into multiple levels with larger capacities. However, owing to technological limits, additional improvements in cache capacity may become problematic in the future. In order to extract more performance out of the finite cache capacity and reduce long-latency memory accesses, future processors must rely on practical cache management algorithms and policies.

Each memory access always comes first to the cache memory as the main memory cannot supply data to the processing unit at the rate they want due to a memory wall problem. The performance gap between processing speed and the operating speed of a RAM creates a bottleneck. It takes more clock cycles to fetch the block from the main memory than it does from the cache. In other words, a cache memory provides data frequently required to the CPU at a higher rate. The two principles of the locality of references: temporal locality and spatial locality of the cache, ensure that once a block is stored in the cache, it will be utilized numerous times before being evicted. Thus, the higher the cache memory hit rate, the less time is spent accessing the main memory. So, cache memory improves the performance(i.e., CPI) by average memory access time.

1.3 Scope of Power Consumption

The multiple processing cores have increased the demand for on-chip caches leading to multi-level cache hierarchy including larger last level caches (LLCs). In order to serve huge amount of data to cores we use cache memory as a staircase. LLCs are the point of contact between processing cores and main memory. Particularly in data-intensive applications, where we can have lots of memory accesses due to working sets. The applications with large working sets, resulting in premature evictions of usable cache blocks that will be referenced in the near future.

Static Random Access Memory (SRAM) technology is known for quick access time and does not need refresh operations. It has often been used to build cache memory cells in CMP-based systems. Typically, six transistors are used to create an SRAM cell (6T cells). SRAM caches have several limitations, the most prominent being their high power consumption and a large percentage of total die area use (due to leakage energy, which is directly proportional to the number of transistors). Future generations of technology are predicted to exacerbate this issue by further decreasing transistor size.

Processing cores are often accounted for high dynamic power consumption among the many on-chip components in contemporary CMPs. On-chip caches built of SRAM/DRAM, on the other hand, are known for their high leakage energy. In addition, it has been seen in current CMPs that as the size of the SRAM/DRAM cache rises, the number of transistors increases, consuming a considerable wafer real estate area. This leads to high leakage power consumption, which significantly contributes to the chip's total power consumption. The total Power dissipated is broadly occurs into the following two categories. Static Power and Dynamic Power

1.3.1 Static Power

Static Power is the power dissipated by a transistor; it is not switching (i.e., when it is static, the transistor is not transitioning from 1 to 0 or from 0 to 1). This static power dissipation is mainly leakage, so it is also known as leakage power. We can have various types of leakage in the circuit: (1) Sub-Threshold leakage, (2) Gate Leakage, and (3)Junction Leakage. The Sub-threshold leakage, as the name suggests something happened below the threshold. The sub-threshold leakage between the two units of CMOS is proportional to the chip temperature and supply voltage. The covalent connection between the atoms in the semiconductor material is broken as the chip temperature rises, releasing electrons that flow in the reverse bias and create the current known as sub-threshold leakage current. Sub-threshold leakage power is the amount of power taken as a result of a sub-threshold current. Leakage, which happens through the gate, is observed when gate oxide is very thin, and it allows current to pass through it. This leads to Gate Leakage. We have many P-N junctions on the CMOS transistors; these junctions in a transistor form diodes. These diodes are reverse biased. However, reversed biased diodes still conduct a small amount of current, which causes junction leakage current.

$$P_{stat} = K_1 V_{DD} T^2 e^{(\lambda V_{DD}+\beta)/T} + K_2 e^{(\gamma V_{DD}+\delta)} \quad (1.1)$$

In the Eq. (1.1), P_{stat} represents the static power consumption due to sub-threshold leakage by a CMOS circuit. T denotes the current temperature and V_{DD} implies the supply voltage. K_1 , K_2 , λ, β, γ, and δ are the empirical constant that represent the different circuits parameters.

1.3.2 Dynamic Power

Dynamic Power is the power that is dissipated when the circuit is active, and a circuit is active anytime voltage on input net changes due to some stimulus applied. It is to be Noted: change in the voltage of input net may or may not lead to change in the logic state of the output. However, in both cases, whether there is a change in output net or not, if there is a change in the input net voltage, dynamic power will be dissipated. Now dynamic Power is again classified into (1) Switching Power and (2) Internal Power. In case when change in input is also leading to the change in output, that's when switching power comes into the picture. The switching power is the power dissipated while charging and discharging the load capacitor at the cell's output. However, for computational elements dynamic power is nothing but the processing power which can be written as follows

$$P_{Dyn} = \alpha.CV^2.f \ (1.2)$$

In Eqn.(1.2), P_{Dyn} represents the dynamic power of the cores, and α, C, V and f denote activity factor, capacitance, supply voltage, and running frequency of the core, respectively, which indicates that core's power consumption has a direct dependency on its operating frequency.

Internal power is the power that is dissipated within the boundary of a cell. It can again be divided into two parts: Short Circuit power and power needed to charge and discharge the internal nodes of a transistor. Short Circuit power is the power dissipated when both PMOS and NMOS are on simultaneously during the input transition, makes a direct link between supply voltage and ground. This power consumption is often ignored due to its negligible value in CMPs.

1.4 Exploring On-Chip Last Level Caches

A last-level cache, like all effective caches, helps enhance speed considerably without relying on unusual or costly (or ill-defined) memory technology. It improves the performance of the generic DRAMs we have. Since the cache is on-chip and substantially quicker than off-chip DRAM and has a broader, faster link to the CPU cluster, it boosts both latency and bandwidth. What's the disadvantage? Of course, a cache takes up die space. The cache-control logic (managing tags, lookup tables, etc.) is insignificant, but the cache RAM takes up substantial space. A last-level cache, on the other hand, saves electricity since nothing costs more energy than reading or writing to/from external

DRAM, especially when you're hitting DRAM the way current workloads do. Every operation that stays on-chip saves a significant amount of power and time. There is always a situation of negotiation between benefits and overheads. Reduction of power consumption by Last Level Cache(LLC) can be a crucial factor for limiting the peak power consumption of the CMP chip.

1.5 Emerging Memory Technologies for Caches

The cache hierarchy employed in the computer design has been the traditional charge-based SRAM memory technology. It suffers from high static power consumption caused by a 6-Transitor design in terms of leakage current, which exists even when the circuit is not switching. In contrast, dynamic power consumption by capacitive load charging and discharging on transistor gates in the circuit [4]. The cache often takes up a considerable amount of the overall on-chip space (e.g., 60% in the Strong ARM [5]). Modern desktop CPUs, for example, have 8 MB last level caches, whereas server systems have 24 to 32 MB last level caches [6–8]. L2 cache uses roughly 24% of overall power usage in both Niagara and Niagara-2 CPUs. As a result, cache power consumption is becoming a significant portion of CPU power usage. The low density and high cost of SRAM, the size of an SRAM cache is usually limited to less than 10 megabytes, which is not large enough for of many memory-intensive applications [9].In order to address such issues caused by these design trends, much effort has been focused into improving cache energy efficiency. [10]

The power dissipation is a major issue for the short channel devices and the performance of digital integrated circuits is challenged by higher power consumption. The mainstream SRAM-based cache technologies need to be improved in terms of power, performance, and scalability. On the other hand, SRAM employs standard technology and has a faster access time, but its primary drawbacks are poor density, high leakage power, and limited scalability [11] for large caches; the most severe problem is the SRAM power leakage. Several memory technologies have been used or proposed for Large Last Level Cache implementation, including SRAM, STT-RAM (MRAM), and embedded DRAM (eDRAM) [12, 13]. Due to the higher density and low leakage, STT-RAMs and eDRAMs are viable SRAM replacements in the context of LLCs. STT-RAM is non-volatile, dense, has read access times equivalent to SRAM, and can be scaled down to 32nm; eDRAM is dense and has low leakage compared to SRAM, with access times and energy close to SRAM.

For the Large Last Level Cache, an alternate memory technology such as Volatile eDRAMs or Non-Volatile STTRAM is explored to address the many difficulties of SRAM. Dynamic RAM-based cache is integrated on the same chip as the CPU, called eDRAM. It becomes a suitable alternate over traditional SRAM whereas STTRAM is another magnetic-cell based cache technology. Density of both eDRAM and STTRAM can expand memory capacity and, hence offer a better hit rate for caches than a comparable SRAM-based cache. These technologies provide perk of low leakage over SRAM but have some weak points. eDRAM has attained a commercial stage in the current scenario. This integration of CMOS technology in particular have a substantial overhaul of the refresh operations which we are targeting in our work [14, 15]. STT-RAM has a longer write latency and needs more write energy compared to reads. In addition, the STT-RAM-based caches also suffers from a write endurance issue, i.e., the cells wear out after a specific number of write operations [12]. Endurance affects the overall lifetime of the cache. We have various wear-leveling techniques to address endurance issues in the STTRAM, i.e., Inter-Set and Intra-Set Variation-based strategies.

DRAM based Embedded DRAM(eDRAM) is also gradually becoming matured for upcoming caches [16]. An on-chip eDRAM Level 3 or L3 cache is included in the IBM BlueGene/L server [17]. IBM's newest CMP high-frequency (5.2 GHz) device uses eDRAM to provide a 24MB shared L3 cache. This device contains four cores, 1.5 megabytes of private SRAM L2 cache, and a shared L3 cache of 24 megabytes. In 2010, The on-chip L3 cache of the IBM POWER7 CPU was made of 45nm SOI eDRAM [2]. This version is further extended up to 22nm in POWER8 CPU [18]. To keep its data, eDRAM must be updated on a regular basis. As a result, eDRAM consumes power to update its data in addition to leaking power. Given these considerations, dynamic voltage scaling appears to be a difficult approach for reducing static power usage in eDRAMs.

More details about the working methodology and the preliminary concepts and characteristics of different memory technologies is discussed in Chapter 2.

1.6 Motivation

While CMOS technology allows for more transistor integration on a device, energy, power, and temperature have emerged as the critical limits for more capable systems. In a system with CMP, the workload is also memory intensive with large working sets. Unfortunately, a substantial portion of the energy required by a big processing chip is lost through leakage. Traditionally SRAM leakage in the on-chip memory hierarchy, in particular, is a substantial source of energy loss. LLCs are built to have a big capacity in order to store frequently accessed data in higher-level cache. LLC memory modules take up a large portion of the chip's real estate. Memory technology having a high density and equivalent read time to SRAM qualifies for LLC implementation. This motivates us to explore alternative memory technologies for caches.

For Large Last Level Cache, an alternate memory technology such as embedded-DRAM (eDRAM) and Spin Transfer Torque RAM (STTRAM) are being investigated to address the problems of SRAM. We get the benefits of dense implementation and low leakage power consumption. When compared to SRAM, eDRAM caches give approximately eight times the density benefit. STTRAM, on the other hand, is non-volatile and dissipates negligible static energy. These memories also suffer from some issues. Due to the capacitor charge leakage in eDRAM, it has the shortcoming that it needs to be refreshed, and a significant amount of refresh power is required for this operation. In eDRAM, refresh latency limits its use due to refresh overhead. On the other hand, STTRAM suffers from high write current, and cells abide by a long retention time. eDRAM consumes a tremendous amount of energy in refreshing cells. STTRAM on the other hand, have a severe downside of restricted "write endurance" in addition to greater access latency. eDRAM and STTRAM are most popular in terms of commercial availability and maturity. To mitigate the expensive write operations, researchers use the best characteristics of each memory technology available.

All the above factors motivate us to develop strategies in order to fetch the best possible characteristics of eDRAM and STTRAM, i.e. high density and low leakage energy. Our work is more inclined towards developing these technologies suitable for large LLCs with minimal overhead. We have proposed different refresh management policies in eDRAM. In order to mitigate the write overhead of STTRAM, we have proposed the volatile-STTRAM (i.e., Relaxed Retention Time-STT-RAM and Relaxed Write-Energy STTRAM). In particular, when we do relaxation over costly write operations leads to new issues. The relaxation on writing current leads to Read disturbance error, and relaxation on the retention time of STTRAM demands periodic refreshing over the cells. These problems, on a broader perspective, behave precisely like refresh management in eDRAM subjected to conditions. All these circumstances encourage us to develop techniques to mitigate the errors mentioned earlier. These leading issue of refresh management is addressed by our different contributions over volatile-eDRAM and volatile-STTRAM. In order to avoid refresh operations, we can have some basic strategies to follow a pragmatic direction:

- Checking Temperature of the cache memory module
- Tracing cache access behaviors through cache references

- Inspecting type of data over the cache block

To avoid the need for specialized external components over the chip, we have avoided temperature-based strategies for refresh optimization. However, Tracing access behavior involves protocol attributes, and inspecting data cache blocks may demand minimal architectural augmentations. The research aims to produce refresh management strategies for volatile memory technologies and maximize the benefits of using them for large last-level caches. To facilitate this, we made different contributions in the following directions.

- Optimizing Refresh Operations by tracing cache access behaviors with the help of coherence protocols
- Skipping refresh operations through inspecting the content of the cache block

1.7 Thesis Objectives

The aim of of this thesis is to design robust refresh management techniques for Large-LLCs while keeping a check on the performance and overhead of the system. The LLCs used are volatile, in that we use e-DRAM and low retention STTRAM based memory technologies. Our contributions include techniques amalgamated with coherence aware access types, access behaviors and data block content. We have introduced cache refresh management techniques that skip refresh operations through inspecting cache data blocks.

In order to optimize refresh operation overheads in eDRAM, we have following objectives

- Optimise refreshes by inspecting the cache blocks content
- Optimised refreshes by inspecting the type of block access using the coherence engine.

Similarly, we have proposed volatile-STTRAM, and maximize the benefits of using them for last-level caches. We have made different contributions for enhancing volatile-STTRAM as a better alternative over SRAM. As explained in the motivation section we propose to use relaxed version of STTRAM. This, relaxation triggers issues that exactly shows refresh-like behaviour. Towards this our objectives are:

- Mitigation of side effects of low write current in STT-RAM Caches in particular reducing the impact of read-disturbance errors.

- Designing methods for optimal block placement while using volatile-STTRAM caches having multiple retention time partitions.

1.8 Thesis contributions

The significant contributions of this thesis can be summarized as follows:

1.8.1 Refresh Optimisation through Dynamic Reconfiguration based policy in eDRAM

We have analyzed based on available literature and observed a significant amount of zero data blocks. These data blocks are periodically refreshed in our baseline, consuming significant refresh energy. In our refresh management strategies, we are trying to optimize the refresh operations for such zero data blocks to reduce overall energy consumption. Besides inspecting zero data from the cache blocks we have analysed the flow of zero data blocks from the main memory. We have reported in our motivation that the flow of Zero Data Block varies throughout the execution. In this work, three approaches are proposed using different refreshing techniques over the eDRAM cache. The architectural proposals are as follows:

- Fixed ZVP: Fixed Zero Blocks partition Based Refreshing where we have fixed sized dedicated partition for keeping Zero data blocks, and refresh operations avoided to this partition.
- ZB_ZVP: Dynamic Re-configurable partition based on Zero Block Count, where partition size varies periodically based on Zero Data block count in the previous interval.
- MR_ZVP: Dynamic Re-configurable partition based on Miss Rate. Here partition size varies periodically based on change in miss rate in the previous interval.

We have an optimized periodic round-robin (PRR) refreshing baseline technique. Over which we have shown comparison of our techniques with the existing techniques in eDRAM Caches. We have reported the results on different metrics, where we get a significant amount of reduction among all the matrices. Results included the total Number of Refresh Count Savings, Energy Savings, and Performance. In this work, our dynamic policies performed better over the policy with fixed partition. ZB_ZVP reduces total refreshes by 38%, and the MR-ZVP policy reduces refreshes by 43%. Note that the fixed-size zero value policy gives 29% refresh savings. Savings in refreshes reduce the stall cycles, leading to a performance improvement of 6-9%; and the proposed policies reduce energy consumption by 12-14%.

1.8.2 A Private Block-based Refresh Management technique with Skipping Zero Data Blocks

In this proposal, we have introduced a refresh management policy that assigns an identifying parameter to each block of the eDRAM cache. This technique leverages access behaviour of the cache blocks. The suggested concept is inspired by the fact that the LLC contains many private blocks, the majority of which contain useless and outdated data. So, the key idea rotates around the existence of private blocks during execution, private blocks refers to the cache blocks held exclusively for writing by upper-level cache (ULC). When the first writeback operation from the ULC for such data entries occurs, the real worthy data is present in the LLC. All of these facts have been recognized, and our refresh management strategy is based on them. In particular, cache blocks that are privately cached by upper-level caches will not be refreshed in the eDRAM based LLC. In our refresh management technique, when the private block is put into the LLC on a miss, the entry is allocated with an additional one to-refresh bit parameter. The refresh-bit parameter is set for the corresponding block when a block (known as a shared block, including instruction blocks) other than private blocks is loaded in LLC on a miss. Eventually, the block parameter gets reset when the corresponding state of the block changes to Exclusive. eDRAM periodic refreshing trigger only for which this parameter is set, otherwise avoid refresh operations. Thus, if a block gets evicted from the ULC, blocks are considered refreshing until they are not evicted from LLC. Besides the private block-based refresh management, to further save the write energy in the eDRAM, we have exploited a zero detection-based logic in eDRAM. That detects the zero data blocks at the time of placement of the freshly loaded blocks in LLC. Thus it is an improvisation over the eDRAM in order to reduce the overhead of refresh operation. The proposal results were compared with the existing baseline technique: Periodic Round Robin Based Refreshing: Periodic Round Robin Based Refreshing. In particular, the following variations are proposed:

- Private Blocks Based Refreshing (PBOR), i.e., with private block-based data
- Private Blocks Based Refreshing with Skipping ZDB (PBOR_SZ), i.e., PBOR with skipping refresh over Zero content blocks

The different variation of the policies saves the refresh operations in the eDRAM Cache in the range. This work, PBOR reduces total refresh operations by 55%, and the PBOR_SZ policy reduces refreshes by 55%. The PBOR without zero value policy gives 29% refresh energy savings. Savings in refreshes also reduce the stall cycles, leading to a performance improvement of 6-9%; and reducing energy consumption by 12-14% in the proposed policies.

1.8.3 Read Disturbance Error mitigation using Coherence And Temporal Locality STT-RAM Caches

STTRAM has an issue of asymmetrical read and write operations with high write energy. We can get a relaxed version of STTRAM with write current closer to read current on feature scaling. The read current's induction effect creates disturbance among the MTJ cell, which leads to the flipping of bit orientations. Thus, it creates RDE issues in STTRAM caches. In order to mitigate the read-disturbance error (RDE) we could perform a restore operations in volatile-STTRAM. It can be categorized similar to refresh-like issue in the STTRAM. In eDRAM, refresh management is an issue due to leakage, whereas we have to ensure the retention of the correct data block after reading. Thus, a presumptive solution is to perform a write operation after each read operation in the cache. However, this will trigger an insignificant rise in system overhead. Thus, we require strategies for replenishing data over the cells after each read operation. There could be the possibility of flipping of data due to closely read and write energy. This work includes three techniques to mitigate the read disturbance error in volatile-STTRAM LLC.

- SKIP-WH: This policy skips restore operations for reads when the upper-level cache requests the data block with exclusive permission.
- RD-BUF: This policy eliminates restore-operations for read-intensive blocks by storing these blocks in an SRAM-based read buffer
- SKIP-ZERO: the third one skips restore operations for zero blocks.
- CORIDOR: Combination of SKIP-WR, RD-BUF, and SKIP-ZERO policy.

All three proposals use different properties of the block access pattern to avoid restore operations. Depending on the application characteristics, a specific solution may work for one over the other. These proposals are complementary to each other and can be combined to give bigger benefits in restore savings. So, we have proposed a fourth policy which is a combined version of all the policies. The combined policy version gave 31.6% savings in total energy over the HCRR baseline, whereas an ideal RDE-free STT-RAM cache gave 42.7% savings. Thus, our proposal achieves three-fourths of the energy-saving achieved by the ideal cache. The proposal also saved 51% energy incurred by the restore-operations by HCRR. Savings in restore operations also reduced the stall time for the cache banks. Further, it reduced execution time to 0.64 times the HCRR, which is almost the same as that of the ideal STT-RAM cache.

1.8.4 Coherence Aware Refresh Operation Optimization and block Placement in Multi-retention STT-RAM Caches

The cache blocks inside LLC do not exist throughout the execution; eventually, they are replaced by the other fresh blocks from the main memory. So, cache block's existence time inside LLC varies depending on the re-usability or on the type of blocks. For example, the data inside the cache block remains unchanged for an instruction block until the block gets evicted from LLC. By default, the STTRAM has a retention time of up to 10 years. A retention time refers to the amount of time data is reliable on a cell and can be sensed correctly. Relaxing the retention time allows the STTRAM to write with less energy. We could have multiple partitions representing regions with different retention times in a single cache; each represents a specific retention interval. Here we proposed a multi-retention cache, which gives cells various retention times in a single cache. Freshly loaded data blocks allocated to different regions. Sometimes, the retention time is shorter than the duration for which data blocks must remain in the cache, or it might get updated soon. In order to prevent this premature eviction or frequent update, we can do an initial assignment of freshly loaded blocks. Such that blocks could get a justified initial allocation. Thus, it would save many unnecessary refresh operations. Needful migrations can be done during runtime in order to leverage the advantage of retention time. A periodic refresh, similar to the DRAM refreshing management type, ensures data retention time over the cells. This work proposes a refresh optimized block placement technique for multi-retention Hybrid Cache architecture comprising ways with different retention intervals.

According to the protocol perspective, data blocks loaded on write access is likely to get more writes in the future, updated more often than the data loaded on read miss. Therefore, assigning high retention intervals to such an ever-changing data block would not be a viable solution. Similarly, instruction blocks are loaded on read miss never change throughout the execution could be allocated to the highest retention time region.

In this work, we proposed a coherence protocol-based approach, which contains the following strategies for refresh management in relaxed-STTRAM cache:

- an efficient policy for choosing best retention time for blocks placement in multi-retention STTRAM Cache, and
- Also proposes a migration policy to relocate the blocks to appropriate regions during runtime.

Furthermore, our improvisation strategy identifies zero data value blocks and does not refresh them by skipping the refresh operation for such cache ways dynamically. It would lead to enhancing the energy efficiency, whereas our proposal safeguards performance loss.

The proposed scheme results are compared with baselines SRAM, Pure STT-RAM, and with two existing policies, DRS and CACHE REVIVE, with no specific placement policy. The proposed work contributes to savings in the number of refreshes by 41% leading to a 36% improvement in total energy compared to baseline. The policies also improve performance over the baseline refresh policy and stay within 5% of the SRAM.

1.9 Summary

In order to fulfill the increasing demand for more cache capacity due to integrated CMPs in the processing cores, larger last-level caches are deployed over the chip. In the cache hierarchy, LLCs play a crucial role by providing needful data to the hungry CMPs. SRAMs are not scalable and need improvement in terms of power, performance, and scalability. In order to deploy large LLCs, researchers are working over the fabrication of caches with alternative technologies containing benefits over the conventional SRAM. These technologies include high scalability, lesser leakage power, and more capacity in the same area footprint of SRAM. However, we need to explore the best out of these alternate options as they are not free from issues.

In eDRAM, we have to deal with the charge loss on the capacitors by periodic refreshing. This periodic operation leads to interruption of normal cache access, if not appropriately managed, can reduce IPC. In this dissertation, we aim to address the reliability of data over the volatile caches by ensuring the retention of data blocks. Here, we deal with volatile memory technologies' challenges and make them suitable candidates for cache hierarchy. Similarly, in STTRAM, the asymmetrical read and write with the costly write operation. Thus, it can be addressed by the relaxed versions of STTRAM(i.e., Retention Time Relaxed STTRAM and Write Current Relaxed STTRAM).

In order to reduce the costly refresh operations in eDRAM, we have proposed refresh-free partitioned architecture, which is reserved for Zero data content. In this way, we could reduce a significant amount of refresh operations over the eDRAM cache. In this regard, we have tracked the presence of Zero data blocks coming from the main memory and inside the cache on writeback. In order to make a dynamic flavor, we have made these partitions re-configurable depending on various parameters. These motivational parameters are reported with needful experiments. For re-configurable parameters, we have considered interval-based zero data block count and miss rate as our key parameters.

Besides the above proposal, we have presented another similar refresh management technique using a protocol-based strategy. We have been analyzing access behavior by protocol. For that, we have tracked down the exclusively held data blocks by upper-level cache and found a significant amount of private blocks data that can be skipped while performing refresh operations. The data blocks loaded for the write-requests will be modified soon by the ULC and eventually written back to the LLC. Hence, the LLC copy became out of date when the block was held exclusive by ULC. This approach is further improvised by restricting the refresh and write operations to detect zero data blocks in LLC.

After targeting refresh management issues, similar refresh-like issues are also addressed in the volatile STTRAM caches. In order to reduce the write energy overhead in STTRAM, we have proposed a relaxed version known as volatile-STTRAM. This relaxed version of STTRAM shows volatile behavior that triggers two common issues:1) Read Disturbance Error and 2) Periodic Refreshing. In order to mitigate read disturbance error in the STTRAM cache, we have proposed a locality-based technique for restoring the disturbed cache lines in the STTRAM-LLC due to read operation. Whereas, for Retention time management, we have to regulate refresh operation. We have performed refresh optimization over the multi-retention cache through the blocks placement strategy. Our proposals reduce the restore/refresh operations significantly with minimal impact on performance.

1.10 Organization of Thesis

The rest of this thesis is organized as follows:

- Chapter 2 summarizes the background and prior works related to the contributions of the thesis.

- Chapter 3 presents the refresh optimization through dynamic reconfiguration in volatile-eDRAM caches.
- Chapters 4 presents the contribution, which is the private block and skipping zero based refresh management technique for the volatile-eDRAM cache.
- Chapter 5 discusses coherence and temporal locality based techniques to mitigate read-disturbance issue in STTRAM cache.
- Chapter 6 addresses volatile behaviour of multi-retention STTRAM cache through block placement strategies.
- Chapter 7 finally concludes the thesis.

CHAPTER 2
Background

There are several memory technologies at the maturity stage that can be considered for implementing the LLCs. The commercial availability of the SRAM is far better than other technologies. As we have discussed, we had to face challenges when we raised the size of the LLCs for the CMPs system. In consideration of the upcoming large working set size arises an alarming situation for leakage power consumption. In particular, dynamic energy consumption plays a vital role in on-chip processing modules. However, the leakage comes from the memory module dominates over the other fraction of on-chip power consumption. If we switch to other volatile caches technologies with low leakage power consumption than SRAM, we have to ensure retention time of data block over the cache. In order to check the integrity of cells, we must require regular check operations. For example, in the case of DRAM, we require Periodic Refreshing for replenishing data on the cells. These periodic refreshes consume significant refresh energy in terms of dynamic energy. It is commonly known that today's chip multiprocessors' large last-level caches contain a lot of unnecessary data. As a result, there is a chance to reduce refresh energy consumption by not refreshing data that is no longer useful for program execution. The problem is locating such information at a reasonable cost.

The main goal of this work is to ensure the reliability of data in the volatile-memory-based technologies and utilize them as large last-level caches. We had a detailed survey over our targeted volatile-eDRAM and Volatile-STTRAM technologies. We initially summarize the preliminary concept of corresponding memory technologies followed by challenges with the employment in the cache hierarchy.

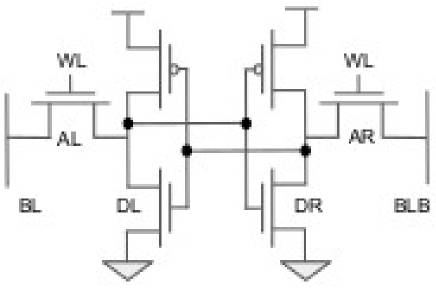


Fig. 2.1 6T SRAM cell schematic.

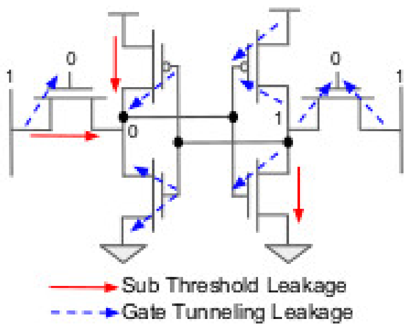


Fig. 2.2 SRAM leakage paths.

2.1 Volatile Memory Based Caches

Volatile memory is the memory that requires a minimal continuous power supply to retain information over it. The loss of charge happens in the form of leakage i.e., loss of data from the memory cells.

2.1.1 Static Random Access Memory (SRAM)

SRAMs are the most common memory technology used for the caches. [19] It contains six transistors. Although SRAM does not require continual electrical refreshes, it does require a steady current to maintain the voltage difference.

Figure 2.1 depicts a typical 6T SRAM cell. On a read operation, the access transistors (AL and AR) are turned on, and one of the pull-down transistors (DL or DR) establishes a current route from one of the bit-lines (BL or BLB) to the ground, allowing for the quick differential sensing operation. On a write operation, the cell content is written based on the write driver's differential voltage signal applied to the bit-lines.

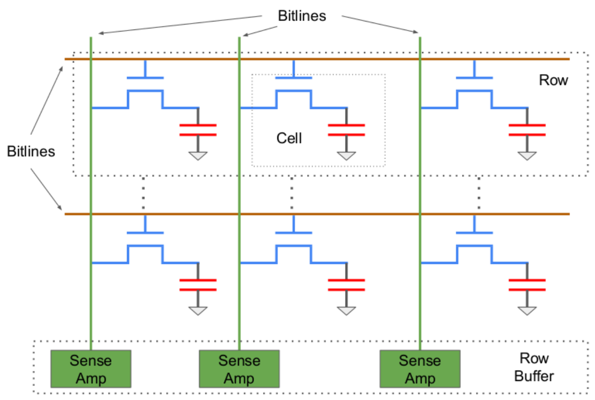


Fig. 2.3 DRAM bank structure

The standard CMOS technology can be used to make SRAMs. They are also the most extensively utilized embedded memory technology because they allow quick memory access. For example, SRAM is used to implement the whole cache hierarchy in Intel, AMD, and Sun processors [18–20]. SRAM cells, on the other hand, are huge in size because of their six-transistor implementation. The cell structure introduces sub-threshold and gate leakage channels (Figure 2.2), resulting in high standby power. SRAM is less suitable for implementing high-capacity caches due to its low density and significant leakage. Each bit requires six transistors in a static RAM chip, but dynamic RAM only requires one capacitor and one transistor.

2.1.2 Dynamic Random Access Memory (DRAM)

DRAM is made up of a two-dimensional array of DRAM cells, each of which has a capacitor and an access transistor, as shown in Figure 2.3. A cell is in the charged state or the discharged state, depending on whether its capacitor is fully charged or entirely discharged. These two states represent a binary data value. Every cell, as shown in Figure 2.3, is at the crossroads of two perpendicular wires: a horizontal wordline and a vertical bit line. A bitline joins all cells in the vertical direction (column), and a wordline connects all cells in the horizontal direction (row). All the access transistors inside the row are enabled when wordline a row is raised to a high voltage. It further links all of the capacitors to their corresponding bit lines. The row's data (in the form of charge) can then be transferred into the row buffer depicted in Figure 2.3. The row-buffer, also known as sense-amplifiers, reads the cells' charge and destroys the data in the process on the cells. Thus, an immediate write of data occurs from the row buffer to the cells (wordline). In other word accessing a row can be considered auto-refresh.

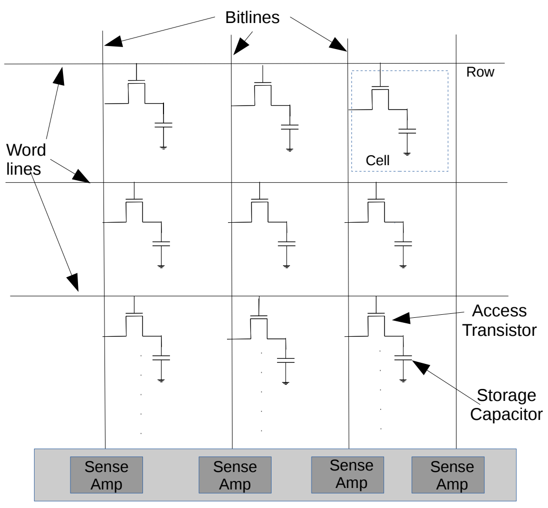


Fig. 2.4 Representational View of eDRAM bank

Refreshing DRAM: The charges over the DRAM cells are not stable. The capacitor stores the charge and leaks the charge in the form of sub threshold leakage and gate-induced drain leakage(discussed in chapter 1). Eventually, the charge over the cells may get reduced up to a level after which it would not be possible to sense it correctly through a sense amplifier. This leads to loss of data from the word line. The duration over which the cells can retain the charge (or data value) is called the retention time of the cell.

The Periodic refreshing provides a replenishing mechanism to the cells. In order to restore the charges over the cell, one needs to refresh the cell within the retention time. The DDR3 DRAM requirements [20] require a retention duration of at least 64 milliseconds, which means that all cells in a bank must be refreshed at least once during this time frame. Traditionally, Memory refresh has been accomplished by employing an internal counter to regularly refresh all of the cells inside a single eDRAM instance. This is accomplished by forcing a refresh operation and prioritizing it, causing competing reads and writes to the same bank to be queued. Because of its simplicity, the policy is widely used. In the past, the penalty for Refreshes was low enough that adding further complexity was unnecessary.

2.1.3 Embedded-Dynamic Random Access Memory (eDRAM)

DRAM cells have been incorporated with logic-based technology, allowing them to outperform traditional DRAM cells in speed. An embedded DRAM cell (eDRAM) is a logic-circuit technology that incorporates a trench DRAM storage cell. The eDRAM trying best to occupy commercial maturity in the current scenario. In particular, technology integration has a substantial overhaul of the refresh operations we are targeting in our work.

In a research, the read operation latency can be comparable to those of SRAM. As a result we can deploy, Last level caches in modern commercial CPUs using eDRAM technology. An embedded DRAM cell consists of an access transistor and a storage capacitor. The capacitor used for eDRAM is either a *deep-trench capacitor* [21] or stacked capacitor, where a transistor controls the access [22, 20]. Over time, the capacitor losses charge, and hence an eDRAM cell requires a periodic refresh operation. The refresh process involves sensing the cell data to the sense amplifier and replenishing the partially lost charge. This capacitor leakage increases with technology scaling. As shown in Fig. 2.4, data is represented by the charge on the capacitor. The Access transistor connects the capacitor to the bit lines, which transfers the data bits to the sense amplifier. The row to be selected is controlled by the word line connected to the transistor. Barth [23] has reported the refresh period of eDRAM to be 40us, whereas it is 64ms in the case of conventional DRAM.

The other properties of eDRAM can be summarized as follows:

Advantages

- Low leakage based memory module
- Larger EDRAM caches always perform better than the smaller SRAM caches. [24]
- In present scenario, researchers implementing eDRAM cells on FinFET technology. [25]

Limitations

- eDRAM requires more refresh operations compared to conventional DRAM.
- The retention time of eDRAM is 10^3 times lesser than normal DRAM requiring one to refresh eDRAM more often.
- Not a good choice for L1-caches due to slow Read and write operations.

The 1T1C eDRAM and the gain cell eDRAM are the two most prevalent varieties of eDRAM. Both of them provide a capacitor to store all the data. A 1T1C cell, for example, stores its data on a dedicated capacitor, whereas a gain cell relies on the gate capacitance of its storage transistor. The data-retention period of an eDRAM circuit determines its refresh rate, which is governed by the rate of cell leakage and the size of the store capacitance [26].

1T1C eDRAM Cell

It has an access transistor and a capacitor (C). The 1T1C cells have a higher density than gain cells, and the cell capacitor fabrication requires some extra stages. The access transistor is turned on, and an electrical charge is transferred from the storage capacitor to the bit-line, enabling the cell to read (BL). As the capacitor loses charge while being read, the read operation is referred to as a destructive operation. In order to recover the lost bits after a destructive read, data write-back is required. Moving charge from BL to C is used to write the cell.

A DRAM cell loses charge over time due to junction leakage, gate-induced drain leakage, sub-threshold leakage, field transistor leakage, and capacitor dielectric leakage [27]. Moreover, eDRAM has a considerably shorter retention time than traditional DRAM because it uses quick and easy access transistors with a larger leakage current than conventional DRAM. For example, an IBM 32nm SOI eDRAM has a retention period of 40us at 85°C [28], but commercial DRAM has a retention duration of 64ms.

Gain Cell eDRAM

Gain cell memories are typically made up of two or three transistors and offer reasonable leakage, high density, and faster memory access. There are lots of different kinds of GCs with two [29], three [27], and four transistors [30] that have an order of magnitude less leakage power and take up less than half the size of a six transistor SRAM cell.

A write access transistor (PW), a read access transistor (PR), and a storage transistor constitute a 3T PMOS gain cell (PS). PMOS transistors are often used because they have a lower leakage current than NMOS transistors of the same size. Reduced standby power and longer retention time seem to be possible without much power dissipation. The read word-line (RWL) and the write word-line (WWL) are both at VDD during steady-state. Therefore PR and PW are turned off, while the read bit-line and the write bit-line (WBL) remained at 0V. PR is switched on in read mode, while RBL is only pulled up when the voltage at the storage node is low. The cell data is then obtained by the sense amplifier based on the voltage level of the RBL. WWL is negatively over-driven in write mode, similar to the 1T1C eDRAM, to avoid threshold voltage loss. PW is then used to write data into the storage node.

2.2 Non-Volatile Memory Based Caches

There are numerous developing Non-Volatile Memories (NVM) that are currently being investigated. Spin Transfer Torque Random Access Memory (STT-RAM) [31, 32], Phase Change Random Access Memory (PCRAM) [33, 34], Resistive Random Access Memory (ReRAM) [35], Ferroelectric Random Access Memory (FeRAM), Nano Random Access Memory (NRAM), Conductive-Bridging Random Access Memory (CBRAM), Single Electronic Memory (SEM), Polymer, Molecular, Racetrack, Holographic, and Probe are among the 12 NVM technologies. Some of the memories in this list have progressed to the point where they can be commercially produced, while others are still in the early phases of development.

In this dissertation, we are restricting to use NVMs as it suffers from asymmetrical, i.e., costly Read and writes operations. Nevertheless, we have used a relaxed version which can be referred as a volatile-STTRAM suitable candidate for large last-level caches.

2.2.1 Volatile STTRAM Caches

A normal STTRAM-based cache could be inefficient, as high write energy consumes a significant amount of the dynamic energy. In order to address this issue, many researchers proposed ideas towards reducing dynamic energy consumption. We have proposed two variants of the energy-efficient volatile-STTRAM cache.

When a large current is applied to an STT-RAM cell for a short duration, the value can be written and read reliably. In order to get the relaxed version of STTRAM, if the current is reduced, a longer pulse duration is required to complete the operation. Both these factors affect the reliability of the cell; in that case, the reads become destructive [36]. The low read current leads to magnetic disturbance in the MTJ, and this can lead to flipping the stored data, thus causing the read disturbance error (RDE). Process variation further exacerbates the issue of RDE [37].

One of the prospective solutions is to mitigate the write latency overhead of STTRAM. STTRAM, by default, has a retention time of up to ten years. A retention time refers to time up to how long data can be retained on the cell. Relaxing retention time can make the memory with lower write energy. Relaxed STTRAM has a better switching performance by properly relaxing its thermal stability and reducing data retention time (e.g., a few seconds) to maintain the trivial performance and energy overhead induced by data refreshing.

This dissertation has a proposed work on a hybrid version of retention-relaxed STTRAM where we have multiple regions with different retention intervals. A multi-retention cache gives choices for cells with various retention times in a single cache. They predict access patterns for direct block placement and migration. Much previous work has explained that how inherent retention time is relaxed in order to make Retention-Relaxed STTRAM [38]. LARS [39] is an approach of adaptable runtime cache depending on the different applications. It proposes an optimal tuning algorithm that determines the retention based on energy consumption at runtime. Sun et al. [5], used retention-relaxed STTRAM to make a hybrid version of the cache with two retention levels. In order to leverage the best features of retention-relaxed STTRAM and considering the versatility of the application, we use multiple retention levels in a single cache.

The challenges and solutions proposed for refresh optimizations in DRAM and eDRAM are not directly applicable to RDE in STT-RAM caches. To mitigate the RDE issue, we need to refresh/restore the data immediately after a read. We cannot wait until some time in the future (called the retention time in DRAM) to initiate a restore. This immediate need to refresh creates further challenges in designing a solution to mitigate them because every solution will affect the hit time of the cache. We use the terms Refresh and restore interchangeably to mean the same thing.

2.2.2 STTRAM Operations

Fig. 2.5 shows the structural view of the STT-RAM cell. The STT-RAM cell consists of an access transistor and Magnetic Tunnel Junction (MTJ). The MTJ contains two ferromagnetic layers separated by a dielectric layer (made up of MgO). A spin-polarized current can control the magnetization direction of the ferromagnetic layer, called the free layer. The direction of the other layer, called a reference layer, is fixed. The free layer's magnetization direction is used to represent the data bit stored in the STT-RAM cell. When the free layer's magnetization direction is parallel with the reference layer, the resistance of MTJ is low and represents the '0' state of the STT-RAM cell (shown in Fig. 2.5(b)). On the other hand, when the magnetization direction is anti-parallel, the resistance of MTJ is high, and it corresponds to the '1' state of the STT-RAM cell (Fig. 2.5(c)).

The read operation in the STT-RAM is performed with the help of a source line and a bit line. A small voltage is applied to these lines, which in turn generates the current. The generated current is compared with the reference current to detect the '0' and '1' state of the STT-RAM cell. Writing '0' in the STT cell is performed by applying a large

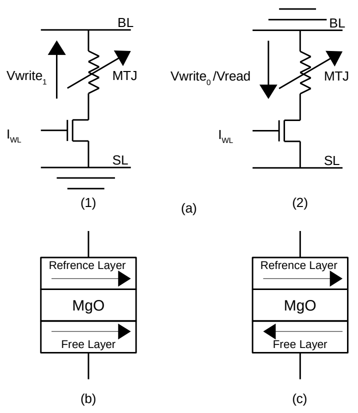


Fig. 2.5 Overview of STT-RAM cell (a) Schematic STT view with (1) Write ‘1’ operation (2) Write ‘0’ and Read operation (b) Parallel low resistance, representing ‘0’ state (c) Anti-parallel high resistance, representing ‘1’ state

positive voltage between the source line and bit line. On the other hand, writing ‘1’ is performed by a large negative voltage.

2.3 Terms Related to Refresh Operations

2.3.1 Retention Time

The duration over which the cells can retain the charge(or data value) is called the retention time of the cell, and one needs to refresh the cell within this retention time. As the retention time reduces, we need to either write back the data block or refresh the contents to maintain the data integrity. A refresh includes the block being written to a buffer and then re-written into the cache. The frequency at which we perform this Refresh is called the refresh interval. The retention time of eDRAM is 10^3 times lesser than regular DRAM requiring one to refresh eDRAM more often. [23]

Effect of Temperature on Retention Time: The refresh operation uses a significant amount of energy, it’s critical to choose the right refresh interval, which is heavily dependent on temperature.

2.3.2 Refreshing Mode in Volatile Caches

Refreshing modes regarding targeted technologies are explained in the following subsections. At the end of retention time, in order to retain the data, the values have to be replenished by reading them and writing the same value back to the location referred

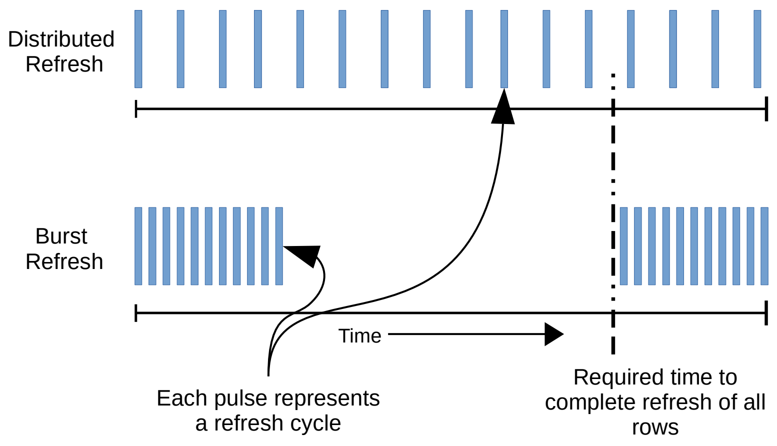

Distributed Refresh
Burst Refresh
Time
Each pulse represents a refresh cycle
Required time to complete refresh of all rows

Fig. 2.6 Representation of Distributed and Burst Refresh Mode

to as a block-level refresh. The negative impact on performance and energy can be mitigated by designing schemes to avoid the Refresh wherever possible. Methods to do this include staggering the refreshes so that the duration of the stall of each bank is distributed evenly, and we can avoid cases when the whole memory is not-available. Other techniques include avoiding periodic Refresh to those blocks that have recently been accessed and scheduling phase-wise refreshing.

2.3.3 Embedded-DRAM Refresh

eDRAM requires a higher refresh rate (or refresh power) to maintain data integrity on the cell. Reducing refresh overhead in eDRAM could save a lot of chip power. Figure 2.6 shows the most basic refresh operations on eDRAM can be performed in two ways [40] (i) distributed or (ii) burst form. In distributed, the refresh operation for all cache lines is distributed along the retention time in the eDRAM bank; Whereas the burst technique issues a refresh command for all the lines in a cache bank at one time in a burst. Both the techniques have advantages according to cache capacity. Normal cache access requests are blocked when the cache bank is busy with the refresh process [41]. eDRAM Cell loses charge through the off-drain current through the access transistor. The refresh process consists of sensing the cell data from the cell to the sense amplifier and replenishing the partially lost charge. A row of the sense amplifier is connected to make an array called row buffer. The wire that connects the eDRAM cell to the sense amplifier is called bit line. Each row is read during the refresh interval, and data is transferred to row buffer followed by written back to row. So, refreshing a row is similar to accessing a row with the help of a row buffer. eDRAM has a substantially shorter retention time than DRAM, it is impracticable to suspend refresh processes in eDRAM for an extended period of time. Furthermore, eDRAM stacked caches have a significantly higher number of read/write operations than DRAM main memory systems. In such circumstance, a refresh activities cannot be suspended indefinitely.

2.3.4 Volatile-STTRAM Refresh

In order to maintain data over the cells in volatile-STTRAM, refresh or restore operations are required. A temporary buffer or row buffer can be used depending on the strategy of refreshing/restoring. Each refresh operation consists of four physical operations to be performed while refreshing a block: 1) STTRAM Read, 2) Refresh Buffer Write, 3) Refresh Buffer Read, and 4) STTRAM Write. The negative impact on performance and energy can be mitigated by designing schemes to avoid the Refresh wherever possible.

Write Current relaxed STTRAM: Calculating the exact probability of an RDE would require detailed modeling of circuit-level parameters and effects. Since this is beyond the scope of our architecture-level study, we assume a 100% probability of RDE occurrence in this paper. In other words, we assume that an RDE happens on each read operation. Thus, we write the data back to the cache line after sensing it on read buffer on each Read, as we assume it is destructive Read.

Retention-time relaxed STTRAM: This refresh operation stalls the memory bank until the Refresh is completed, and this also leads to additional energy consumption. Here, the retention time value will be considered from the timestamp value of the last access. We trigger periodic refresh operation for cache line until the cacheline is referenced by ULC. As soon as cacheline is accessed, the timer for the corresponding line will be updated with a new timestamp value. However, we can use those refresh techniques for the same which we used in the case of eDRAM refresh management. The retention interval will be considered from the timestamp of the last reference to the cacheline.

2.3.5 Multi-Retention STTRAM

Shortening the retention time STT-RAM in which cell minimizes its write dynamic energy and latency of STT-RAM by lowering data retention time. This can be accomplished by lowering the free layer's surface area. Two variants of data retention time decreased STT-RAM is shown in reference [45]: 26.5 s and 3.24 s. STT-RAM cells are used in all cache levels in work in References [42, 45] because they have lower write dynamic energy and latency. However, in order to keep the data in their STT-RAMs, they must perform periodic refresh operations. DRAM-style Refresh was utilized in

Reference [42]. (refresh all cache lines). First, our research focuses on large LLCs where dense STT-RAM-based caches plays important role. It has higher write costs than SRAM caches, which can be reduced by relaxed-STTRAM caches. Second, our research aims to lower the system's energy usage while also improving its performance. In particular write operations consumes energy in STT-RAM-based caches, a more aggressive cache partitioning strategy is required. This inspires us to investigate a STT-RAM-based LLC design with several retention levels.

2.4 Challenges to Employ Volatile Caches

Static power consumption becomes an essential worry as many cores use dynamic energy more effectively increase. Leakage through on-chip cache memory modules adds significant overhead to the chip's power. We must use a low leakage cache memory model in manycore systems. As maximum portion of the chip occupied by cache, large caches would consume a significant amount of energy. To draw the com

1. Periodic Refreshing:
2. Effect of low write current:
3. Effect of Temperature:
4. STT-RAM with a long retention time has a high write dynamic energy. The major challenge in multi-retention cache management is whether the data should be stored in short- or long-retention STT-RAM.

2.5 Cache Coherence

2.5.1 Inclusive and exclusive caches

In an inclusive cache system, shared cache memory has a copy of every cache block in all the private caches. A block is first searched in the private caches on a cache miss, followed by a search in shared caches depending on the cache hierarchy. In case blocks are not present in LLC, the cache block is fetched from the main memory into the private caches via the shared cache. When a cache block is evicted from a shared cache, all of its copies in private caches get invalidated such that inclusivity is never violated. Eviction of a cache block from any private cache does no effect on its counterpart in the shared cache.

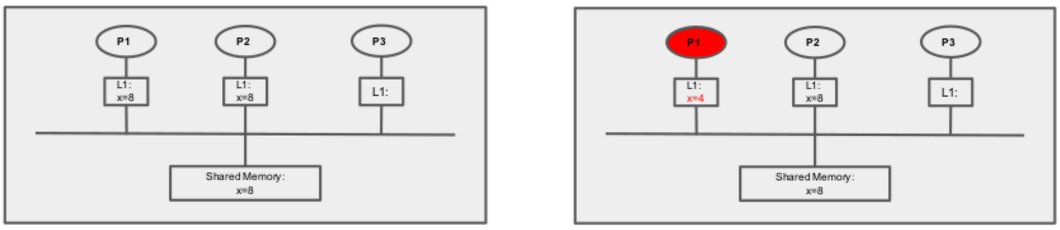


Fig. 2.8 Non-coherent view of cache memory hierarchy

For an exclusive cache system, on a miss in both the private and shared caches, the cache block is brought directly into the private cache, bypassing the shared cache. On eviction from the private cache, it is transferred into the shared cache. In the exclusive policy, this is the sole option to populate the shared cache. When a cache block is hit in the shared cache, it is sent to the requestor's private cache, and its duplicate in the shared cache is invalidated. In our work, we have investigated inclusive cache policy.

2.5.2 What is coherence?

In a multiprocessor system, cache coherency is defined as the consistency of shared resource data across the cache memory hierarchy as seen by any of the processors. Data is shared throughout multiple threads that have been part of the same application. This could lead in many threads acquiring and modifying the same data at the same time. Private caches are maintained in synchronization to guarantee proper execution of the program. This is achieved by using shared cache folders. The directory stores information about each L1 sharer who already has cached that data.

Coherency checks the uniformity of shared resource data across the cache hierarchy visible to all the processors in the CMP. A multiprocessor memory system is coherent if the results of an execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which:

1. Operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor.

2. The value returned by each read operation is the value written by the last write to that location in the above sequence.

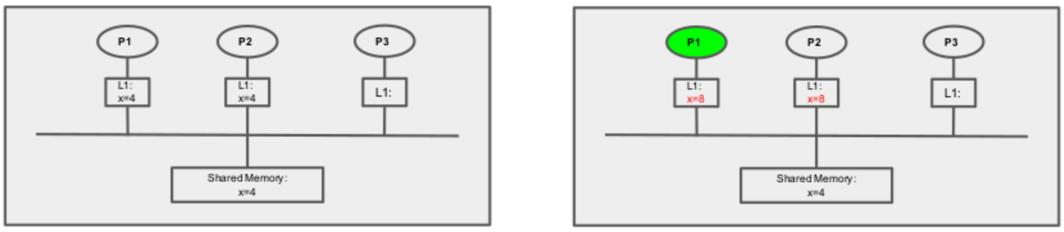


Fig. 2.10 Write-Update Policy After condition

2.5.3 Write-Update and Write-Invalidate Caches

The system could perform multiple approaches to update the data in any processor's private caches. There are two approaches for propagating writes to other processors' private caches.:1)Write-update, and 2)write-invalidate are the two commands.

In the write–update policy (shown in figure 2.7), an update to 'x' in P1's private cache immediately updates its copies in private caches of other processors. The processor that is writing the data broadcasts the new data over the bus (without issuing the invalidation signal). All caches that contain copies of the data are then updated.

In the write–invalidate policy (shown in figure 2.8), an update to 'x' in P1's private cache invalidates its copies in private caches of other processors. Before changing its local copy, the processor that is writing data invalidates copies in the caches of all other processors in the system. This is accomplished by the local computer sending an invalidation signal over the bus, causing all other caches to look for a copy of the invalidated file. The data on the local system can be updated until another processor requests it after the cache copies have been invalidated. Here we take benefit of parallelism and update propagates whenever data is read. Thus, it reduces traffic over the bus. However, several updates can take place before communication. Invalidation cost is still an issue before starting writing on the copy.

The write-update scheme differs from write-invalidate in that it does not create only one local copy for writes. Its advantage is that any delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme since the written data are immediately updated in the reader's cache. We also get some disadvantages where we require multiple writes by one processing element (PE) before another PE reads data. Also, Junk data accumulates in large caches (e.g., process migration).

We examine the write-invalidate policy in this study because it saves CPU cycles by not updating copies in other caches straight away.

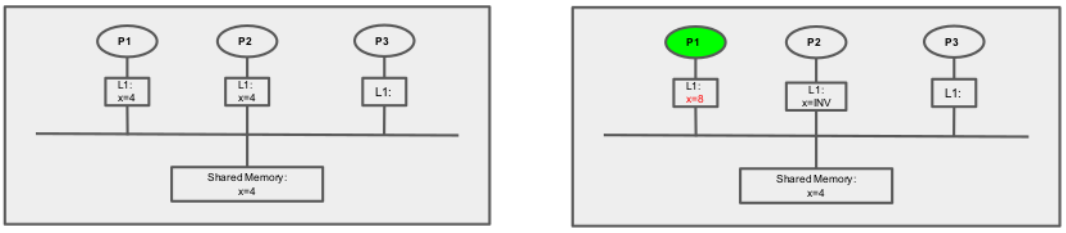


Fig. 2.12 Write-Invalidate Policy After condition

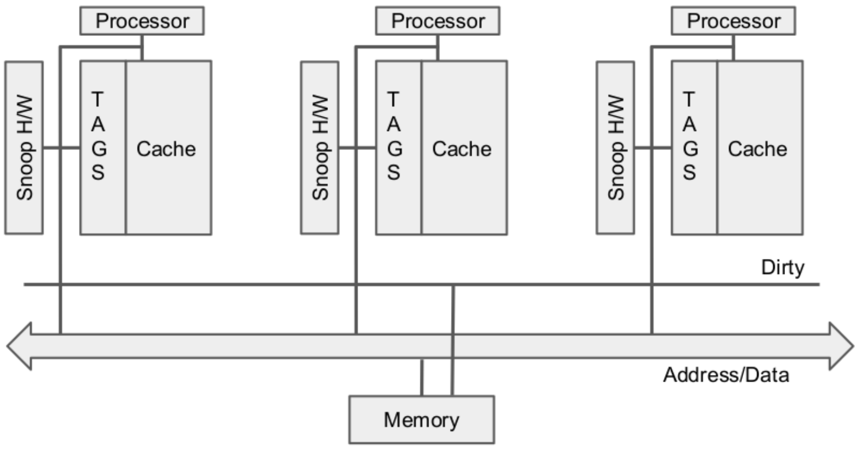


Fig. 2.13 Snoopy Protocol

2.5.4 Write-Through and Write-Back Caches

The write-through and write-back strategies ensure that writes to the shared cache are propagated on a change to the data in any of the private caches. If P1 updates the value in block ’x’ under the write-through policy, as shown in figure 13, the update is promptly propagated to shared cache; but, in write-back policy, as shown in figure 14 (a) and (b), the update is NOT propagated to shared cache immediately (b). Figure 14 (c) and (d) demonstrate how the copy in the shared cache is updated only when the following events occur:

1) An updated block is ejected from L1.

2) Another processor requests an updated block.

If coherence is discovered suitable coherence actions can be done. Snoopy protocols are what they’re called.

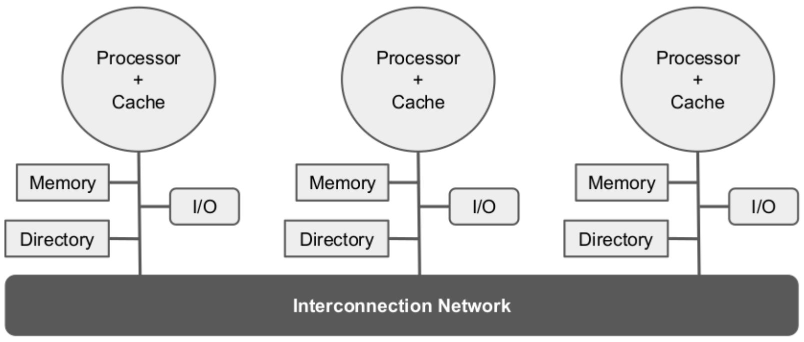


Fig. 2.14 Directory Based Protocol

2.5.5 Snooping Protocol (Snoopy)

As each cache snoops bus transactions to monitor memory operations of other processors, the name snoopy derives from the word snoop (Fig. 2.13). Snoopy protocols are limited to small-scale bus-based multiprocessors since they necessitate the usage of broadcast media in the machine. The most frequent way in commercial multiprocessors is this form of protocol. Snoopy protocols have been proposed in a variety of ways. This protocol is inclined to bus-based multiprocessor systems, where a suitable coherence action can be done. Snoopy protocols are extensively used in commercial multiprocessor systems like Pentium4 and IBM PowerPC.

2.5.6 Directory Based Protocol

The cache coherence protocol that does not rely on broadcasts will keep track of and preserve the locations of all cached copies of every block of shared data. Technically, the directory where this protocol maintains the cache block location can be centralized or distributed in the cache hierarchy. A directory entry with many pointers exists for each data block; request/response is a direct link between the CPU and the directory. Any update to a block in any private caches is messaged to the directory, which then updates/invalidates its copies in other private caches (depending on write-update or write-invalidate policy). A dirty bit is included in each directory entry to indicate whether or not a unique cache has permission to write the corresponding data block. For example, if processor P1 asks for a copy from P2's private cache, P1 makes the request to the directory, which further propagates to P2. P2 responds with the needed data, which is transmitted to P1 through the directory. To provide cache coherency, the directory is added to each node(shown in figure 2.14). A directory protocol must handle two core operations: managing read misses and handling writes to a shared, clean cache block.

2.5.7 MESI Coherence Protocol

The MESI protocol is an Invalidate-based cache coherence protocol and is one of the most common protocols supporting write-back caches. Invalidate-based means that if a shared cache block is being written by one of the cache processors, the other one instead of immediately getting the latest copy of that block invalidates the block. Write-back cache means that every write need not be updated in the memory. This results in those caches that contain that cache block invalidating the block. The memory will be updated eventually but later, as discussed below.

States of MESI Protocol

The 4 states of MESI are characterized as:

- **I**: If a block is present in state "I" in its FSM, this means that the data is garbage and of no use. This could have happened because of either of the following reasons-
    1. Block was never brought to the cache
    2. Block was brought but had to be replaced
    3. Block was brought, but some other processor modified it, and the latest value is yet to be fetched.
- **E**: If a block is present in state "E" in its FSM, this means that this is only a copy among all the caches (read-only). FSM can go to state E in the following scenario If it’s a read-miss for the block by the processor and no other processor has the block.
- **S**: If a block is present in state "S" in its FSM, this usually means that this block is present with at least two processors. FSM can go to state "S" in the following scenarios-
    1. When the block was brought to this cache, at least another cache already had the same block in either state "E" or state "S".
    2. Initially, this cache was the only owner, but some other processors also started to read that block into their cache in due time.

  Now, generally state "S" for a block means at least two sharers, but sometimes a cache can be the only owner and present in state "S". This can be illustrated in the following example-

Suppose three processors, P1, P2, and P3, send read requests for the same block X and eventually all get the block. However, let's say P1, and P2 replace block X in due time, thus leaving P1 the only owner, still in the state "S".

- **M**: If a block is present in state "M" in its FSM, this means that the corresponding processor has modified this block and protocol being write-back, this new value is not updated in the memory hence going to state "M". FSM can go to state "S" in the following scenarios-

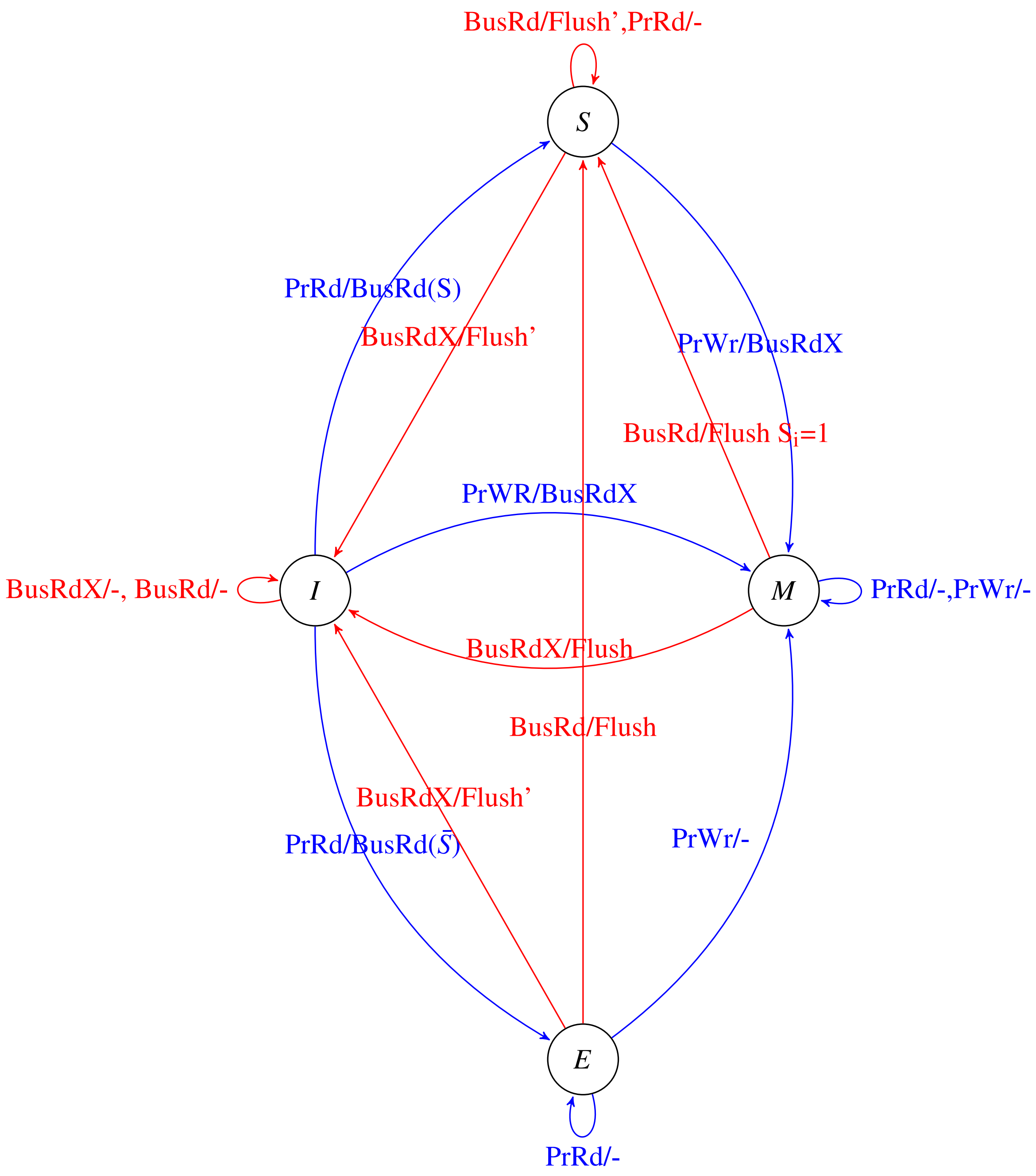
  1. In case of a write-miss by the processor
  2. In case of a write-hit by the processor

Explanation of FSM

Notation

(1) FSM has red-colored arrows for bus actions while blue-colored for processor actions.
(2) Command and its response are denoted as follows Cmd/Action for e.g. BusRd/- means the command was bus read, and no action was required.
(3) Flush' means the processor may have to provide the data depending upon whether the system has cache-to-cache transfer or not.

FSM of MESI Protocol



Working of MESI protocol

In the case of cache read, any cache state other than Invalid can be used to serve a cache read. In the case of cache writes, only possible states of a block are in a Modified or Exclusive state. If the cache line is in the Shared state, we must first initiate a request for ownership, which will invalidate all shared lines. In order to intercept any request to the exact location in the lower memory level, a cache line in the Modified or Exclusive

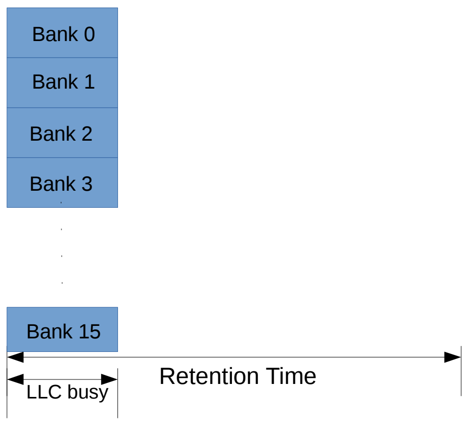


Fig. 2.15 Diagram Representing a LLC containing 16 banks and all the banks triggers refresh operation in burst mode.

state must "snoop" any other cache visits. If in case the line is moved to the Shared state, and if it was previously in the Modified state, it is written back to the lower cache before being forwarded to the requester in the Shared state.

Read requests are normally termed as *GetS*, write as *GetX* and instruction reads are called *GetInstr*. Namely, because reads result in shared blocks, write result in Exclusive blocks and pure instructions are only read and never modified. We will use these distinctions in MESI protocol while proposing our optimisations policies.

2.6 Reducing Refresh Operations in eDRAM

2.6.1 Memory Access Pattern Based Policies

The main idea would be to trace the cache blocks' access pattern during the lifetime of the block inside the cache.

The very naive and basic technique for refreshing the eDRAM is the burst mode. In a Simple periodic refresh technique, all the banks are refreshed parallel as shown in figure 2.15(concerning multi-banking cache structure). All cache lines of all the banks would be busy while refreshing, and cache will not be available while the Refresh is going on. Regular access would be restricted for a predefined interval of time for all the banks simultaneously; an LLC would be unavailable for a long time. This would lead to colossal performance degradation as the critical loads from the main memory are stalled due to refresh operation. This is a naive technique that the maximum DRAM-based model uses for refreshing. If we use DRAM-based technology in the cache, we have to optimize refreshing techniques to get minimal overhead.

If we trigger to perform refreshing over the bank in a single set, this will affect the performance badly. The naive approach to perform the Refresh is burst mode. This work [41] proposes a fine-grained round-robin approach to refresh an eDRAM bank. The refresh operations are triggered across banks in a round-robin fashion. They compute the global refresh interval from the given retention time, which is nothing but the number of cycles after which each bank needs to be refreshed. Depending on the number of cache banks, we divide this global refresh interval such that the banks trigger Refresh in a staggered fashion. The fine-grain round-robin refresh technique minimized the usual request latency caused by refresh operations, but it does not lower refresh energy because all cache lines in the LLC are updated once within the retention period. The results were further improvised by a selective refresh strategy based on inclusive information to decrease refresh operations in the LLC. The cache line in the upper-level cache may or may not be present in the lower-level cache in non-inclusive caches.

Refrint [42] is an intelligent refresh management strategy. The proposed work lies in the category of tracing memory access patterns. They have introduced the concept of refreshing hotlines and cold lines in eDRAM caches. The coldlines refer to those that are not being used or are being used far apart in time but are still getting refreshed, whereas hotlines get access actively within the retention interval and naively refreshed periodically. In their proposal, they have suggested time-based policies for hotlines, where they use to avoid refresh operation on particular cacheline with an assumption that cachelines get automatically refreshed when ULC is accessing them. For coldlines, they have suggested data-based policies, which detect and refresh only those cache lines that are likely to be utilized again in the near future. The cache is invalidated, and the remainder of the lines are not updated. In particular, results are reported for a three-level cache hierarchy system where L1 is conventional technology; L2 and L3 are eDRAM memory modules. This eDRAM setup consumes 56% of total energy consumed by a conventional SRAM with a 25% increase in execution time. The Refrint policy consumes only 30% power consumption and 6% performance degradation over the baseline SRAM caches.

Aditya et.al [43] proposed a spacial co-relation based model for eDRAM. The work develops a model of an on-chip eDRAM cache module where retention time represents distribution. The eDRAM cell's retention time relies heavily on its access transistor's V_t, where V_t has well-known spatial correlation qualities. The retention time also exhibits spatial correlation, according to the analysis presented in the proposed work. They derived a mathematical model of the variation and showed the spatial correlation in eDRAM retention time. Mosaic suggested a tiled architecture(a tile is a logical region) where each tile is associated with a different refresh rate. They reported a reduction of Refresh counts up to 20times over the baseline. Also, they have shown a saving of 43% in terms of refresh energy in the Last level cache.

Versatile Refresh [44] proposes concurrent refresh techniques over the different banks while read and write accesses remain undisturbed. The comparison is made over the naive periodic refresh technique. They have suggested two different modules for refresh management in multi-banked eDRAM architecture. The tracking module and Back-pressure module: The "Tracking" module is made up of a few pointers that answer the query of which cells to refresh next. In each time slot, the Tracking module maintains a note of which row needs to be refreshed. The Back-pressure module enforces this criterion by back-pressing memory access whenever necessary. The "Back-pressure" module is a simple bitmap that answers the question of when to back-pressure memory accesses to induce an idle cycle in all banks. The Row Refresh Schedule is determined by the Tracking module. It tries to refresh the rows in a round-robin order during typical operations. When a row is blocked and cannot be refreshed on its turn, a deficit counter for that row's bank is increased.

Zoran et.al [45] proposed a DRAM-based coherence cache for upcoming 10nm SOI FinFETs. They have suggested a dynamic gain cell(structure explained in previous sections) as an alternative to the 6T SRAM cell. The proposed architecture is for 2 level DRAM cache hierarchy. They consider refresh operations depending on the cache block's coherence state. In order to make this possible, they have extended the MESI protocol and made some changes to the operations. If a Modified line expires, it should either be refreshed to retain it in the Modified state or evicted to a lower memory level before being labeled as Invalid; depending on the individual refresh strategy if the line is refreshed while in Exclusive mode, it will remain in Exclusive mode. Otherwise, it will have to be revoked (no eviction is needed since the correct data exist in the lower cache). If a cache line in the Shared state is updated, the state remains Shared. The line must be invalidated if there is no refresh (same as Exclusive). In the case of an Invalid state, the line doesn't hold valid data it would never be refreshed. They have reported a significant savings of 38% in refresh energy with a performance loss of 8%.

This work [46] proposed a reuse-based refreshing policy to reduce overall energy consumption with minimal overhead on the system. In this direction, they have proposed one MRU-Tour replacement policy and another enaw-architecture for saving energy overhead due to refresh operations. In particular proposed architecture reduces energy through bank prediction and then refreshing selective cache blocks depending on the provided conditions. The three variant of refreshing policy: *Alw*, *cond*, and *Adp* reduces refresh energy by 72% in out-of-order systems with 58% of dynamic energy reduction. In *Alw* policy, blocks always get refreshes regardless of the value of the MRUT-bit and the LRU-stack position of the block. In **Cond**, a refresh operation is applied whenever the target block is stored in the MRU bank, or its MRUT-bit is set (multiple MRUTs). Otherwise, the block is marked evicted and early writeback (if applicable). In *Adp* policy, it dynamically adapts both Alw and Cond according to the condition.

The Gain Cell eDRAM has independent Read and write ports; distinct rows can be read and written simultaneously. A sequence of successive refreshing accesses can be pipelined since refreshing needs a read and write operation. Kazimirsky et.al. [47] proposes a memory availability problem by refreshing on demand without interfering with memory access. They are leveraging temporal and local memory idleness, and refreshing occurs synchronously with R/W operations.

The study in reference [48]presents a concurrent eDRAM refreshing regime that he claimed resulted in negligible memory availability degradation. The cache was divided into 16 banks to do this. The authors found 96% to 99% availability in memory for random access(banks with 512 to 128 lines). This approach adds an independent line counter to each bank, which generates the address of the presently updated line. While one or two other banks, defined in a circular manner, are refreshed, and their corresponding counters are increased. Internal R/W ports are used in this Refresh, with the underlying premise that banks be accessed independently. However, we assume that all banks are visited at the same time just for simplifying the cache. The authors did not explain what occurs when a specific bank is contacted repeatedly over an arbitrarily long time, preventing Refresh during that time.

The suggested technique [49] is based on the observation that cache demands of different applications vary considerably both within and between applications. Based on the above, this technique dynamically increases the active cache size of an application, resulting in minimum performance degradation while achieving significant energy savings. In terms of Refresh, this approach just refreshes valid lines in the cache on a regular basis. There are three reasons why it minimizes the number of refreshes. For example, it prevents invalid blocks from being refreshed. Second, the number of active and valid blocks in the cache is reduced. Thirdly, as there are fewer refreshes, and the execution time is reduced, which minimizes the number of periodic refresh events.

Based on the notion of selective Refresh, Patel et.al. [50] propose a low-overhead refresh power reduction mechanism. They made use of the well-known prevalence of zero bits in DRAM data by adding a small amount of redundant storage and circuitry to index the memory blocks that contain a zero value, allowing them to avoid refreshing such blocks. The fact used to save energy is that the cache memory filters the memory references, and secondly, the distribution of zero's and one is inside cache. However, they have not mentioned the percentage of zero's among benchmarks but show the motivational analysis on the distribution of zero clusters. This paper presents two different approaches for analyzing granularity for application, namely horizontal and vertical zero clustering. Here, cluster refers to all the lines in case of horizontal or all the bits in case of cacheline. The first proposed concept of Horizontal Zero Clustering checks for the zero clusters horizontally. A Zero Indicator Bit(ZIB) is set when all the bits are zero; otherwise, reset. Refresh operations are avoided if the ZIB is one. The bits in the cluster will not be read if ZIB stores a one, but they will be read if it stores a zero. If all cluster bits are 0, the Zero Detect Circuit (ZDC) detects this and writes a '1' to the ZIB. Another approach, Vertical Zero Clustering(VZC) goal is to detect and leverage clusters of zeros in columns of the DRAM array. For such that Zero indicator Row bit is set for each row. The dynamic logic-based decoders have a data matrix complexity of 5 to 8%; the overall area overhead is quite negligible. As a result, their method minimizes the number of refresh processes, lowering idle power consumption. According to preliminary findings, refresh procedures have been reduced by 31% on average across all granularities.

The main concept [51] is allocating the correct amount of cache to each application and then shifting the rest of the cache to a low-power state with minimal performance loss. Now, this reduces the cache's active fraction, where the idle portion of the cache does not need to be updated. Thus it does not require leakage energy; this saves both leakage and refresh energy. Furthermore, only valid blocks are refreshed in the active area of the cache, reducing refresh energy. It uses way-based reconfiguration in which it turns off the same number of ways for all the sets throughout the cache. It may trigger some flexibility issues, which can be resolved by turning off the different number ways in each module. The algorithm initially determines if a module is non-LRU by comparing the number of hits at various LRU places. Thus, it calculates the total number of hits up to that LRU level for each LRU position. The LRU position determines the number of methods to stay active.

2.6.2 Work related to Temperature based eDRAM Refresh Architecture

Smart Refresh was proposed by Ghosh and Lee [52] for DRAM chips. Their technique is based on the DRAM access characteristic. When a DRAM row is accessed, it is not necessary to update it until the next refresh interval. A DRAM read operation, in other words, has the same impact on the DRAM row as a refresh operation. Their technique employs refresh counters for all DRAM rows to take advantage of the refresh effect of DRAM access. When a DRAM row is accessed, the refresh counter for that row is reset, because the access refreshes the associated DRAM row. However, the number of refresh counters is equal to the number of DRAM rows, their design has a substantial space overhead. In addition, their proposal ignores the effect of temperature on retention time. As retention time varies with temperature, taking into account the effect of temperature on retention time can significantly minimise the number of refresh processes.

The traditional refresh approach, assumes the worst-case temperature for all eDRAM stacked cache banks, resulting in needlessly frequent refreshes. As the die farthest from the heat spreader has limited heat radiation capabilities, the temperature of each die is expected to rise as the distance from the heat spreader increases. The temperature of each die is influenced by the temperature of adjacent dies. Gong et.al. [53] proposed a unique temperature-aware refresh strategy for 3D stacked eDRAM caches, taking into account the varied retention duration of eDRAM stacked cache banks owing to temperature fluctuations. Thermal sensors are used to detect the temperature of each cache bank. The retention period of eDRAM stacked cache banks is investigated in this research as a function of temperature. Because the temperature of each eDRAM stacked cache bank changes, the retention period for each cache bank may fluctuate. It's worth noting that an eDRAM cell operating at a lower temperature keeps its data for considerably longer without needing to be refreshed.The approach suggested in this work applies a temperature-aware refresh interval to eDRAM LLC based on temperature from thermal sensors. Furthermore, based on the run-time temperature, this technique dynamically alters the refresh interval. As an outcome, the number of refresh processes is significantly reduced. This technique, unlike the smart refresh scheme, uses varied refresh periods based on temporal and spatial temperature variations. As a result, it significantly minimises the number of needless refresh operations in eDRAM stacked caches. Furthermore, because the spatial temperature change between eDRAM rows is small, the suggested concept does not require refresh counters for all eDRAM rows. As a result, it has a significantly reduced area overhead than smart refresh. The approach suggested in this research uses temperature-aware refresh intervals to decrease significant refresh operations in eDRAM stacked caches, it can effectively reduce performance overhead due to refresh.

Elastic Refresh was proposed by Stuecheli et al. [54] for JEDEC DRAM devices. Through postponing refresh operations, they are able to reduce performance overhead caused by refresh. When memory systems have a lot of read/write traffic, refresh operations are postponed until the memory rank is idle or the retention period is approaching. The DRAM chips have a far longer nominal refresh interval (64 ms) than eDRAM stacked caches (100 s), postponing refresh operations allows DRAM chips to handle more read/write operations between refresh operations. However, because eDRAM stacked caches should be updated with a considerably shorter refresh interval than DRAM chips, postponing refresh operations allows eDRAM stacked caches to have just a few read/write operations between refresh operations. As a result, the performance overhead reduction of their approach in eDRAM stacked caches must be negligible.

Liu et al. [55] proposed RAIDR is one of the retention time-based techniques, which makes a group of cells, e.g., retention time bins. We can get DRAM cells with different retention duration at the same temperature through process variations. Based on the retention period of DRAM rows, their approach divides all DRAM rows into different refresh interval bins. These bins perform the Refresh of the corresponding blocks according to the refresh period associated with the bin. Status of the bins maintained in the memory controller with very low storage overhead. Initially, each row is defined with a minimum row's retention time across each cell. Some set of bins is defined in the memory controller, each bin associated with some range of retention times. Bins contain the rows according to the row corresponding retention time of the row. RAIDR contains three operations. First: retention time profiling that profiles retention time of all rows. It measures the retention time of each cell in a row. Second: storing rows into the bins by retention time. For this, they deployed bloom filters, which is a structure that facilitates a compact way of representing sets. Third, the memory controller takes part in issuing refreshes when necessary. However, their approach does not account for temperature, the data contained in the cell may be jeopardised if the temperature rises during execution. This study did not takes temperature into account while calculating the refresh interval.

For JEDEC DRAM devices, Nair et al. [56] suggested Refresh Pausing. Their technique decreases delay overhead due to refresh by halting refresh operations, similar to Elastic Refresh [54], which is based on JEDEC DRAM chips. Obviously, suspending refresh activities should be done with the retention duration in mind in order to minimise data loss. Refresh Pausing is not useful for minimising latency overhead due to refresh in eDRAM stacked caches for these reasons.

2.6.3 Work related to Hybrid eDRAM Refresh Architecture

Pure eDRAM cells do not appear to be suited for implementing first-level data caches, where availability is a key design problem due to frequent refresh operations. Valero et al. [57] proposes a hybrid cell structure, where the word macrocell refers to a suggested memory structure made up of n memory cells, one SRAM cell, and n-1 eDRAM cells, as well as pass transistors that connect static and dynamic cells. Therefore, leakage and area are tackled by design—here eDRAM cells, which give six to eight times the memory of SRAM in the same space. This reduces the leakage by about 75% compared to the baseline.

Lira et.al [58] shows that a well-balanced SRAM / eDRAM NUCA cache may achieve equal performance as a NUCA cache made up entirely of SRAM banks. NUCA on the base mitigates the effect of increased wire delay over the chip access latency. It consists of architecture with multiple small banks distributed throughout the on-chip network, and each can be accessed independently. The hybrid NUCA architecture, where half of the banks are SRAM and the rest half is eDRAM, used the fact that a significant amount of data on LLC are not being re-accessed. Thus, they have proposed heterogeneous bank clustering, which improved the performance by 4%. They have reported area-saving of 15% and power savings of 10%. Accessed data blocks are constantly moved from one SRAM bank to the next. Gradual promotion of data is indeed in SRAM banks. However, it is not applicable for eDRAM banks. Also, NUCA cache eviction is not triggered by an SRAM bank replacement. In fact, an eDRAM bank may be thought of as additional storage for an SRAM bank.

The process variation inside the eDRAM at the time of retention time persists in some disparity. The eDRAM exhibits a non-uniformity in retention time. Thus it follows failure patterns. The work [59] is to manage to reduce side=effects caused by retention time variation. This work proposes a one-cell failing line (OFL) buffer which manages the status of the one-cell failing cache lines. They concentrated their efforts on repairing one-cell failed cache lines in this study. They make use of the well-known fact that retention-time-failing eDRAM cells can still store '0' values since the leaking charge in the eDRAM cell only impacts data' 1'. As a result, failed cells can be thought of as cells with a stuck-at-0 defect, in which we only store '0' data.

2.7 Works exploiting Presence of Zero Valued Data Blocks

Zero value cache [60] reported the presence of 18% Zero Loads in the critical loads. Loads that must complete early to avoid performance degradation are Critical Loads (CLDs). They have proposed a non-speculative architectural design for Out of order system for load scheduling. ZVC keeps track of zero-valued locations compactly and responds quickly to such load requests. The idea on the core level is similar to the Frequent Value Cache, where ZVC is inclined towards the Zero Loads. They have reported improvement of performance by up to 78% with low additional complexity.

In order to increase the cache effectiveness better to go for cache compression. We show that compression can improve performance for memory-intensive commercial workloads by up to 17%. L2 contains compressed and uncompressed data locations. Where each set can store up to eight compressed lines but has space for only four uncompressed lines. Each set is an 8-way set-associative, with a compression information tag stored with each address tag. The data array is broken into eight-byte segments, with 32 segments statically allocated to each cache set. Thus, each set can hold no more than four uncompressed 64-byte lines, and compression can occur at most double the effective capacity. Each line is compressed into between one and eight segments, with eight segments being the uncompressed form. Besides the above information, the compression tag is maintained coherence state information which is used by the adaptive compression policy. Evaluated on the two proposed models, Never models a standard 8-way set associative L2 cache design, where data is never stored compressed, and Always models a decoupled variable-segment cache, but always stores compressible data in compressed form. Limitation for low-miss-rate benchmarks shows performance degradation up to 18% due to unnecessary decompression. As compression helps eliminate long-latency L2 misses, it increases the latency of the (usually more frequent) L2 hits. They have used an adaptive predictor that monitors the actual effectiveness of compression and uses this feedback to dynamically determine whether to store a line in a compressed or uncompressed form(calculates cost and benefits of compression). Thus adaptive scheme successfully predicts workload behavior, provides a performance speedup of up to 26% over an uncompressed cache design for benchmarks that benefit from compression [61].

In this work [62], the concept of Zero indicator Bit (ZIB) represents optimized zero content blocks into one bit. An auxiliary circuitry was implemented, which handles compression with 9% area overhead. On read access and ZIB is set, the local read access is disabled. If ZIB is clear normal Read occur. Similarly, it will write only ZIB when we have to update zero data block on the cacheline. Thus scheme gives benefit in Read and write energy consumption. Our DZC technique allows us to retain the benefit of differential bitlines while taking advantage of the asymmetric distribution of 1's and 0's to reduce energy dissipation. Additional hardware embedded in the RAM array to detect and eliminate the reading and writing of zero bytes. They get 26% energy reduction on data cache accesses and 10% on instructions cache accesses with a minimal area overhead of 9% due to additional circuitry.

The access energy of the STTRAM is dominated by leakage consumed by the Read and write circuits required to sense and flip the bit-cell state. This [63] contributes a significant part to the total energy consumption of the chip. Like the drowsy cache, one of the methods is power-gating, which requires an auxiliary circuit over the chip. They have proposed a normally-off STT-MRAM cache architecture that exploits that leverage zero-byte compression for energy reduction. The gem5 simulation reported the result of 60% reduction in total energy consumption On a read operation, M-bit is checked first; if the value is 0, the full cache line is read and sent to the cache output port. If it is 1, the shifting logic restores the original data, including the zero-bytes, using the saved Z-Tag data.

Dusser et.al. [64] proposed a dedicated hardware Zero Content Cache augmented parallel to the main cache. ZC Cache is in a compressed form where compression just requires a tree of OR gates for detecting a null block, and Decompression does not induce extra access latency. Moreover, it reduces traffic on the main cache and without inducing latency overhead. Here, the zero detectors perform a global OR on the value to be stored in the cache. According to the protocol perspective, ZCA caches enable multiprocessor coherency protocols without any extra coherency state bits. Also, the ZC cache is never the owner of a modified copy of the block. When a local processor writes non-null data to shared memory space, the block must be acquired: if the block was previously present in the ZC cache, it must be invalidated. When a distant processor writes non-null data to shared memory space, the block in the ZCA cache must be invalidated. On a complex memory hierarchy, implementing ZCA caches at every level is possible. Such a hierarchy allows a design optimization that may significantly reduce the miss ratio on the faster cache. According to the sensitivity analysis over the zero blocks, the number of null blocks per valid sector is relatively high for applications featuring a high proportion of accesses to null blocks. The proposed ZCA cache significantly reduces the cache miss rate up to 80% (for a few applications).

According to the Frequent Value Cache proposal [65], frequently used values, especially null blocks, might be represented in an adjunct cache retrieved in tandem with the cache. Overall they have proposed three ideas to retrieve frequent values inside cache blocks. In the first proposal, once identifies a *fixed frequent value* set during a profiling run, which the program subsequently uses in all subsequent execution runs. In the second approach, in each execution run of the program, this procedure finds a *fixed frequent value*. The set of values is discovered using *restricted online profiling* during the program's initial run, after which the values are fixed, and profiling is turned off. The program then uses these values for the remainder of the execution. The third proposal continuously changes or does profiling in the program throughout each execution cycle; this approach keeps a changing frequent value set. Profiling is also performed via specialist hardware. An application can benefit from the adaption of the frequent value specified throughout a run using this technique. These are entirely software-based methods. The address tag and a single validity bit would be used to denote a null block. The address tag would be the principal storage expense in such an auxiliary cache.

Villa et.al. [66] have a detailed analysis over the zero content data in the benchmarks. According to their reporting, somewhere 70% of the bits are zero bits read from or written to the cache. Researchers suggest a differential design for a larger SRAM array, as they are immune to noise and fast sensing data from the cacheline. DZC approach preserves the benefits of differential bitlines while reducing energy dissipation by making use of the asymmetric distribution of 1s and 0s. They have combined many bits and added an additional zero indication bit to show when the entire bit field is zero, rather than handling each bit separately. It gives a flavor of dynamic compression. This could save energy on both write and read accesses as they only write the ZIB rather than the entire bit field. When driving the data link between the CPU and the cache, they also use the ZIB to save energy. In our improvisation technique, we have used a similar concept of Zero indicator bit(ZIB).

The zero awareness techniques can use for SRAM power reduction. Change et.al [67] proposes power reduction technique for '0' bit. They have analyzed the bit level for tracing zero majority among the benchmark programs. They have proposed a zero-aware SRAM cell architecture that reduces power dissipation while writing zero over the cell. This would create asymmetry in writing data over the SRAM cells. The negative aspect of the architecture would be an increase in the area overhead of 8.8%. However, stability and performance remain stagnant throughout the execution.

STTRAM writing contributes a significant amount of energy to total energy consumption. Jung et.al. [68] proposed an idea exploiting the zero data content for energy reduction in STTRAM LLC. It reveals that 68.40% of the bytes and 54.02% of the words written to the L2 cache comprises only zero-valued data, on average. The suggested cache design provides extra *all-zero-data flags* in cache tag arrays at a given granularity, such as a byte or word, to take advantage of the prevalence of zero-valued data for STT-RAM write energy reduction. For 64-byte cache lines, the proposed AZBF cache architecture adds 64 bits to the SRAM tag array in each cache route, but the proposed AZWF cache only adds 16 bits to the tag arrays. Therefore, the access latency of the SRAM tag array with all-zero-data flags at the byte granularity (the all-zero-byte flags: AZBF) is greater than that of the tag array with all-zero-data flags at the word granularity (the all-zero-word flags: AZWF) in the proposed cache. The suggested design's tag arrays have long latencies due to the higher access time of SRAM arrays.

Dynamic Zero Sensitivity scheme proposed by Chang [69] reduces average cache power consumption. They have examined the skewed distribution of the Zero Data bits over the read bits from instructions and the data cache. The DZS method avoids the bitlines from discharging while reading a 0, resulting in significantly less energy consumption when reading a 0 than when reading a 1. The bitline discharge is determined by whether the value stored in the accessible cell is a 0 or not. Both the bitlines should be kept prevented from discharging if the stored value is a 0. Otherwise, a normal read operation will cause one bitline to discharge.

2.8 Existing Work Related to Relaxed STTRAM (Volatile-STTRAM)

2.8.1 Existing work related to Read Disturbance Error in STTRAM

In the context of retention-relaxed STT-RAM, the technique of Sun et al. [70] seeks to schedule the refresh operations when the banks are idle. Their technique does not reduce the number of refresh operations but merely reschedules them. Also, its efficacy is greatly reduced in the case of bursty reads. In order to increase accuracy, they have proposed two designs Shadown reqerite buffer and write bit inverting. They have achieved an improvement of 6.7% in IPC followed by the 34% improvement in the EDP. The fast switching version of STTRAM gives 19% improvement over the conventional-STTRAM version.

Mittal et al. [71] use data compression for reducing RDE. However, the efficacy of their technique depends on the compressibility of data. They have proposed their technique with the name of SHIELD. Different compression algorithms are practical on certain data patterns only, such as pointers or integers. Also, compression/decompression circuits lead to complexity. By contrast, our zero-value-based policy is straightforward to implement. SHIELD reduces restore operations where other techniques try to reschedule and trigger when the cache bank is idle. It performs restore operations immediately and hence, always keeps the L2 cache RDE-free. The main work is based on data-width only and can be integrated with any cache coherence scheme.

The technique of Jiang et al. [72] selectively performs HCRR and "low-current long latency" (LCLL) read based on whether a bank is idle. Their technique is proposed for main memory, where the memory controller buffers read/write requests. Since main memory and cache have substantial differences in their architecture, their policy may have reduced applicability or effectiveness for the caches. They have introduced Smash Read(S-RD) for reducing the latency in HCRR. In addition to the above technique, they have proposed a Flexible Read (F-RD) which dynamically switches techniques to maximize system performance. Flexible Read adaptively issues destructive Smash Reads, or non-destructive LCLL reads to minimize memory bank occupation time.

A previous work detects dead reads using compiler analysis [73] and then avoids the restore operations for such blocks. However, compiler analysis requires ISA modifications to pass this information to the hardware. This may require recompilation, which may not be feasible in many scenarios. Further, compiler analysis works well for register files but not for caches since the caches are hardware-managed.

Hosseini et al. [74] present a compiler-based technique that identifies the vulnerability of load operations to RDEs. Then, it uses a code-transformation scheme for reducing the number of vulnerable loads. Finally, with the help of the compiler, their technique inserts restore operations after the vulnerable loads. They evaluate their technique using SE (syscall) mode with the "TimingSimpleCPU model" for the ARM ISA. They only evaluate single-core applications since compiler-analysis becomes infeasible for out-of-order scheduling and multicore processors. By contrast, we assess a much-more realistic configuration with a 4-core processor with the Ruby memory module in the full-system mode for x86 ISA.

Wang et al. [37] propose a technique for skipping restore operations for blocks that are likely to see a write operation in the future. For performing restore operations in the L2 cache and modifying the metadata in L1 or L2, their policy needs to observe the block state in other (i.e., L2 or L1) caches. Since their policy postpones restores, it may lead to a situation where a dirty L2 block has an RDE issue. Hence, their policy adds complications to cache management. Their approach requires substantial changes for working with other cache coherence techniques, but they do not discuss those details. Neither do they discuss the compatibility with other coherence techniques.

Cheshmikhani et al. [75] study RDE issue in cache tags. They note that many tag reads are due to parallel accesses to the tag, which create RDEs. Their technique avoids reading those tags that are unlikely to lead to a hit based on the LSBs of the tag. By contrast, our technique focuses on RDE in the data array.

Cheshmikhani et al. [36] note that in the modern caches, all the blocks in a cache set are accessed in parallel without checking their ECCs. Then, only the requested block is supplied to the processor after ECC verification. Due to this, in the remaining blocks, the RDEs accumulate. The proposed technique avoids mentioned problem by checking the ECC of the requested block and all the blocks in a cache set that have been read in parallel. However, their technique [14] is applicable only for the caches that perform parallel access to all the cache ways. Generally, the last-level caches use a sequential tag-data access scheme, where in the first step, all the tags are accessed. Then, in the case of a hit, only the requested data block is accessed. Hence, the technique of Cheshmikhani et al. This policy is helpful for first-level caches only.

2.8.2 Existing work related to Multi-Retention STTRAM

Prior work has shown a significant reduction in the processor's power; thus, replacing SRAM with STT-RAM can considerably improve the energy efficiency. A significant reduction in leakage power, reduced the total energy consumption in DRS as compared to the SRAM. In DRS, retention time is kept constant throughout the execution of the system. Technically it performs Refresh, which requires a temporary buffer to hold data during refresh operations. They have proposed Two variants of data retention time decreased STT-RAM: 26.5 s and 3.24 s[76].

In Cache Revive [77], the author used a temporary buffer of 1900 slots. We have scaled down the size of the buffer as per our corresponding configuration. The Cache Revive policy is writeback based, in that beyond a certain number of blocks that get refreshed, most of the expiring blocks get written back to the next level. This results in increased miss-rate and affects the performance. CACHE-REVIVE includes three types of overhead:1) Static overhead due to buffer slots, 2)Migration overhead from STTRAM to buffer and reviving from buffer to STTRAM, 3) Increase in Writeback percentage. Cache-revive performs operations over blocks with elapse time 3/4th retention time interval. If a block belongs to the first half of the MRU slots, a particular block will be migrated to the temporary buffer if space is available, otherwise evicted. If the block is dirty, it will be written back to the main memory. If the block belongs to the other half, in that case, the block is either evicted or written back to the main memory. CACHE-REVIVE policy contains overhead due to early evictions and writeback.

HALLs [78] proposes an iteration-based approach to select the best possible cache configuration and retention time interval, whereas our approach is focused on block placement to reduce refresh energy. The HALLS proposes to refresh the free approach by per-block counter lifetime tracking. If counter lapses, the cache block will be evicted, and if they are dirty, they are first written to the main memory before eviction. We are not performing a comparison with HALLs as the approach focuses specifically on configuration and Retention time selection. Besides that, it is a pure writeback-based strategy rather than refreshing a block. As our proposal is Refresh based and HALLS is writeback-based, we have not quantitatively compared it with HALLS.

LARS [39] proposes an adaptable retention time model for multi-retention STTRAM caches. It depends on the different applications and behaves dynamically. It proposes an optimal tuning algorithm that determines the retention based on energy consumption. The policy proposed by the author gives a significant reduction of 35.8% in total energy and a 13.2% reduction in overall access latency. They have provided results on the basis of sensitivity analysis over different cache configurations for the same policy.

In case the retention time is longer than the duration for which data blocks must remain in the cache. In order to prevent this premature eviction, we can do the initial assignment of freshly loaded blocks.

MirrorCache [79] is a work conducted that deals with the refreshing in the volatile-STTRAM by altering the way. This approach employs two identical cache segments termed the main and auxiliary segments to use the high density of STT-RAM. Instead of transferring data into the refresh buffer when a block expires, the data is transferred into the auxiliary segment, and further access to those blocks is diverted. This eliminates the requirement for an external refresh buffer, as well as the accompanying leakage energy overhead. Due to the obvious significant proportion of refreshes, the energy optimization of this approach is solely based on reduced static energy. However, there is still a potential to minimize the dynamic energy spent in refresh operations, but they have reported a reduction in leakage power.

MH-Cache [80] proposed Hybrid LLC architecture for Mobile Hardware Rendering system. They are using a Low-Retention STTRAM cache with SRAM. The purpose of SRAM is to behave like an absorber for write-intensive applications. Their proposal uses a variable threshold for a process' write-intensity to determine cache line placement. They get a significant reduction of 32% and 32.2% in total energy consumption in single- and quad-core systems, respectively w.r.t normal STTRAM last level cache.

Rabiee et.al [81] proposed a lifetime-altering technique for Low-Retention-Time STTRAM Cell-based L1-cache architecture. It would not be a viable solution to use normal-STTRAM at ULC. In order to choose a suitable working point, they have investigated the trade-off between reliability, write latency, and write power consumption of relaxed STTRAM. They eliminate redundant write operations by considering dead blocks in the L1 cache. They proposed a modified L1 cache coherency protocol which reduces 60% write operations from the different sources. For this, they augmented two states(M' and S') in the MESI protocol. They have reported the lifetime improvement analysis with the performance penalty of STTRAM-based cache. The possible overhead of area and power consumption are considered with the simulation. The proposed architecture also includes an 8-entry buffer. Here, low-retention STTRAM reduces leakage power consumption, which incorporates a total power consumption of up to 57% concerning baseline.

CHAPTER 3

Refresh Optimisation through Dynamic Reconfiguration Based Policy in eDRAM

In this chapter, we discuss the first contribution to the Refresh Management in volatile last level cache. We propose a content based refresh saving method. In particular, certain blocks of data may have zero content and such blocks do not need refreshing. Such zero value blocks are kept in a dedicated partition in the cache and this partition is never refreshed. The size of the zero value partition is also dynamically reconfigured depending on the variation in the number of zero blocks and the miss rate during the execution of the application. These two dynamic reconfiguration policies save a significant number of refreshes over existing policy with minimal overhead.

3.1 Introduction

In a study, eDRAM consumes eight times less leakage power and has high density as it utilises fewer components in comparison with other technologies, thus we can have more capacity of LLC packed in the same area footprint compared to SRAM [82]. Memory manufactured using eDRAM is based on dynamic RAM technology which has the disadvantage of charge loss on its storage capacitors, requiring the memory cells to be periodically refreshed. The duration over which the cells can retain the charge (or data value) is called the retention time of the cell and one needs to refresh the cell within this retention time. The frequency at which we perform this refresh is called the refresh

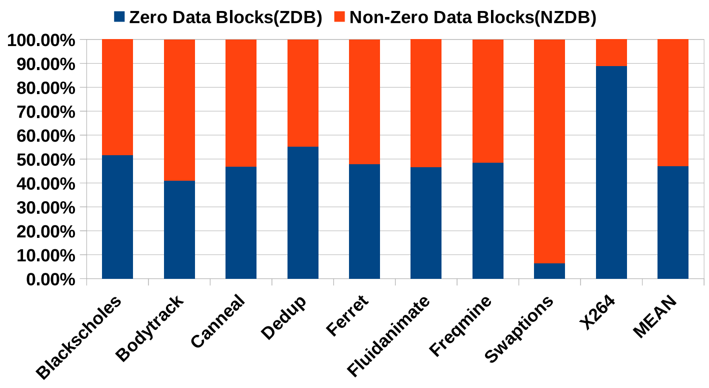


Fig. 3.1 Percentage of Zero Data Blocks and Non-Zero Data blocks for various benchmarks

interval. As the capacity of the eDRAM caches increases, the number of refreshes increase leading to more refresh energy consumption. It is reported that refreshing consumes up to 70% of total energy consumption [44]. It must also be noted that during refresh the bank accesses are stalled and this stall duration depends on the number of cache lines. The stalls will affect the performance and hence optimising refreshes is also necessary from the performance perspective. Therefore, controlling refreshes is the primary research challenge to make eDRAM as replacement candidates over SRAM in LLCs. We primarily focus on the refreshing of the eDRAM cache only.

According to Ekman and Stenstorm [83] 30% of blocks have all 64-bytes as null/zero. We considered the existence of the Zero Data Blocks in the applications that have null or zero content in our previous work. We leveraged the approach of frequent value cache [84] and zero augmented caches [85] that use separate caches to manage data having zero values. These zero data blocks also exhibit high spatial locality and resources are wasted in storing such zero value blocks in the cache. Current existing literature claims our point regarding existence of zero blocks in the benchmarks. [85, 86]. Experiments conducted by us on PARSEC benchmark suite, reports that zero content ranges from 6-80% with an average of 41%.

Thus observing the content of the data blocks, we save on the refresh time and energy. This work is an extension of our previous work where we keep Zero Value Partition Fixed through out the execution.

The main contributions of this chapter are as follows:

- The cache is logically divided into 2 partitions: one for storing blocks with zero valued data & second for non-zero valued data blocks. As the number of zero

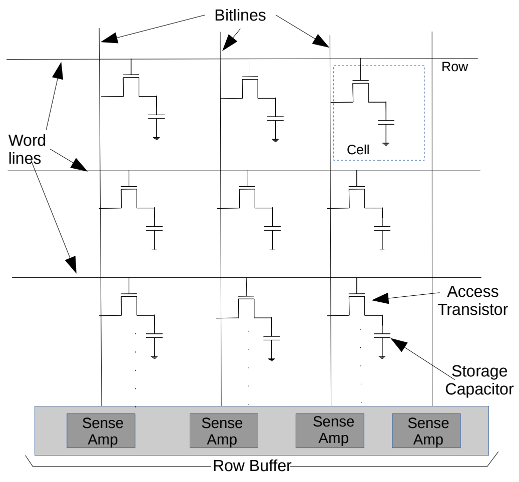


Fig. 3.2 Abstract Diagram of an eDRAM bank

valued blocks change over execution period, we dynamically resize the zero valued partitioned bases on this metric.

- Reserving Zero Partition in LLC will also alter the miss rate, and this may affect performance of the system. We propose a miss rate based dynamic resizing algorithm by taking it as a metric for reconfiguring ZVP size.
- We have proposed two variants of techniques: Fixed and Dynamic version whereas dynamic containing further two different proposals with Dynamic Zero value partition based refreshing.
- Experimental evaluation is done on the full system simulator Gem5 [87], which is integrated with modules to support eDRAM caches. Results show significant improvement in refresh energy savings over the existing technique and the baseline.

The rest of the chapter is organized as follows: Motivation and Background related to baseline works are presented in Section 3.2. The proposed Zero Detection Based Refreshing techniques are reported in Section 3.3. Section 3.5 illustrates the experimental methodology. Results and analysis are discussed in Section3.6. Section 3.7 reports the overhead analysis. Finally, we summarize this chapter in section3.8.

3.2 Motivation and Background

3.2.1 Prevalance of Zero Data Blocks

Figure 3.1 shows the percentage of blocks with zero and non-zero content for various benchmarks. In order to track the overall zero block count, we have kept one Zero Detection Logic(explained in Section 3.3), i.e., XOR-based marking technique. Eventually, we can get zero block from main memory, and upper level cache. We have tracked the freshly loaded block from the main memory and the writeback blocks from the upper-level cache. One can also analyse the number of zero content blocks averaged over different intervals of execution. According to our current experiments over different intervals, we claim that the frequency of the zero content block varies during execution and motivates us to reconfigure Zero Value Partition dynamically. In order to measure dynamic zero data blocks behavior, we have used fixed size interval in cycles(in our experiment we have used intervals of 2 million, 5 million, and 10 million ruby cycles). Tracking among the interval of such zero content blocks gives us many opportunities to save refresh energy. In that, we identify the corresponding metrics for reconfiguring interval and do needful operations on partitions before the block is loaded and categorizing between zero and non-zero. The blocks with zero content are segregated in a region/partition of the eDRAM cache and are not refreshed.

3.2.2 Periodic Round Robin Refresh(PRR)

In baseline, we are using the variant of distributed mode, where different banks are refreshed at different times in a staggered fashion. Accesses only to the banks getting refreshed are stalled and the other banks are available to serve requests. Thus, distributed refreshing helps in controlling performance degradation due to stalls.We use this distributed refresh as our baseline refreshing technique.

The refresh operations are triggered across banks in a staggered round robin fashion. In particular we formulate it as follows. From the given retention time, we compute the global refresh interval, which is nothing but the number of cycles after which each bank needs to be refreshed.

$$\textit{Global refresh interval} = \textit{retention time} \times \textit{clock frequency}$$

Depending on the number of cache banks, we divide this global refresh interval such that the banks trigger refresh in a staggered fashion, as shown in Fig. 3.3. In the

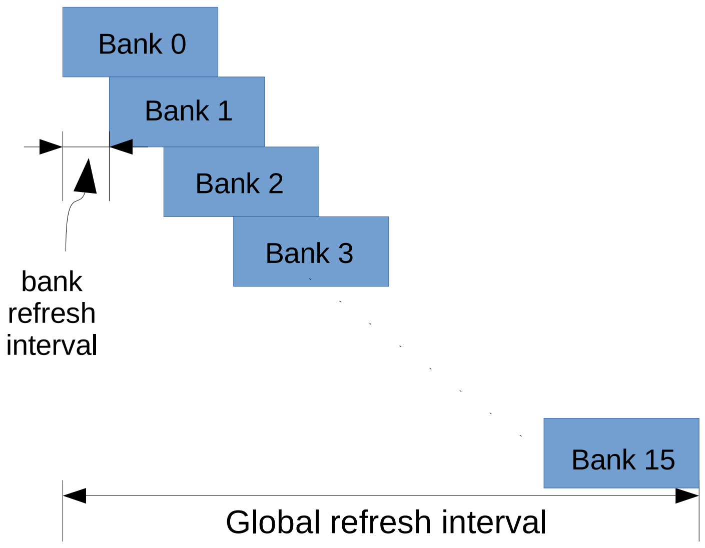


Fig. 3.3

figure, the grey boxes representing the banks indicate that the bank is undergoing refresh operation in that time line. As time proceeds, the other banks get refreshed and at the end of the global refresh interval the first bank again starts its refresh operation. The gap between refreshes of two banks allows the next (and the remaining) banks to serve normal cache accesses. This reduces the one time stall for all banks, as faced by the all bank refresh scheme. Formally, this is called the *bank refresh interval* and is calculated as follows:

$$\textit{Bank refresh interval} = \textit{global refresh interval} \div \textit{number of banks}$$

Note that depending on the global refresh interval and the size of the bank, some banks may get refreshed in an overlapped manner for a small duration.

Whenever a bank is getting refreshed it causes a contention with the normal cache access. Here we have to handle two cases: (i) a normal cache access comes when a bank is being refreshed; and (ii) bank refresh trigger comes when a normal access has come. For the former, the normal cache access is stalled and will be served only after the bank refresh operation has completed. In case the bank refresh trigger comes when a normal cache access is pending, we give priority to bank refresh and hence stall the normal cache access. This access can resume once the refresh operation has completed. As can be noted, the quicker the refresh completes, the faster the stalled requests can be processed. This method of refreshing is used as the baseline architecture in this work.

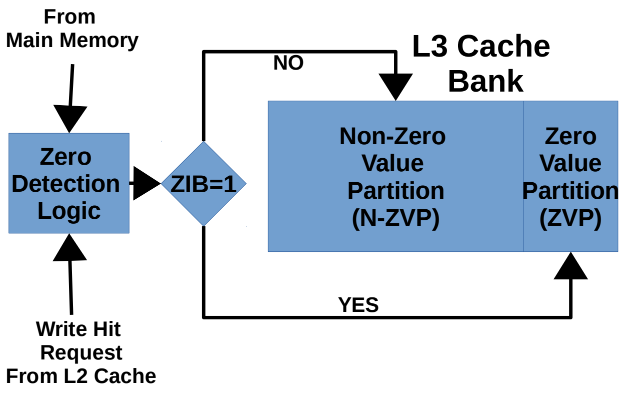

From Main Memory
Zero Detection Logic
Write Hit Request From L2 Cache
ZIB=1
NO
YES
L3 Cache Bank
Non-Zero Value Partition (N-ZVP)
Zero Value Partition (ZVP)

Fig. 3.4 Diagram of Fixed Zero Value Partition Based Refresh Technique

3.3 Proposed Architecture

3.3.1 Zero Detection Logic (ZDL)

We use a logic component that takes as input the data block and indicates whether the contents of the block are zero. We name this logic as the *Zero Detection Logic (ZDL)* and the output of the ZDL is a single bit indicating whether the data block is null. In particular the output bit called the *Zero Indicator Bit (ZIB)* is set when the block has null data and it is reset otherwise. A basic XOR function is used to compute the output.

Single Zero Detection Logic is sufficient to track Zero Data Blocks as we handle one block at a time, whereas we require a Zero Indicator Bit with each block.

Figure 3.4 shows the block diagram of ZDL. Blocks get allocated or written to the LLC when either they are loaded from the main memory or they get written back from a higher level cache. For every block loaded from the main memory, it goes through the ZDL. In case ZIB=1, the block is allocated to Zero Value Partition (ZVP) otherwise it gets allocated to Non-Zero Value Partition (N-ZVP). Similarly, for the blocks that are written back, each block goes through the ZDL. If the written back block has null data and it is allocated in N-ZVP it is migrated to ZVP and vice-versa. Details of the operation are given in the next subsection.

3.3.2 LLC Architecture

Figure 3.5 shows the high level diagram of the proposed eDRAM architecture. The eDRAM banks are shown along with the row and column addresses, the row decoders,

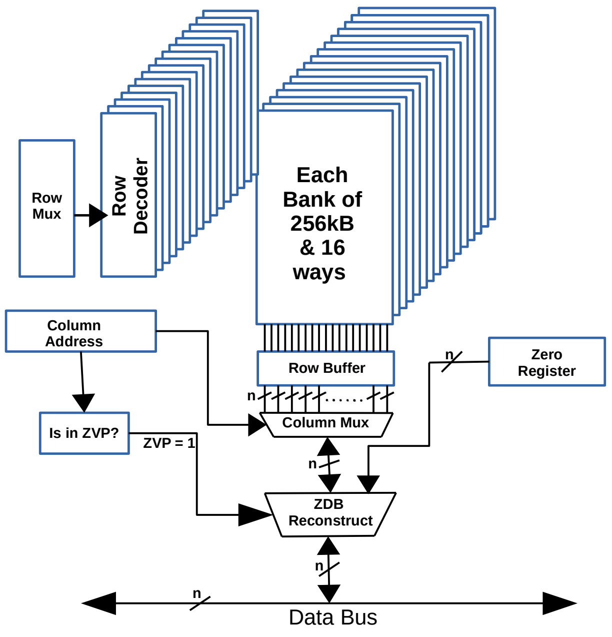


Fig. 3.5 High-level Diagram of the proposed architecture.

the row buffer that contains the data of the enabled row and the column MUX that selects the appropriate block from the row buffer in order to transfer the data on to the data bus. Note that the refresh circuitry is not shown in the figure. The refresh controller is modified such that the part of eDRAM consisting of the ZVP is not issued refresh signals.

For every read request to the eDRAM the row is read into the row buffer and using the column address the corresponding data block is read out. When a block is read from the N-ZVP the data is sent as it is. However, when we need to read a block from ZVP, the data present at the location has arbitrary content as these locations are never refreshed. Therefore, for the blocks read from ZVP we need to reconstruct the data by sending actual null values on that read. For this we maintain an additional register with zero content of the size of a cache line. Whenever a read happens to ZVP, we send the contents of this zero register to the requesting module. This requires us to install an additional 2:1 n-bit MUX at the output of the column-MUX, called the *ZDB Reconstruction MUX*. The column address is used to identify whether the block is in ZVP, and this information is used to control the extra MUX. If block address is in ZVP, the contents of the zero register are sent to the data bus, otherwise the contents from the row buffer are forwarded on to the data bus. All the signals for these units are managed by the cache controller.

Algorithm 1: Fixed Zero Value Partition Based Refreshing

```
repeat
    for each refresh issued for the cache block B do
        if $B \in ZVP$ then
            Do not refresh the block
        else
            Refresh the block
        end
    end
    for each request R coming from L2 to the block B in L3 cache do
        if $R == Read\ Hit$ then
            if $B \in N\text{-}ZVP$ perform the read as in the conventional cache,
            if $B \in ZVP$ reconstruct block and serve response
        else if $R == Write\ hit$ then
            if to be written data block is Zero data block and cache entry at ZVP then
                Write operation is performed on B as in conventional cache
            else if to be written data block is Non Zero data block and cache entry at N-ZVP then
                Write operation is performed on B as in conventional cache
            else if to be written data block is Non Zero data block and cache entry at ZVP then
                Write request is redirected to Non Zero Value partition in the same cache set
            end
        else
            (cache miss)
            forward the request R to main memory to fetch the block
            if fetched block is Zero data block then
                Keep the newly arrived block in Zero Value Partition (ZVP)
            else
                Keep the newly arrived block in Non Zero Value Partition (N-ZVP)
            end
        end
    end
until End of execution
```

3.4 Proposed Methodology

This work containing multiple policies which are as follows:

1. **Fixed Partition:** The main idea is to save refreshes by identifying the data blocks having zero content and segregate them in a separate fixed logical partition of the eDRAM based LLC. The tracking done with the help of zero indicator bit(explained in section 3.3).

2. **Dynamic Partition:** We have proposed one partition reconfiguration algorithms which is extended version of fixed partioning algorithm.

We have discussed the components and architecture required to implement the base. This section contains the previous algorithm and its limitations followed by the proposed dynamic reconfiguration algorithms.

3.4.1 Fixed Partition Base Refreshing

Fixed Partition Based Refreshing where we have fixed sized dedicated partition for keeping Zero data blocks, and refresh operations avoided to this partition. Here, LLC is accessed whenever there is a request for a block *B* from the higher level cache. For each request, the tag search is performed, and depending on the outcome a set of actions take place as dictated by the coherence protocol. Below we discuss all the relevant steps in the context of zero-valued blocks. The zero data blocks with be allocated in the Zero Valued Partition(ZVP). The tags with be managed in the traditional method. For every request on block *B*, the following operations are performed depending on the outcome of the tag comparison. The detailed steps are described in Algorithm 1.

- **Cache miss:** Here the requested block *B* is loaded from the next level of memory. The block passes through the ZDL, and the ZIB is set accordingly. If ZIB=1 then the newly arrived block *B* is allocated to the ZVP by evicting the LRU from the ZVP. If ZIB=0 then *B* gets allocated to N-ZVP by evicting the LRU from that partition (line 25 to 31).

> The eviction via LRU means we are evicting the complete cacheline from the LLC. It would be an early eviction for the cacheline.

- **Read hit:** If $B \in N\text{-}ZVP$ then the read is performed as in a conventional cache (line 10). If $B \in ZVP$ then we need to reconstruct null data before serving the read request. This is because the data in ZVP is not guaranteed to retain the null values as the ZVP is never refreshed. We, therefore, use the content from an additional zero register and send its data in response to the read request(line 15 to 17).
- **Write hit:** Block being written back from higher level caches also pass through the zero detection logic (ZDL) and set/reset the ZIB bit. We have two cases to consider:
  (i) $B \in N\text{-}ZVP \wedge ZIB = 0$ and (ii) $B \in ZVP \wedge ZIB{=}1$: Write is performed as in a conventional cache.(line 19 to 23)
  (iii)$B \in N\text{-}ZVP \wedge ZIB{=}1$: Keep in N-ZVP as migration has overhead.
  (iv) $B \in ZVP \wedge ZIB{=}0$: This is the scenario when there is a confirmation that the data block is modified during program execution.
  For case (iv) we must move the block to N-ZVP because if the block stays in ZVP it will not get refreshed and will lose its data content. To move this block, we

need to identify an empty location in N-ZVP. If such a location is not available, then we evict LRU from N-ZVP and redirect block *B* to N-ZVP. However, there is a confirmation that the data block is modified during program execution. For instance, a zero data block becomes non-zero. In this situation migration of blocks can be done. For case (iii) we do not consider migrations, This migration is not mandatory for the correct operation of the proposed policy as migrations may increase the miss rate as we possibly do early eviction of blocks during migration.

Working example of Fixed ZVP Based Refreshing

Figure 3.6 shows the working example of the proposed idea: Zero Detection Based Refreshing. It shows a sample 16-way associative cache that uses 12-ways (way 0-11) as N-ZVP and remaining 4-ways (way 12-15) For ZVP. For a cache miss, the block is fetched from main memory (shown by arrow-1). Consider, data block B as an input to Zero Detection Logic. For block B, if ZIB = 1, this depicts that B is a Zero Data Block. According to the cache availability, B will be redirected to ZVP (shown by dotted arrow-3), i.e., to the ways 12-15. In case, for Block B if ZIB = 0, this depicts that B is Non- Zero content data block. Hence, it has to be redirected to N-ZVP (shown by filled arrow-2). In case, if there is no invalid location in the corresponding partition, LRU victim is selected (shown by arrow-4) and the request is redirected to the location and block is written in that location. At the time of write-back from the higher-level cache, this block may return as dirty with Non-Zero value (shown by arrow-5). If ZIB = 0 and B belongs to ZVP then cache entry will be migrated and write back will be redirected to new location(shown by dotted arrow-7 and 8). In case, if there is no empty space in the partition in which block has to be migrated then LRU victim is evicted from the corresponding partition (shown by arrow-9). If ZIB = 1 and B belongs to N-ZVP, then cache entry will be written back to the same location. Due to overhead, We will not consider this migration (shown by black arrow-6).

Limitations of Fixed ZVP Based Refreshing

In the above discussion the partition size of ZVP is fixed (i.e. ZVP = 25% of the total cache) Therefore blocks in the Last level cache are forced to get accommodated in the provided ways assigned to them. In particular, for blocks loaded in the LLC the possible locations, i.e. the number of ways are less in compared to those in normal cache due to the partitions. Thus, early evictions of the LRU blocks from cache sets may raise miss

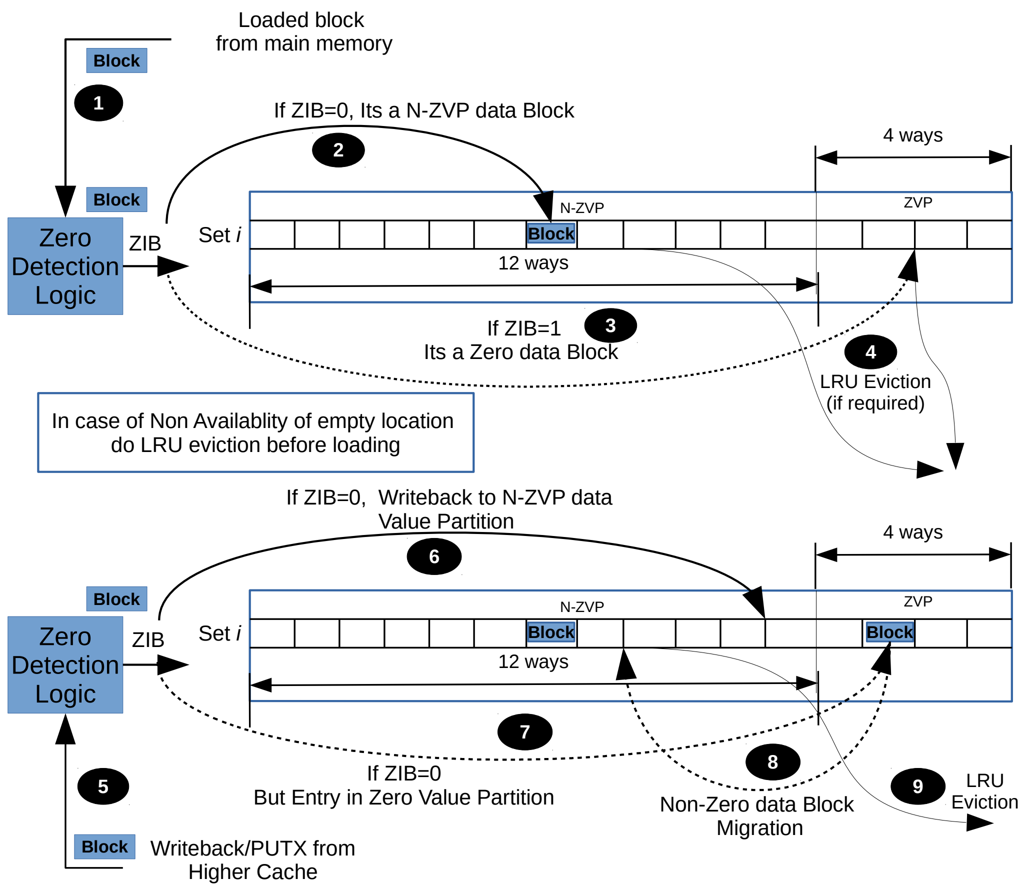


Fig. 3.6 Working Example of Fixed Zero Value Partition Refreshing

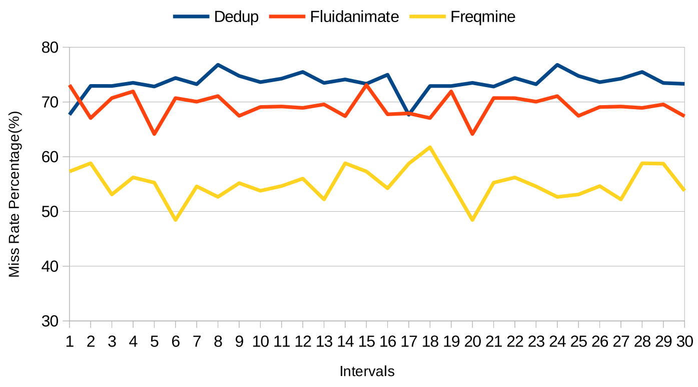


Fig. 3.7 Variation of miss rate during execution. Here Y-axis shows the overall miss rate for the application during execution after a specific interval. X-axis represents different intervals.

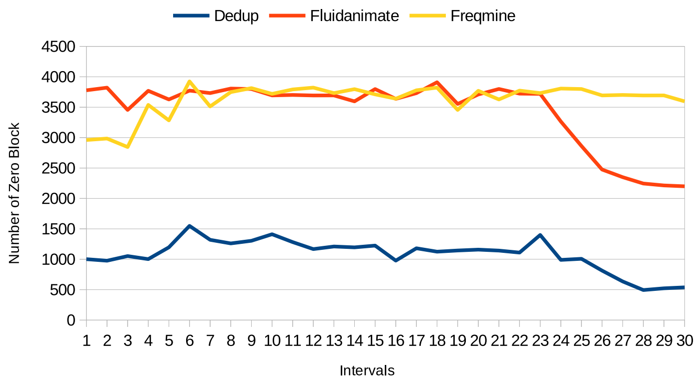


Fig. 3.8 Variation of Zero Block Count during execution

rate and degrade system performance. Average miss rate fluctuates with an average of 10%.

Figure 3.7 shows fluctuations in Miss rate during execution at end of specific intervals. Our hypothesis is that the smaller size of the N-ZVP increases miss rate & that we can use this measure to resize the partition. Another observation is that the number of zero blocks vary across the intervals as shown in Figure 3.8. If we can track these then the information can be used to resize the partition. Hence, we need some dynamic decision algorithm in order to vary zero value partition size.

Algorithm 2: Zero_Data_Block_Count_Algorithm( )

```
1  $T$ : Predefined reconfiguration interval
2  $T_{normal}$ : Warm up interval treating the cache as normally available cache
3  $ZDB_{prev}$ : Zero Data Blocks in previous interval
4  $ZDB_{cur}$ : Zero Data Blocks in current interval
5  $NZDB_{prev}$ : Non-Zero Data Blocks in previous interval
6  $NZDB_{cur}$ : Non-Zero Data Blocks in current interval
7  $ZVP_{cur}$: Zero Value Partition Current Size
8  $ZVP_{min}$: Minimum ZVP Size to be maintained
9  $ZVP_{max}$: ZVP Size limit of Maximum expansion
10 Run system for $T_{Normal}$ clock cycles (cc).
11 for Every Reconfiguration interval T do
12     if $NZDB_{cur} - NZDB_{prev} > 0$ and $ZDB_{cur} - ZDB_{prev} > 0$ then
13         if $NZDB_{cur} - NZDB_{prev} > ZDB_{cur} - ZDB_{prev}$ then
14             if $ZVP_{cur} <= ZVP_{max}$ then
15                 Expand the size of ZVP
16             end
17         else
18             if $ZVP_{cur} > ZVP_{min}$ then
19                 Decrease the size of ZVP
20             end
21         end
22     else if $NZDB_{cur} - NZDB_{prev} > 0$ and $ZDB_{cur} - ZDB_{prev} > 0$ then
23         if $ZVP_{cur} <= ZVP_{max}$ then
24             Expand the size of ZVP
25         end
26     else if $NZDB_{cur} - NZDB_{prev} > 0$ and $ZDB_{cur} - ZDB_{prev} > 0$ then
27         if $ZVP_{cur} > ZVP_{min}$ then
28             Decrease the size of ZVP
29         end
30     else if $ZDB_{cur} - ZDB_{prev} == ZDB_{cur} - ZDB_{prev}$ then
31         Do not alter the size of ZVP
32     end
33     $ZDB_{prev} = ZDB_{cur}$
34     $NZDB_{prev} = NZDB_{cur}$
35 end
36 Use the new value of the ZVP and N-ZVP during the next interval by running
     Algorithm- 1
```

3.4.2 Reconfiguration of Zero Value Partition based on number of Zero Blocks

We examine the direct factor which affects system performance. During execution, Zero data block loading in LLC varies its acceleration at different intervals. We make a history-based decision while altering the Zero Partition size. Below we discuss all the relevant operations to be done before altering partition size. The detailed steps are described in Algorithm 2. Execution starts, the system will run for some initial clock cycles called as warm-up cycles (T_{normal} cycles) and during this period no changes will be made (line number 10). ZVP_{min} refers to the minimum ZVP Size to be maintained throughout the execution e.g. 25% of the cache size. To keep track of number of zero blocks we use two counters: ZDB_{prev} = stores the value for Zero data block count in stipulated predefined interval and ZDB_{cur} = Zero Data block count for the current interval. Similarly, we maintain 2 counters $NZDB_{prev}$ and $NZDB_{cur}$ in order to maintain count of non-zero data blocks. On completion of warming up, the application will run for another T cycles (line number 10). We consider the following cases at the end of each reconfiguration interval:

- **When NZDB increases and ZDB decreases:** The Non-Zero data block count increased and Zero data block count decreased w.r.t previous interval. With an assumption that, this may affect the system performance, we balance by reducing the ZVP size by one provided it is more than minimum limit ZVP_{min}. Where ZVP_{cur} refers to the present volume of ZVP and ZVP_{min} refers to the minimum size limit to which we can decrease.
- **When NZDB decreases, and ZDB increases:** Similarly, just opposite to the former case, Zero data block count increased, and Non-Zero data block count decreased w.r.t previous interval. To balance the performance, we increase the ZVP size by one provided it is less than maximum limit ZVP_{max}. Where ZVP_{max} refers to the maximum size limit of ZVP to which we can increase.
- **When NZDB increases and ZDB increases :** When both NZDB and ZDB counts increased over the last stipulated predefined interval, then priority will be given to the category having maximum variation.
  - **When NZDB increases more comparison to ZDB:** Here priority will be given to the N-ZVP & ZVP size is decrease.
  - **When ZDB increases more in comparison to NZDB:** Here priority will be given to the ZVP & size of ZVP is increase.

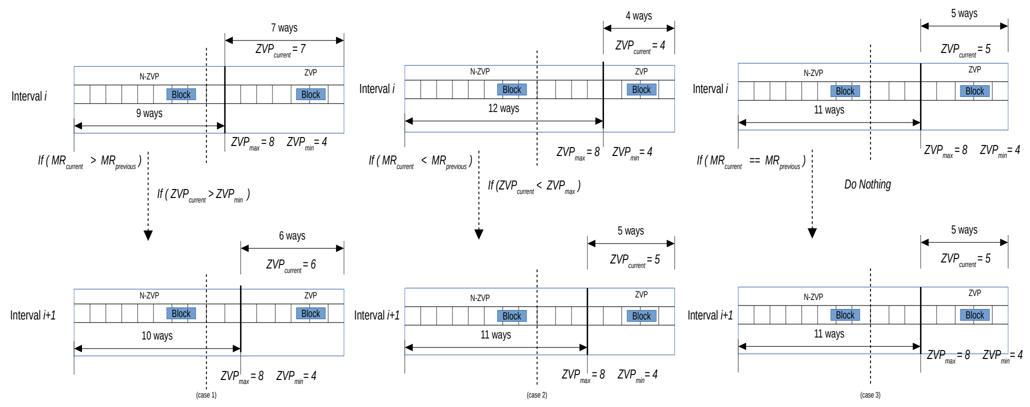


Fig. 3.9 Working example of Miss Rate Based Reconfiguration with different cases.

- **When NZDB decreases and ZDB decreases in previous interval:** This situation does not require any changes.

In the end, all the counters associated with N-ZVP and ZVP ways are reset for the next reconfiguration interval. Current values will be assigned to Previous value counters for the next upcoming interval. The algorithm is repeated until the end of execution. Using the new value of the ZVP and N-ZVP over the next interval, we run Algorithm- 1.

ToRefresh bit

In order to maintain the altered size of ZVP, we need to keep track of blocks into extended ZVP region (i.e. way-8 to way-11). In particular, we set a bit for the blocks written on way-8 to way-11 which tells whether the block has to be refresh or not; and is termed as $ToRefresh$ bit. As we can see in Figure 3.10 and Figure 3.11, The recent extended ZVP region may contain a NZDB, so $ToRefresh$ bit of each set for the corresponding way is checked, and in case it is set, the refresh to that block is considered otherwise skipped. Note that within a few cycles the $ToRefresh$ bit will not be required once the block gets evicted from the ZVP. We will set this $ToRefresh$ bit for ways 8-11 only explicitly by looking ZIB bit in Algorithm 1. 25% ways are fixed to ZVP & these are never refreshed. Hence they do not have $ToRefresh$ bit. the middle range ways-8 to way-11 need $ToRefresh$ bit as they may belong to ZVP or N-ZVP dynamic at run-time.

3.4.3 Reconfiguration of Zero Value Partition Based on Miss Rate

To balance the affected miss rate due to the early evictions of blocks during the execution, we make a history-based decision while keeping the block during migration. Below we discuss all the relevant calculations to be done before altering partition size. The detailed steps are described in Algorithm 3. Execution starts, the system will run for some initial clock cycles called as warm-up cycles (T_{normal} cycles) and during this period no changes will be made (line number 7). $MR_{previous}$ stores the value for previous Miss Rate and $MR_{current}$ calculated for the current interval. On completion of warming up, the application will run for another T cycles (line number 9). At the end of each reconfiguration interval we have:

Algorithm 3: Miss_Rate_Based_Algorithm( )

```
1  $T$ : Reconfiguration interval
2  $T_{normal}$ : Warm up interval treating the cache as normally available cache
3  $MR_{previous}$ : Missrate of previous interval
4  $MR_{current}$ : Missrate of current interval
5  $ZVP_{current}$: Zero Value Partition Current Size
6  $ZVP_{min}$: Minimum ZVP Size to be maintain
7  $ZVP_{max}$: ZVP Size limit of Maximum expansion
8  Run system for $T_{Normal}$ clock cycles (cc).
9  for Every Reconfiguration interval T do
10     if $MR_{current} < MR_{previous}$ and $ZVP_{current} <= ZVP_{max}$ then
11         Expand the size of ZVP
12     else if $MR_{current} > MR_{previous}$ and $ZVP_{current} > ZVP_{min}$ then
13         Decrease the size of ZVP
14     else if $MR_{current} == MR_{previous}$ then
15         Do not alter the size of ZVP
16     The $ZVP_{min}$ is the minimum size of ZVP that is maintained for refresh
         reduction. end
17     $MR_{previous} = MR_{current}$
18 end
19 Use the new value of the ZVP and N-ZVP during the next interval by running
     Algorithm- 1
```

- **On Miss Rate decrease:** Decrease in the miss rate indicates that the system is stable with current ZVP size & thus we can try to increase $ZVP_{current}$ by one to give more space to ZVP & save refresh energy.

- **On Miss Rate increase:** With an assumption that an increase in the miss rate going to affect system performance, we will decrease $ZVP_{current}$ by one. This is the value of the number of ways reserved for Zero Value partition.
- **case Miss Rate stable:** This situation does not require any changes in the $ZVP_{current}$ value as miss rate is not fluctuating.

The ZVP_{min} is the minimum size of ZVP that is maintained for refresh reduction. ZVP_{max} is the value of a maximum number of ways reserved for Zero Value partition which indicates a limit on its size. In the end, all the counters associated with N-ZVP and ZVP ways are reset for the next reconfiguration interval. The current value of Miss Rate will be assigned to Previous value counters for the next upcoming interval. Run algorithm 1 for the next interval. The procedure is repeated until the end of execution.

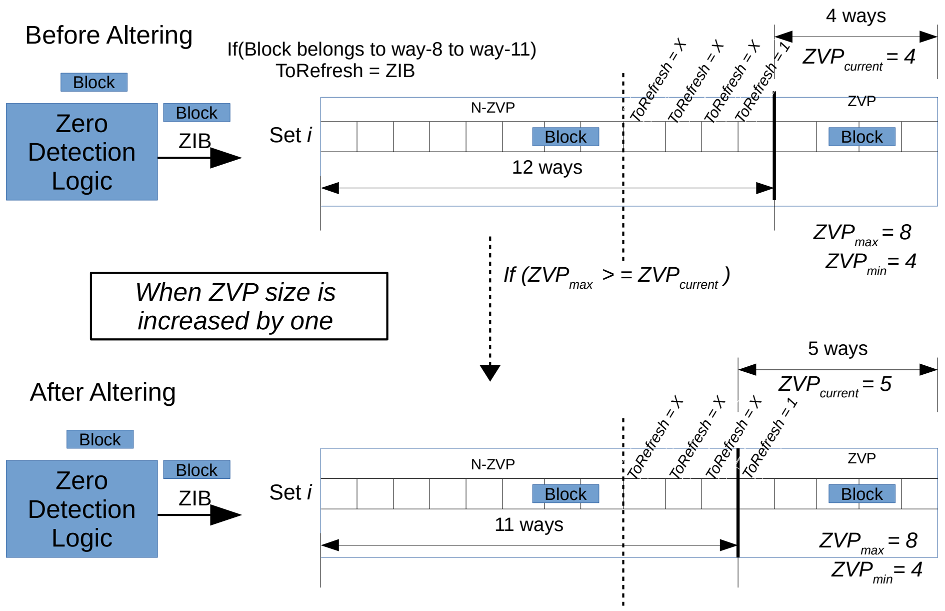


Fig. 3.10 Reconfiguration of ZVP when size increases

3.4.4 Illustration of ZVP resizing

Figure 3.10 and Figure 3.11 show the working example for the resizing of Zero Value Partition. The applications are run for T normal cycles treating the Default ZVP Size as a minimally available region for Zero Blocks in the cache. At the beginning of each interval T, the reconfiguration of ZVP takes place. Figure 3.10 shows an increase of ZVP region on the basis of the conditions provided in Algorithms 2 and 3. $ZVP_{current}$ stores the current value of ZVP region which is to be updated by one during reconfiguration,

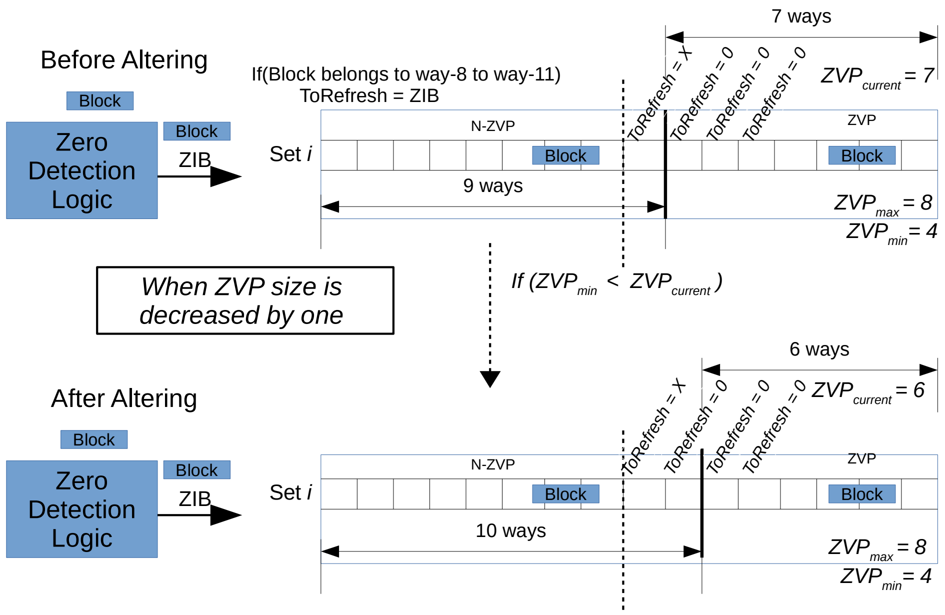


Fig. 3.11 Altering of ZVP when size decreases

provided the value is not greater than ZVP_{max}. An extended region from N-ZVP region may contain NZDB. In order to refresh those NZDB, a $ToRefresh$ bit attached with the corresponding way is considered. The $ToRefresh$ is set by the ZDL when block check for Zero Data. $ToRefresh$=1 depicts that block is to be refreshed until it is not evicted by LRU otherwise it is not refreshed. We decrement the ZVP size (denoted by figure 3.11) by one when performance is lost according to the conditions in the resizing algorithms. In case of no performance loss, where fluctuation in between the intervals is zero or negligible, we do not vary ZVP and keep configuration unchanged. Figure 3.9 shows the working examples for different cases of the proposed policy Miss Rate Based Dynamic ZVP resizing.

3.5 Experimental Methodology

We evaluate our proposal on cycle-accurate full system simulator GEM5 [87] Which comprise ruby memory module for cache implementation. The MESI CMP based cache coherence protocol is used to implement the proposed refresh optimization proposal. Table 3.1 shows the architecture parameters used in our setup. In particular, L1 and L2 are private caches and an eDRAM based L3 as the shared last level cache. The LLC has 16 banks, and we perform a block level refresh at each bank. The connections of LLC with the four cores are maintained over an interconnection network. The baseline refresh

Table 3.1 Architectural Parameters

Components	**Parameters**
Processor	3Ghz, Quad-core, X86
L1 Cache	Private SRAM based 32KB 8 way set-associative, split I/D caches, 64Byte block, 1 cycle access latency, LRU, write back policy
L2 Cache	Private SRAM based 256KB 8 way set-associative, 64Byte block, 8 cycle access latency, LRU, write back policy
L3 Cache	Shared, eDRAM based 4MB, 16 banks(each bank of 256KB) 16-way associative (12ways N-ZVP and 4ways ZVP), 64 Byte block, LRU, 15 cycle access latency [88], 15 cycle refresh latency for one cacheline, $40\mu S$ retention time
Protocol	MESI CMP Directory

method (as given in Section 2.2) uses a time staggered refresh timing for all 16 banks so that all the banks are not stalled at the same time. If a single bank LLC is considered that will hamper the parallel access to the LLC which can cause performance degradation. To show the variation, we experiment with two different values of retention time, namely $40\mu s$ [89] and $50\mu s$ [90]. Currently our design is based on UCA architecture, however this is also applicable to NUCA design. [91, 92]

DESTINY [93] is used to model the wire delays, cache area, energy, and power values for eDRAM based cache. The energy values are given in Table 2. The inter-bank refresh trigger value is computed such that within the retention time all banks are refreshed in a staggered fashion. Tag-in-SRAM method is used for keeping the tag in order to get faster access during tag matching. So, we do not need any additional refreshing for tag array. The leakage power of an eDRAM cell is 1/8th of that of an SRAM cell. Also, for an eDRAM, the time and energy to refresh a line are almost equal to the time required to access the particular cache line. Applications from multi-threaded PARSEC [94] benchmark suite with medium input sizes having emerging applications are used to validate the proposed architecture. We executed the benchmarks over three variations of reconfiguration intervals of T: 2 million, 5 million and 10 million and

T_{Normal} at 5 million ruby cycles for all of our simulations. The initial study on these benchmarks verified the presence of Zero Data Blocks.
ZVP is dynamically decided by the Miss rate variation or number of zero block at the beginning of the reconfiguration interval. We compared our dynamic policies with our prior implemented existing technique of fixed Zero Data Block Detection policy where ZVP is fixed. We have also compared our proposed technique with an existing Refrint Polyphase Refresh Policy. The Refrint method tries to save refreshes based on the observation, that upon an access to a block, the block gets refreshed; and such blocks need not be refreshed until the complete refresh interval starting from the time of access. For this they divide the refresh interval across phases and allocate the blocks in these phases depending on their access time. Some frequently accessed blocks get refreshed based only their accesses and do not go through the routine refresh cycle.
Following policies are shown in the results:

- Periodic Round Robin Refreshing (Baseline)
- Refrint Poly Phase Refreshing (RPP)
- Zero Data Detection Based Refreshing (Fixed_ZVP)
- Dynamic Reconfiguration of ZVP based on Zero Block (ZB_ZVP)
- Dynamic Reconfiguration of ZVP based on Miss Rate (MR_ZVP)

3.6 Results and Analysis

We compared our proposed policies Zero Block Count Based policy and Miss Rate Based Policy with existing eDRAM baseline and with the previously proposed technique of Fixed ZVP based refreshing. We also compared our techniques with Refrint, which is one of the traditional refreshing policy in eDRAM Caches. We implemented RPP over our Baseline technique We present the results on different metrics: Total Number of Refresh Count Savings, Energy Savings and Performance. We also present results on different variations of our proposed technique. These variations include a different size of eDRAM LLC, different retention interval and different reconfiguration intervals of proposed policies. For the optimality of the result, We fixed the interval size of 2million instructions according to our dynamic policy behavior. Besides that a significant amount of reduction in total energy can be reported over different intervals in both the policies.

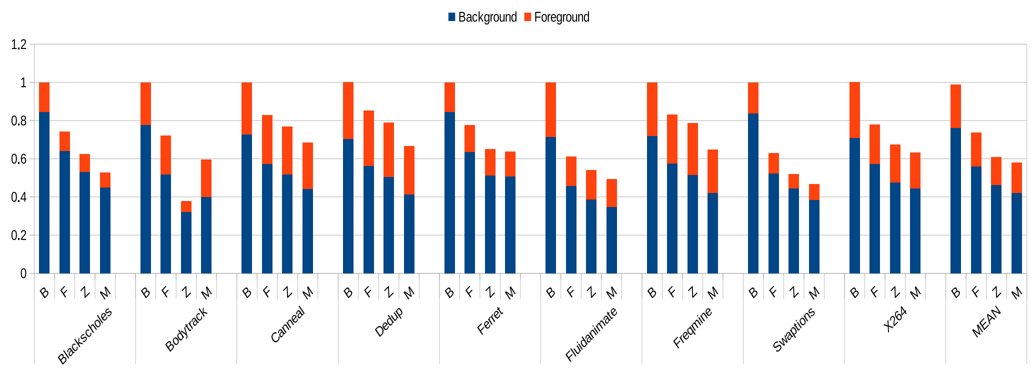


Fig. 3.12 Normalized Refresh Count for 4MB LLC 40μs

3.6.1 Refresh Counts

Figure 3.12 represents normalized refresh counts with respect to baseline(B) policy. All the eDRAM banks get periodically refreshed. A refresh is a high priority task; it always get preference over normal cache access. In particular, when a request comes to access the LLC and the bank is Currently undergoing a refresh, the request is stalled. We categorize the refreshes into two types: (i) foreground refresh occurs when the current cache accesses are stalled due to the bank getting refreshed; (ii) background refresh occurs when the bank is not accessed while it gets refreshed, i.e., there are no pending cache accesses to be served. In both cases, the LLC consumes refresh energy. In the case of foreground refreshes the accesses are stalled, and it affects the performance of the system, whereas background refreshes do not affect performance. According to our experimental evaluation dominance of zeros is not equal in all the applications always. So, our previous policies, implemented with fixed size ZVP have limitations where we have a limited amount of space to keep Zero Data Block. Present policies ZB_ZVP (Z) and MR_ZVP (M) are dynamic improvisations on Fixed ZVP.

Figure 3.12 show the normalised values of foreground and background refreshes for 4MB LLC. In particular, the break-up of foreground and background refresh can be seen with different colors. In the case of ZB_ZVP dynamic policy(Z), Reduction in foreground refreshes on an average of 33%. For background refreshes the reduction is in the range of 29-60% with an average of 40%. In the case of MR_ZVP dynamic policy(M) we get a reduction in refreshes by 35% in foreground and 45% in background. In total We get 38% and 42% reduction in total refreshes for ZB_ZVP dynamic policy and MR_ZVP policy, respectively. Previously when we are reserving 25% of ways as ZVP, we get 29% savings in refresh counts. This is mainly because, reduction in refresh counts completes the execution of application earlier in our proposed method compared

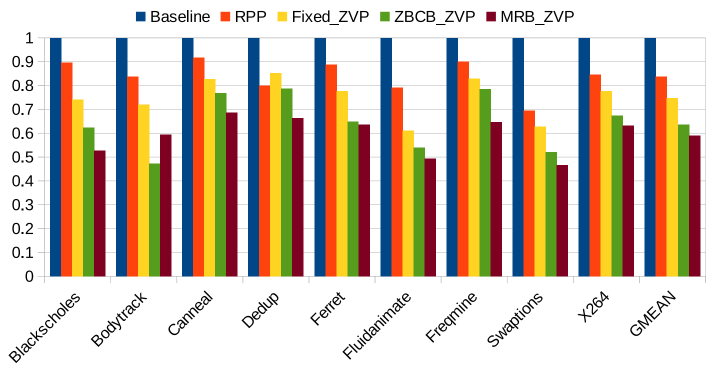


Fig. 3.13 Normalized Refresh Count for 4MB LLC proposed policies w.r.t baseline and Existing RPP policy with 40μs retention time

to the baseline, which would have in turn, incurred one or more periodic refresh cycles. We get a good saving in case of bodytrack as it shows a substantial amount of data sharing. So, the fluctuations due to the miss rate are very low as benchmark taking benefit of data sharing in last level cache. Due to which we can see, the total number of refreshes in miss-rate based policies is more. ZB_ZVP performing more aggressive in changing reconfiguring partition size of the region in comparison to MR_ZVB. Currently, we get an advantage of the fluctuating Zero Block counts and miss rate during execution because Zero Data Block are not fixed in all the benchmarks during execution. Thus, our dynamic resizing gives additional 13% reduction in refreshes. A detailed breakup of bank-wise reduction in refresh counts is shown in figure 3.24. As can be seen, we get considerable reductions in every bank.

3.6.2 Comparison with Other techniques

Agrawal et al.[42] proposed a set of techniques for optimizing refresh energy over the eDRAM caches. Refrint suggests timing based policy (Refrint Poly Phase Policy) as well as data-based policies(Writeback based policy). Data based policies are more aggressive in evicting data block from the cache in comparison to timing based policies. Hence, Data based policies may spike cache misses in LLC and affect system performance directly. So, We implemented RPP on our baseline distributed refreshing technique for comparison with our proposed policies i.e., ZB_ZVP and MRB_ZVP. Figure 3.13 shows the results of 4MB LLC with refresh period of 40μs with baseline. We implemented existing policy over baseline in order to show the advantage of our proposed policies w.r.t existing Refrint Poly Phase Policy. We used 4 phases in a single retention time in RPP policy as it is an optimal value which gives a significant amount of savings. We can

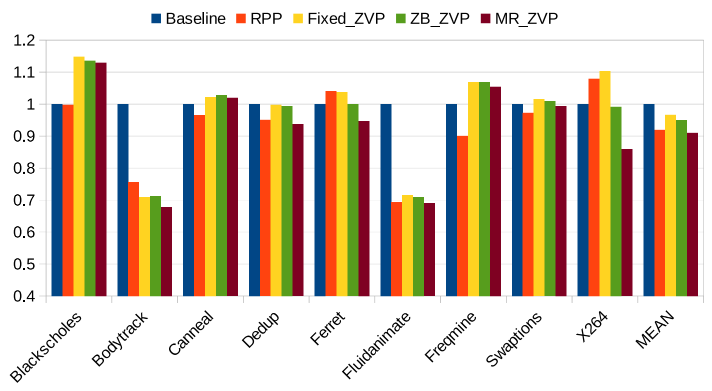


Fig. 3.14 Normalized CPI of proposed policies for 4MB LLC w.r.t baseline and existing RPP policy with 40μs retention time

see Fixed_ZVP policy can save refreshes more in comparison to existing RPP policy, as it is restricting refresh operation for Zero data block whereas RPP refreshes entire cache content and saves refreshes only on frequently accessed blocks.

3.6.3 Performance

Figure 3.14 shows the normalised cycle per instructions (CPI) with respect to baseline. Cache accesses are stalled while the cache bank gets refreshed. As we get considerable savings in the number of refreshes, the stall duration of each bank is reduced significantly. The early evictions of blocks among N-ZVP and ZVP marginally increases the miss-rate in our previous proposed work. This may affect the performance as well as energy consumption. Therefore, even if the miss-rate has increased, the reduction in stall cycles due to saving in refreshes gives an improvement in the performance concerning the baseline. Figure 3.14 shows the normalised performance with an average improvement of 6% in ZB_ZVP dynamic policy and 9% in MR_ZVP policy. On average existing policy RPP provides 8% savings over the baseline. Some spikes of performance loss informed in the case of blackscholes in Fixed_ZVP policy. This exceptional behavior of the performance degradation is due to the early evictions of the blocks from the corresponding partitions while doing migrations from N-ZVP region to ZVP region and vice versa. If we consider the proposed techniques, we are getting improvement and significantly less performance degradation in comparison with Fixed_ZVP.

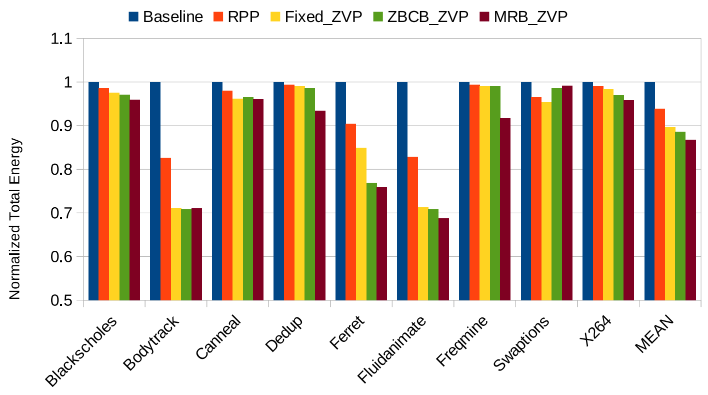


Fig. 3.15 Total energy consumption of 4MB LLC of proposed policies w.r.t baseline with 40μs retention time

3.6.4 Energy Consumption

Reduction in the number of refreshes reduces the energy consumed by the refresh mechanism. We get 38% reduction in refresh energy in ZB_ZVP and 42% reduction in MR_ZVP over baseline, as shown in figure 3.16. Besides that we get an additional 14% improvement over the Fixed_ZVP. Figure 3.16 shows a comparison between our proposed policy and previously proposed technique RPP. Refrint does not substantially reduce refreshes over the eDRAM caches due to the overhead of refreshing unwanted zero block content, where Our proposed policy consider it as of priority and gives an additionally 10% more saving in terms of refresh energy.

LLC consumes energy in the form of static and dynamic, where the dynamic energy includes the refresh energy. Saving refresh counts saves in dynamic energy compared to baseline resulting in savings in total energy. Figure 3.15 shows the normalised total energy consumption. We get 12% and 14% reduction in total energy for ZB_ZVP dynamic policy and MR_ZVP policy over baseline, respectively.

3.6.5 Reconfiguration Interval Based Analysis

Figure 3.17 and Figure 3.18 shows the experiments over various reconfiguration interval of 2 million, 5 million and 10 million instructions. We have fixed the ZVP region size in our previous policy of the Fixed Size ZVP technique. In the case of dynamic policies, we compared the results with the fixed ZVP based policy. For 2million interval size, we get a reduction of up to 15% in the total number of refreshes in ZB_ZVP and 22% reduction in MR_ZVP with respect to Fixed_ZVP. If we consider the Zero Block count as a measure, we can see optimal results in case of 5 million intervals. In the case of

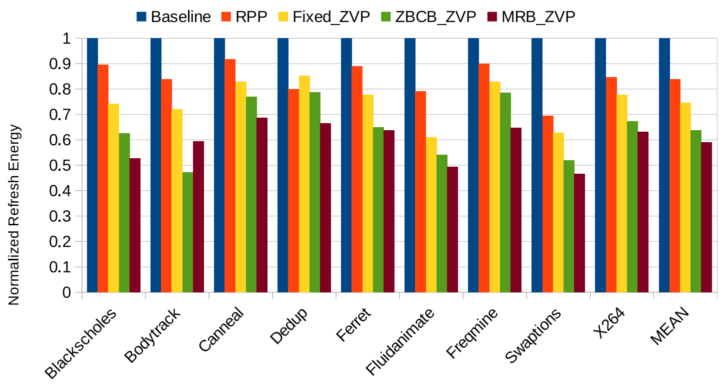


Fig. 3.16 Normalised Refresh Energy for 4MB LLC w.r.t baseline and existing RPP policy with 40μs

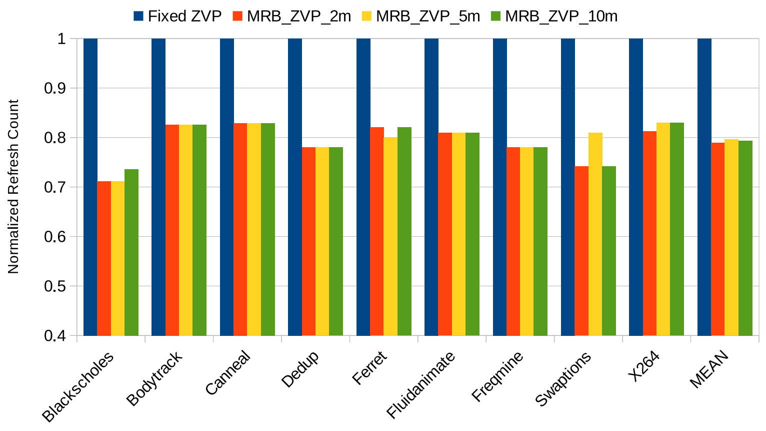


Fig. 3.17 Normalized Refresh Count for different interval for 4MB LLC in Miss Rate Based Policy

ZB_ZVP, the smaller interval gives very less possibility of altering ZVP which results in fewer savings in terms of refresh counts, as where in the case of MR_ZVP, we get more miss rate fluctuations in smaller intervals, and we can see a slight degradation in refresh counts higher for reconfiguration intervals.

3.6.6 Analysis on smaller Cache Size

Towards showing the smaller cache adaptability, we further simulate our policy with 2MB L3 cache and significantly save the total number of foreground refreshes 21%, and a total number of background refreshes 46%, with an average reduction in a total refresh of around 40% for ZB_ZVP policy. The reductions further improved in case of MR_ZVP dynamic policy, where we get 21% reduction in foreground refreshes and 51%

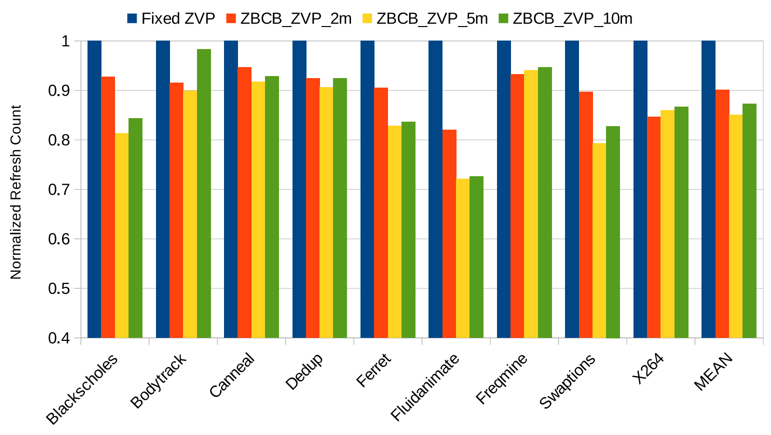


Fig. 3.18 Normalized Refresh Count for different interval for 4MB LLC in Zero Value Data Block Count Based Policy

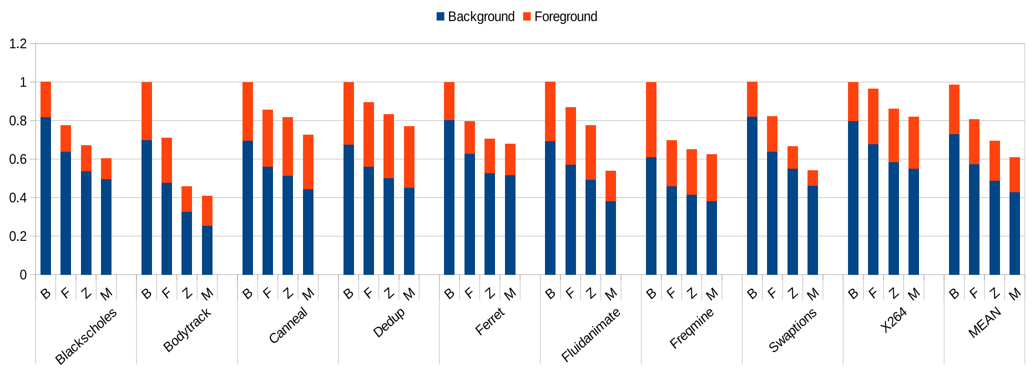


Fig. 3.19 Normalized Refresh Count for 4MB LLC with 50μs retention time

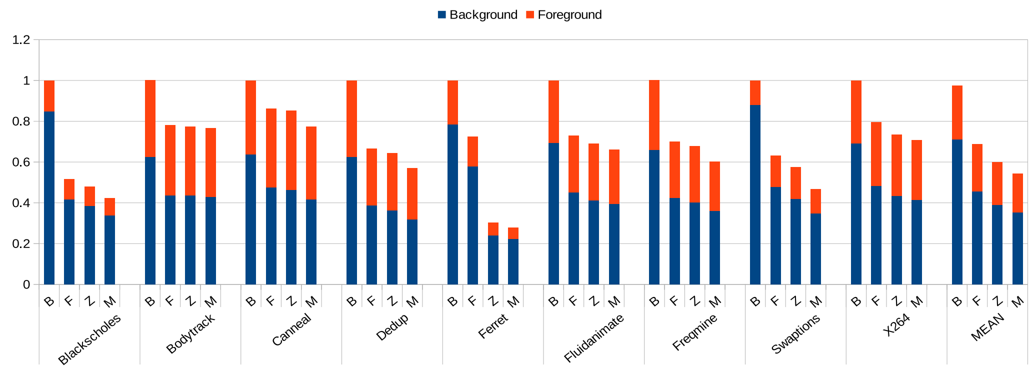


Fig. 3.20 Normalized Refresh Count for 2MB LLC at 40μs

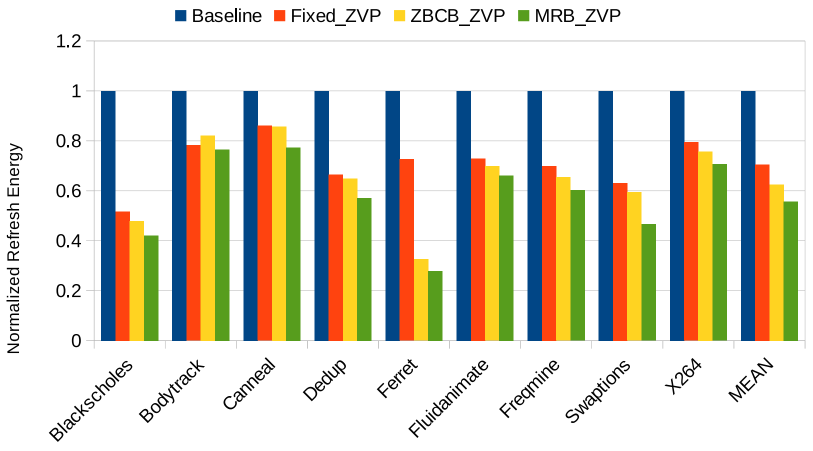


Fig. 3.21 Normalised Refresh Energy for 2MB LLC w.r.t baseline with 40μs

reduction in background refreshes with an average reduction of 45% in total number refreshes. Figure 3.20 shows the foreground and background refresh for 2MB L3 cache 40μs split with different colors. The results for refresh energy savings for a 2MB cache 40μs are shown in Figures 3.21. We get a reduction of 45% in refresh energy over baseline.

3.6.7 Analysis based on Retention Time

We tested our proposed policies with another retention time of 50μs. Figure 3.19 shows the normalized refresh counts with foreground and background refresh break-ups. The overall reduction degraded up to 8% in compared to 40μs eDRAM L3 cache. In case of 4MB with 40μs retention time we get a reduction of 38% in case of ZB_ZVP policy and for 50μs we get 30% savings. We get a reduction of 42% in total refresh count

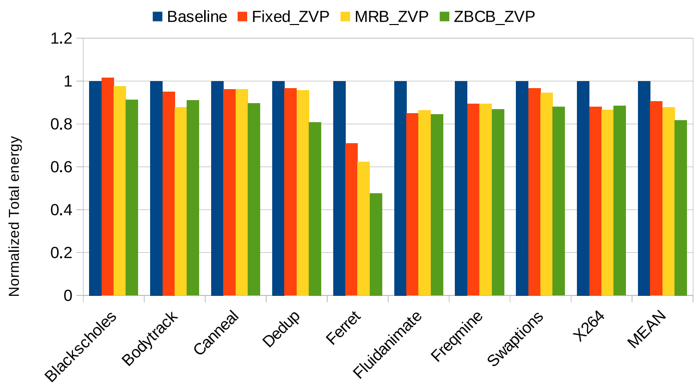


Fig. 3.22 Normalized Total Energy for 2MB LLC at 40μs w.r.t baseline

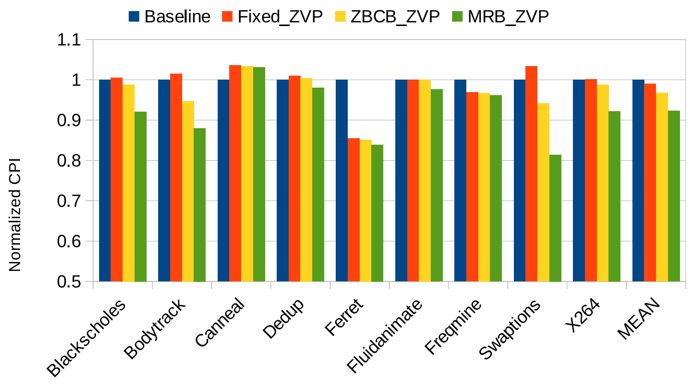


Fig. 3.23 Normalized CPI for 4MB LLC with 50μs Retention time

in MR_ZVP policy for 40μs and for 50μs 38% reduction. We can conclude easily that our policy adapts well to low retention times. This is because of the fact that for smaller refresh period, a larger fraction of energy is spent in the form of refresh energy and hence, the benefits provided by any refresh energy optimising technique are also increased. Figure 3.23 shows the Cycle per instructions graph where we can see the saving degraded by half in comparison with 40μs eDRAM LLC.

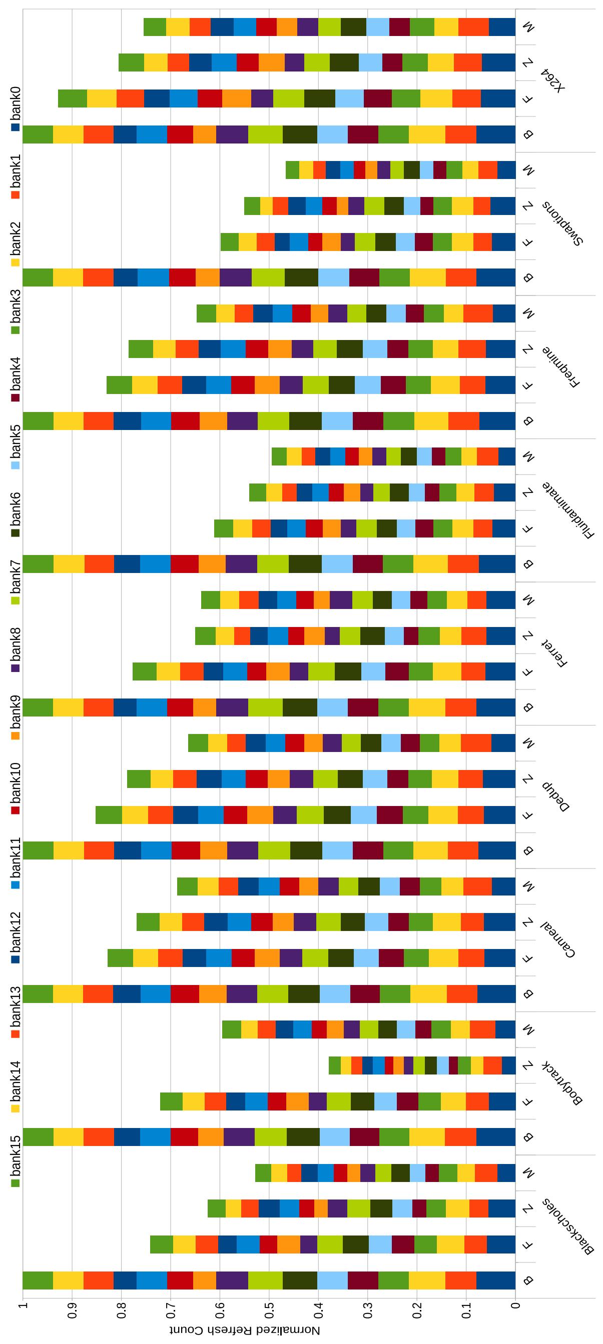


Fig. 3.24 Bankwise normalised Refresh Counts for Fixed_ZVP (F), ZB_ZVP (Z) and MR_ZVP (M) w.r.t baseline (B).

3.7 Overhead Analysis

The additional hardware blocks required for the design are (i) one zero detector circuit for the n-bit data block, (ii) an n-bit register (with zero value), (iii) one 2:1 n-bit MUX that sends output to the data-bus. It chooses between the output of the column-mux or zero-register, (iv) one swap buffer of 42 bits used to migrate the tag from one partition to another, and (v) one $ToRefresh$ bit one each for way-8 to way-11 for each set in all banks. For 4MB, Total number of $ToRefresh$ bits used *256sets $\times$ 4 bits $\times$ 16 banks=16384*. So for 4MB, we required 2KB as storage overhead in the form of $ToRefresh$ bit. Therefore, the total storage overhead associated with $ToRefresh$ bit=$\frac{2KB}{4MB}\times 100\%=0.048828\%$. In this the n-bit data is equal to the block size, i.e. 64B in our case. The miss penalty for the LLC is very high and therefore the extra time required for the zero detection logic is negligible compared to the main memory access time. There is no auxiliary energy overhead for reconfiguration as the normal access is not stalled anywhere during execution. These few logic units add negligible overhead to the overall system design.

3.8 Summary

Chip multiprocessors with multiple processing units require larger on-chip caches to commensurate the memory demands. The shared last level cache has the highest capacity and occupies significant area on the chip as well as consumes considerable power. Caches made up of SRAM can be replaced by energy efficient technologies like NVM or eDRAM. However, caches made of eDRAM require frequent refreshes that consume energy and time. To make them good replacement candidates, we must optimise on the refresh aspect. Towards achieving this goal, the paper presented methods which check the contents of the cache blocks and segregate them into two separate logical partitions. When the blocks are loaded from the memory, those blocks having zero value are kept in a zero partition (ZVP) and non-zero blocks are kept in the other partition (N-ZVP). For the blocks getting written back from the higher level caches, we check their new data content and assign them to the appropriate partition. During the refresh, the blocks residing in the ZVP are not refreshed thus saving on the refresh time and energy. For the blocks being read from the ZVP, we reconstruct the block with the zero data content before sending to the requesting core. It has been observed that the number of zero content blocks vary during the execution. If one can track this number and accordingly adjust the size of ZVP we can further enhance the refresh savings. Towards this we propose a policy (ZB-ZVP) that dynamically changes the size of ZVP.

The number of ways available to non-zero blocks are restricted due to the presence of ZVP. This might affect the miss rate. To consider the impact of this miss rate, we propose a policy (MR-ZVP) that observes the changes in miss rate and reconfigures the size of ZVP. Both the policies aim to reconfigure such that the impact on performance is minimised and the savings in refreshes is maximised.

The proposed policy gives significant savings in the number of refreshes as well as refresh energy. Also, we did a comparison of our proposed work with existing Refrint Poly Phase (RPP) refreshing policy. Compared to baseline, the RPP provides savings of 17% in total refresh count where our proposed policy save more. In particular, the ZB-ZVP policy reduces total refreshes by 38% and the MR-ZVP policy reduces refreshes by 43%. Note that the fixed size zero value policy gives 29% refresh savings. Savings in refreshes reduces the stall cycles, leading to a performance improvement of 6-9%; and reduces energy consumption by 12-14% in the proposed policies.

Carefully managed refreshes can help in making eDRAM as viable replacement of SRAM based on-chip caches.

CHAPTER 4

Coherence based Refresh Optimization in eDRAM caches

Cache coherence protocols in shared memory multicores ensure that shared data is consistent or not. We can reduce the amount of refresh operations by utilising protocol level transitions to discover path to avoid unneccessary refresh triggered by refresh controller. Blocks that requested for writing will get modified in the upper level cache and the copies of these blocks in the last level cache will remain stale. Such stale copies in the LLC need not be refreshed as they will eventually get updated. In this chapter we use coherence messages to flag those blocks which are private in Upper Level Cache (ULC) and therefore do not need refreshing.

4.1 Introduction

In the existing literature, several policies to optimize refresh operations in eDRAM have been proposed. But as per our knowledge, no previous technique uses the concept of private blocks to minimize the refresh operations. Private blocks are those blocks that are cached by a single core with exclusive access permission [95]. To decide, in shared memory systems, cache coherence protocols assist us in determining the type of block requests. We check particular request coming from the upper level cache(ULC), and fetched data from main memory. Shared blocks are blocks that have been cached by several cores and have read-only permission. With shared blocks, the block in the Upper-Level Cache (ULC) contains the same data as the block in the LLC throughout its existence in cache. However, in the case of a private block, the owner's copy of the block may contain data that differs from that of LLC. In other words, the LLC copy

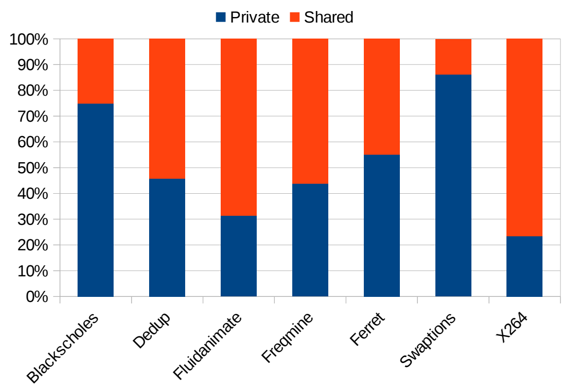


Fig. 4.1 Percentage of private and shared blocks for various benchmarks

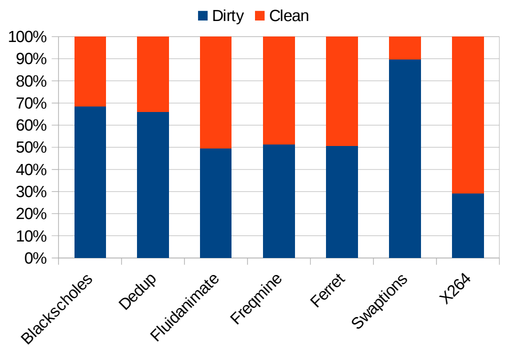


Fig. 4.2 Percentage of clean and dirty status of the LLC blocks at the time of replacement.

may contain old or useless data, whereas the ULC copy is updated and fresh. When there is a write-back from ULC or when the block is replaced solely from the ULC, the stale data at LLC is updated. To measure the impact of private block and its content, we presented an analysis in Figure 4.1 and Figure 4.2 (Please refer to the experimental setup in section 4.4.1). Figure 4.1 shows the percentage of private and shared blocks in the LLC and Figure 4.2 shows the percentage of the private blocks containing dirty data at the time of replacement. The conclusion that can be derived from these figures is that on an average, 51% of the blocks are private blocks and among these blocks 58% of the time this exclusive data was dirty at the time of replacement. Both these facts motivate us to identify the private blocks and propose a technique that avoids the refreshing these blocks in eDRAM based LLC.

4.2 Background

One of the major observations done over the different benchmarks and literature survey, that we have a considerable amount of data blocks with zero bits as content. Ekman and

Stenstorm [83] have found that for their workloads, 30% of blocks have all 64-bytes as zero. In particular, if the data value is zero, the refresh is not required. We can reduce the number of refresh operations of blocks, if we can detect such blocks and maintain track of their contents. The section 3.2 shows the percentage of zero-blocks present in cache blocks when loaded from the main memory and writeback. Towards designing methods to overcome refresh overhead and its impact on performance and energy, we have made following contributions:

- We identified the blocks privately cached by the higher level cache and avoid refreshing of these blocks in the eDRAM based LLC.
- To identify private block and to minimize the refreshes, we make some changes in the conventional MESI coherence protocol.
- In order to improvise our proposed policy, we augment our policy with zero data detection logic and avoid refresh for particular data blocks.
- Experimental evaluation is done on a full system simulator Gem5[87] which is integrated with modules to support eDRAM caches. Results show significant improvement in refresh energy savings and the overall performance.

4.2.1 Private Blocks

As previously indicated, this study considers the presence of private blocks. We have reported presence of private blocks in 4MB eDRAM LLC. On the LLC miss, Fig. 4.1 illustrates the fraction of private and shared blocks loaded from main memory. We may conclude that certain percentage of the blocks are requested by a single core, i.e., private blocks, while the remaining blocks are either instruction blocks or shared blocks requested by several cores. In addition, Figure 4.2 illustrates the percentage of blocks with exclusive authorization (i.e., private blocks) that have dirty data when they are replaced, and almost half of these blocks have dirty data when they are replaced. This demonstrates that at some point throughout their lifespan, the blocks loaded with exclusive rights in the LLC contain useless or outdated data. This motivates us to detect private blocks and avoid periodic refreshing their data in the volatile eDRAM LLC. After the write-back operation from L2 cache to L3 cache, the real worthwhile data are present in the cacheline entries (i.e., the LLC). After writeback, the cachelines would be considered for periodic refreshing until they are not evicted from LLC.

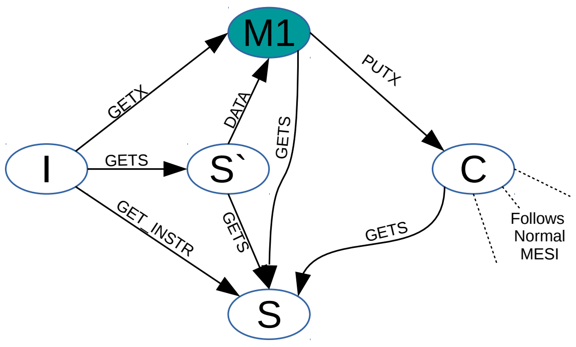


Fig. 4.3 State Diagram showing transitions of states in eDRAM according to the MESI protocol

4.2.2 Changes to the MESI protocol

Figure 4.3 shows the state diagram of the new states added to the protocol to handle the situations arising from the non-refreshed private blocks. The remaining states of MESI (not covered by the figure) behave as in a normal MESI protocol. Below we describe each state and actions associated with them. A request/response reaching to the LLC can belong to the categories mentioned in table 4.1.

On an LLC miss, the block is brought from the next level before sending it to the higher level (L2 in our case). According to our proposal, we have to identify private blocks and manage the to-refresh bit accordingly. Subsequent requests to such blocks and actions taken are discussed in this subsection. First, we list all the relevant states of the state diagram.

- I: Invalid block entry of L3 Cache.
- S′: This state acts as an intermediate state in the state diagram. The transition from this state happens when data block is loaded from main memory on GETS. If in between loading of the block into the cache another request meddles for the same block, it will be served as a shared block. Otherwise, if no other request issued by L2 for this block it will be served with exclusive permission to L2
- S: Block is in the shared state in L3 with read permission. This state is considered for the refresh operation
- M1: Block is held in exclusive permission by the L2 cache. The copy present in the L3 cache is stale, i.e. out of dated as the block suppose to modified in the upper level cache.

- C: Data block written back from L2. L3 has a fresh copy of data block. Considered for the refresh operation.

4.3 Proposed Methodology

4.3.1 Private Block based Optimised Refresh (PBOR)

This section describes the proposed policy, Private Block based Optimised Refresh (PBOR), that reduces the number of refreshes based on the concept of private blocks. In particular, the work proposes that the cache blocks that are privately cached by higher level caches will not be refreshed in the eDRAM based LLC. When the block is loaded from the main memory for the first time, depending on its category it will be identified as a private block or a prospective shared block. If the block is marked as private, the cache entry will include an extra bit to reflect the status. In particular, this bit tells whether the block has to be refresh or not; and is termed as *to-refresh*. This bit is set to 1 by default for each block, and it is reset to 0 when a block is designated private.

At the end of retention time, the cache bank is refreshed. During this, the to-refresh bit of each block is checked, and in case it is reset, the refresh to that block is skipped. Note that this saves a lot of refreshes, thus leading to lesser stall time for normal cache accesses, improving performance and saving refresh energy. This facility comes at the cost of some additional consideration during normal cache access.

Consider the situation when the bit=0 and a new request comes for this block to the LLC. In this case, the block in LLC might have invalid data as it was not refreshed regularly. Therefore the data for this block has to be fetched from the higher level cache. The entry in the directory is used to identify the owner of the block and it is used subsequently to provide the data to the requester. This requires changes to the MESI protocol(as explained in section 4.2.2).

Table 4.1 Table of request and responses

L2 Request or Response	Description
GETS	Read Request
GETX	Write Request
GET_INSTR	Instruction Request
PUTS	Clean block Replacement
DATA	Data from Main memory

Handling response from the main memory on a miss

The description of transitions of states according to L2 request or main memory response are given below:

- Initially all blocks are in invalid state in the cache. A block also becomes invalid after a valid data item is evicted from it. In this case the to-refresh bit is 0.
- If the LLC receives a request to load an instruction, GET_INSTR, it loads the block from the main memory, sends the block to the requesting L2 cache and makes the state of the block as S. Blocks having state S have the to-refresh bit set to 1.
- Consider the request GETS, which indicates that the requester wants the block in read-only mode. Here, the block is likely to be loaded in private mode if there are no sharers. We therefore change the state to an intermediate state S$'$. While the block is loaded from the memory, in case there is an intervening GETS request from another L2, the next state becomes S, as we have more than one sharer for the block. However, if there is no intervening request, then the block is given to the requester with exclusive permission. Here the state changes to the new state M1. This state indicates a block in a private/exclusive mode. Such a block is likely to have stale data in the LLC and hence need not be refreshed. The to-refresh bit for this block in state M1 is set to 0.
- Consider the request GETX, which indicates that the requester wants the block with exclusive permission to modify the content. Here the block is clearly a private block and hence the state of the block becomes M1, having to-refresh bit=0.

Changes in the states of the block when the block is either replaced from L2 cache or if there is a new request or response from L2 cache

The description of transitions of states are given below:

- Consider the case when the block is in state M1 and another L2 cache requests access to this block by issuing a GETS. When the LLC receives the request from the new L2 cache, the LLC fetches the block from the owner and forwards it to the requesting L2. The block is now shared by multiple caches and the state becomes S. This state has the to-refresh bit equal to 1.
- Consider the case when the block is in state M1 in the L2 cache and the cache evicts the block. The owner issues a PUTX request and sends the block to the LLC.

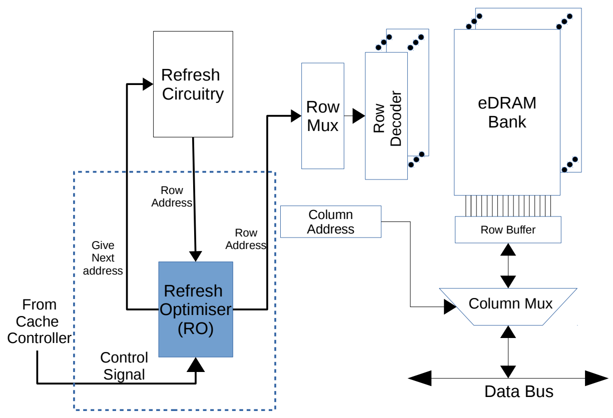


Fig. 4.4 Proposed eDRAM Architecture

LLC updates its copy and the directory entry of the owner for the cache block is also deleted. The block is now present only in the LLC and its state becomes C. Note that, as this is the only copy of the block, we have to keep it valid by refreshing it. Therefore the to-refresh bit is 1. When the block is in state C, it handles all the future requests as per the original MESI protocol.

4.3.2 PBOR with Skipping Zero(PBOR_SZ)

Based on this information, we have performed an augmentation over PBOR; Skipping zero over PBOR policy. Our next proposal is to skip the refresh operations on the Zero-Data Blocks (ZDB). Whenever a block having zero content is read, we do not refresh its contents. The blocks get flagged by the ZIB bit at the time of allocation and writeback. In order to detect ZDB, we need to check the tag array. This saves both read energy and the need to restore the block. In this way, the zero-detection approach saves both access time and dynamic energy. If the data content is not zero, then a read operation happens normally, and the block is refreshed periodically. We further propose another optimization, where a writeback to a block with zero data content will not be written to the LLC. This optimization would further reduce the number of writes performed in the LLC.

4.3.3 LLC Architecture

Figure 4.4 shows the high-level diagram of the proposed eDRAM architecture. The eDRAM banks are shown along with the row and column addresses, the row decoders, the row buffer that contains the data of the enabled row and the column MUX that selects the appropriate block from the row buffer in order to transfer the data on to the data bus. The dotted rectangle shows the new components added to the architecture. It mainly consists of a logic block *Refresh Optimiser (RO)* that receives inputs from the cache controller. The cache controller detects private blocks based on the states as per the changes in the MESI protocol and communicates the information to the RO unit. The RO unit maintains the information of addresses and the associated to-refresh bit for every address. During the periodic refresh, the refresh circuitry generates the address to be refreshed and sends it to RO. As we need to avoid refreshes of the private blocks the address generated by the refresh circuitry is controlled by the RO-unit. The RO unit gets the address from the refresh circuitry and checks the corresponding to-refresh bit. The following cases are considered:

- If to-refresh=1, then the row address is forwarded to the row decoder so that the address gets refreshed.
- If to-refresh=0, then the address is skipped and a signal '*get next address*' is sent to the refresh circuitry to get the next address to be refreshed.

Augmentation for PBOR_SZ: Here, the LLC tag array is associated with one zero indicator bit (ZIB) to indicate whether the data block has zero content or not. If this bit is set, then the block has zero value content. We use a comparator unit called the "zero detection logic" (ZDL) to detect the data value. Whenever a block is loaded in the LLC from the main memory or written back from the ULC, it is passed through the ZDL. If the block has a zero value, then the block is not written in the LLC, but the ZIB is set. When a block is read from the LLC, which has the ZIB set, we do not perform an actual read operation on the data block from the LLC. Instead, we use a zero block reconstruction logic (in particular, a register storing zero value) to send a block with zero content. Note that the ZIB bit information is maintained with the tag array.

4.3.4 Limitations of Proposed Policies

In the above discussion, improvisation policy is mainly dependent on the existence of Zero Data blocks in the application. As shown in Table 5.1, the zero data traffic is mainly divided across blocks loaded from main memory and blocks written back from ULC (column-4). This percentage distribution of blocks leads to savings in write energy for our optimization; Factually, it will directly impact the refresh savings. In fact, among all the zero data content blocks, PBOR_SZ saves periodic refresh operation,as well as read-write energy on such blocks. The percentage read-hits to ZDBs are shown in

Table 4.2 Architectural Parameters and Benchmark Details

Components	**Parameters**
Processor	3Ghz, Quad-core, X86
L1 Cache	Private SRAM based 32KB 8 way set-associative, split I/D caches, 64B block, 1 cycle access latency, LRU, write back policy
L2 Cache	Private SRAM based 256KB 8 way set-associative, 64B block, 8 cycle access latency, LRU, write back policy
L3 Cache	Shared, eDRAM based 4MB, 16 banks(each bank of 256KB)16 way associative, 64 Byte line size, LRU, 15 cycle access latency, 15 cycle refresh latency(one cache line), 40μs retention time

column-5 of Table 5.1, and this percent age value is much lower than the percentage of total ZDBs (column-2). Note that PBOR_SZ reduces the number of writes on ZDBs; however, the refresh operations are avoided periodically to the ZDBs until they are not evicted from LLC. Therefore, augmenting skipping zero policy with our previous proposal (PBOR) is highly advantageous.

4.4 Experimental Evaluation

4.4.1 Experimental Setup

We evaluate our proposal on a full system simulator GEM5[87]. The memory module is simulated using the Ruby module inside GEM5. The MESI CMP based cache coherence protocol is used and modified to implement the proposed refresh optimization proposal. Table 4.2 shows the architecture parameters used in our setup. We use 4 threads in a quad-core setup where L1 and L2 as private caches and an eDRAM based L3 as the last level cache. The LLC has 16 banks and we perform a block level refresh at each bank. The proposal uses a time staggered refresh timing for all 16 banks so that all the banks are not stalled at the same time. The retention time is taken as 40μs [89]. CACTI [96] is used to model the wire delays, cache area, energy and power values for SRAM based cache. The inter-bank refresh trigger value is computed such that within the retention time all banks are refreshed in a staggered fashion. PARSEC [94] benchmarks with medium size input are used to evaluate our proposed policy. These benchmarks include a large variety of multi-threaded applications such as mining, multi-media, animation,

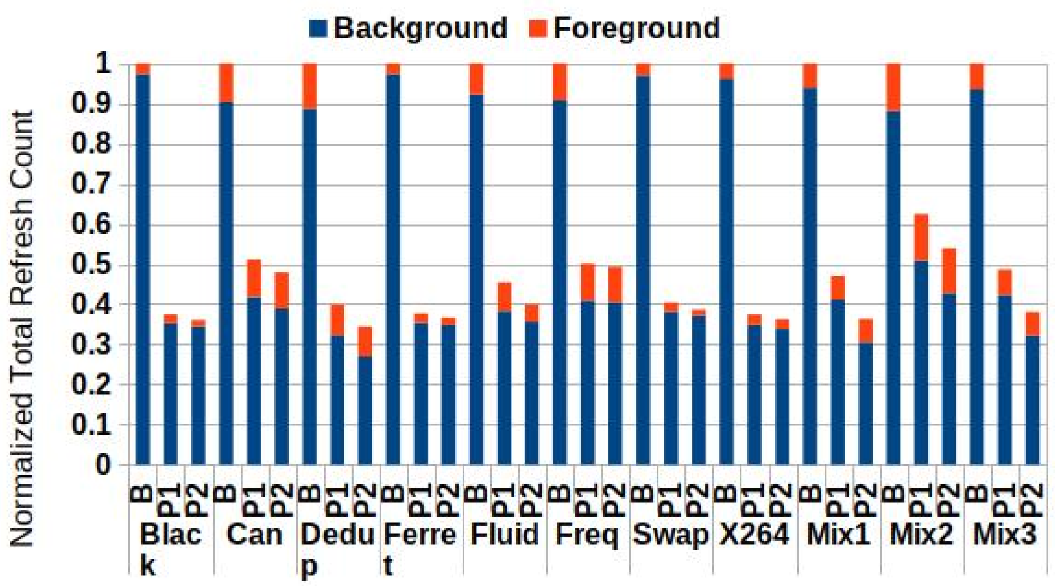


Fig. 4.5 Normalized Total no. of refreshes incurred in PBOR w.r.t PRR

computer vision, etc. To simulate the behavior of multi-programmed benchmarks, we constitute three mixes with four benchmarks taken from the PARSEC benchmark suite, as shown in Table 4.3. All the design overheads and effects of protocol changes are taken into consideration during simulations. We compare the proposed policy (PBOR) with baseline refresh scheme (PRR) that refreshes each and every cache line.

We show the results for the following techniques:

- **SRAM**: This is a baseline SRAM cache which needs no refreshing and has the best access latency. However, such aches suffer from high leakage energy.
- BASELINE : Periodic Round Robin Refreshing
- REFRINT: Refrint Poly Phase Refreshing
- PBOR : Private Block Optimized Refreshing
- PBOR_SZ: PBOR with Skipping restore of zero data blocks.

4.4.2 Results and Analysis

Number of refreshes: All eDRAM banks get periodically refreshed. As refresh is a high priority task, it gets preference over normal cache access. In particular, when a

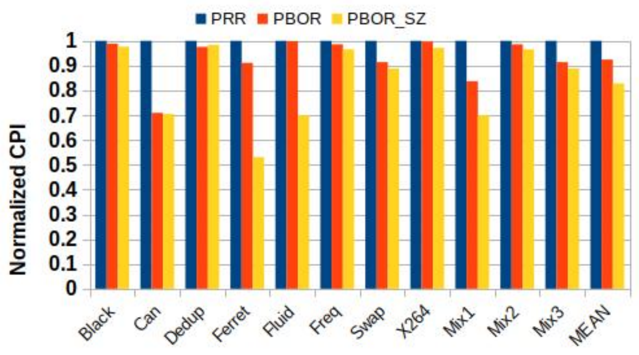


Fig. 4.6 Normalized CPI for PRR and PBOR

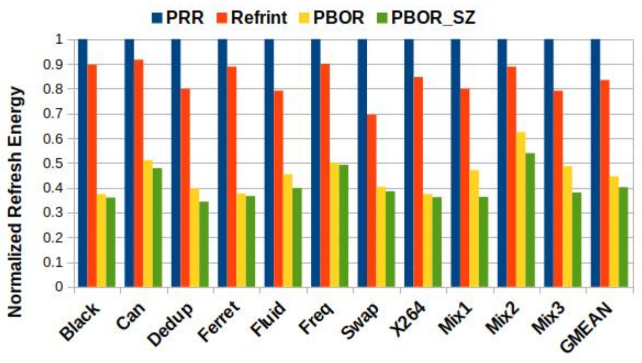


Fig. 4.7 Normalized Refresh Energy of proposed PBOR over PRR

Table 4.3 Experimental workloads

Category	Benchmarks
Multi-Threaded	Blackscholes (Black), Canneal (Can), Dedup (Ded), Facesim (Face) , Ferret (Ferr), Fluidanimate (Fluid), Freqmine (Freq), Swaptions (Swap), X264
Multi-Programmed	Mix1: Dedup, Fluid, Swap, X264 Mix2: Black, Dedup, Freq, X264

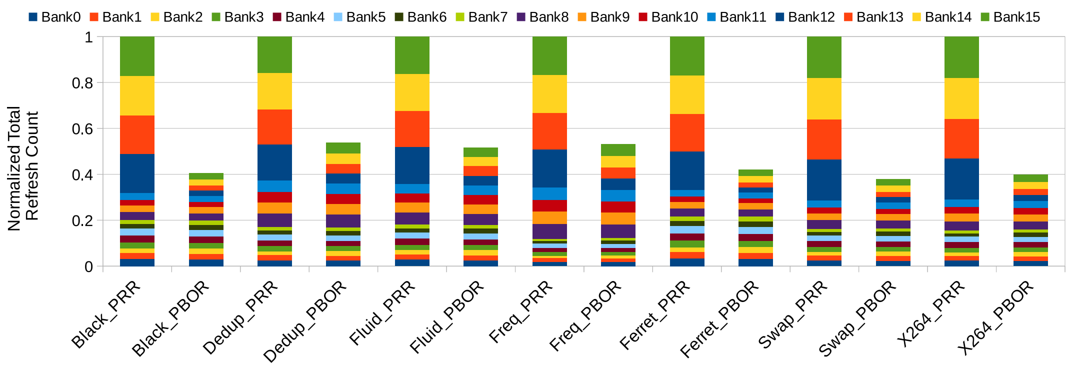


Fig. 4.8 Bankwise Normalized Refresh Counts of PBOR and PRR

request comes to access the LLC and the bank is currently undergoing a refresh, the request is stalled. We categorise the refreshes into two types: (i) foreground refresh occur when the current cache accesses are stalled due to bank getting refreshed; (ii) background refresh occur when the bank is not accessed while it gets refreshed. In case of foreground refreshes the accesses are stalled and it affects the performance of the system, whereas background refreshes do not affect performance. However, they contributes significant amount of total energy consumption.

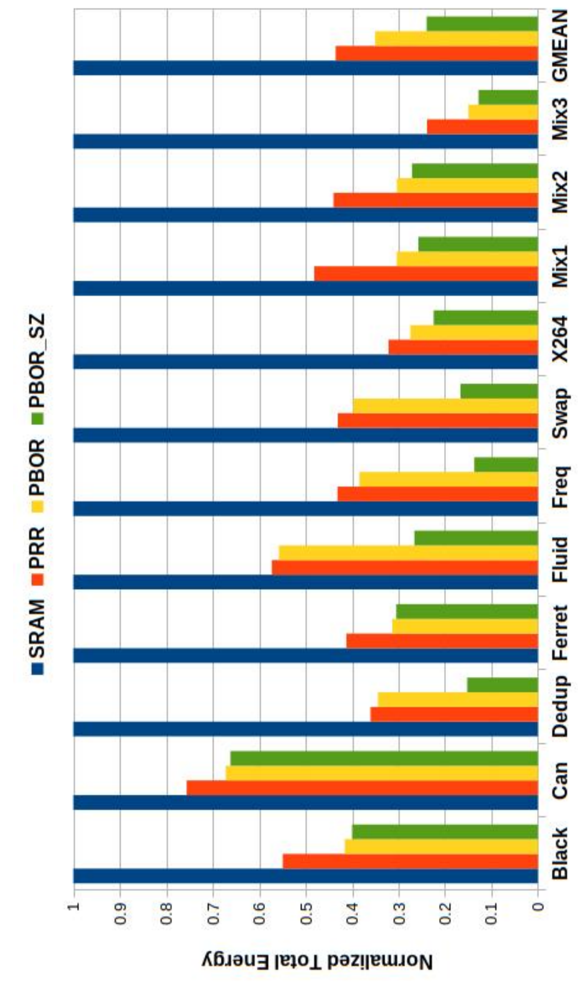


Fig. 4.9 Normalized Total Energy Consumption for PRR and proposed technique w.r.t SRAM

As we avoid refreshing private blocks, we save on several un-necessary refreshes. Figure 4.5 shows the reduction in refreshes achieved by our proposed methodology against the baseline which refreshes every cache line(As explained in the background section 4.2). Our previously implemented policy PBOR leverages the GETX requests operations, it is beneficial for write-intensive applications. In MESI, GETS request by a single requestor may directly allocated a block into exclusive mode. Over the PBOR we have implemented improvisation of SKIP-ZERO policy. Here, savings provided by this upgraded version of PBOR policy depends on the frequency with which accesses over zero-value blocks. With the PBOR_SZ, we get a further reduction of 2-3%in the range of 36-65% in total refreshes with an average reduction of 52%. The blocks which are held privately by the higher level caches are not refreshed. Such blocks will incur a refresh only after they are written back to the LLC.

The data blocks get distributed across the LLC banks depending upon the application behavior and working set size. Some banks get more accesses whereas some other banks get fewer accesses. Also, some banks have more blocks cached privately by higher level caches compared to other cache banks. Figure 4.8 shows the bank-wise reduction in the number of refreshes for all benchmarks with respect to baseline PRR policy. One can see that some banks benefit more from the private blocks compared to others. Such banks get released from refreshing overhead much earlier giving quicker opportunity to start normal cache accesses. We also compared our techniques with Refrint, which is one of the traditional refreshing policy in eDRAM Caches. We have implemented Refrint version over our baseline technique results are in next sub-section.

Comparison with Other techniques: Agrawal et al. [97] proposed a series of strategies for eDRAM cache refresh energy optimization. Refrint recommends two strategies: 1)timing-based policies (Refrint Poly Phase Policy), and 2) data-driven policies (Writeback based policy). In compared to timing-based strategies, data-based policies are more aggressive in evicting data blocks from the cache. Therefore as consequence, data-driven rules may cause a rise in cache misses in LLC, trigger negative impact over the system performance. As a result, we used Refrint Poly-Phase to compare our baseline distributed refreshing approach to our suggested strategy. The baseline (PRR), which is an optimized policy upon which we have applied refrint. In particular, PBOR gives refresh energy savings of 56% percent over the baseline where Refrint gives only 17% savings w.r.t to PRR baseline. The improvisation of skip zero gives a further reduction in the refresh energy consumptions and give a savings of 58% over the baseline. If we compare our policy with the basic SRAM we didn't have any refresh operations. In case of comparison with SRAM, we have compared the energy savings and the performance w.r.t SRAM in the next section.

Performance: While the cache is getting refreshed, the normal cache accesses have to be stalled. The reduction in the number of refreshes implies that the cache is not blocked for a long time. This implies performance improvement due to faster access latency of the LLC. Figure 4.6 shows the normalized Cycles Per Instruction (CPI) for the baseline and proposed approach. In particular, for PBOR policy get an improvement in CPI with an average of 8%. We get this reduction, as background refreshes are more than foreground refreshes. We get a performance degradation due to the eDRAM refresh latency and read/write operations. The improvised policy PBOR_SZ skips unnecessary read and write of zero blocks. Besides getting reduction in refresh operations over Zero Data Content, we get advantage of skipping read and write over those blocks. Technically, to read zero block content, we are only looking at the tag-array. If ZIB=1, then a zero block is served in response. Similarly, we are not performing write for those blocks for which ZIB=1. We only write block when block contains a non-zero data content. We are saving extra cycles consumed on writing and reading Zero blocks, i.e., 15 cycles on each operation. Thus, we get a performance improvement in PBOR_SZ. We get an improvement of 18% over the PRR baseline. If we compare with the SRAM we get a performance degradation due to the stalls which we get with respect to eDRAM. So, we are not comparing performance with respect to SRAM.

LLC Energy Consumption: LLC consumes energy for normal accesses as well as for refreshing the cache lines. Even in the cases when the refreshes do not stall the normal cache access, they still incur energy. Reduction in the number of refreshes thus also leads to a reduction in energy consumption. Figure 4.7 shows the normalized reduction in refresh energy. We get a reduction in the range of 46-63% with an average of 54%.

Replacing SRAM with eDRAM based LLC gives savings in leakage energy. The read and write energy required by eDRAM is also less compared to SRAM. Hence we get a considerable reduction in the access energy for eDRAM over SRAM. The effect of refreshes on the total energy is shown in Figure 4.9. Here the values include the static, dynamic as well as the refresh energy. We get an average reduction in total energy by 57% and 65% respectively in PRR and PBOR over baseline SRAM. The improvisation of skipping zero policy over the PBOR gives savings of 76% over the SRAM. It gives higher savings due to optimization of not writing zero-blocks to the LLC, which is not adopted by the PBOR policy. The PBOR_SZ policy leverages advantage by existence of Zero Data blocks in the application. As shown in Fig. 6.12, we get an average of 42.5%

zero data traffic at the last level. As shown in Table 5.1, this traffic is mainly divided across blocks loaded from main memory (column-3) and blocks written back from ULC (column-4). This percentage distribution of blocks leads to savings in write energy for our optimization; also, it directly impact the refresh energy savings by skipping unnecessary refresh of Zero blocks.

4.5 Overhead Analysis

4.5.1 Hardware Overhead

The additional hardware blocks required for the proposed architecture is the Refresh optimizer. Refresh Optimiser consists of a to-refresh bit for each particular cache block. In particular, for a 4MB 16way set associative cache, 65,536 bits are required which is only 0.19% of total eDRAM cache size. Fault tolerance and reliability aspects over the to-refresh bits are not considered in the design. Our technique incurs the following overhead: (i) one To-Refresh bit, one Instruction bit, and one zero indicator bit(ZIB) (ii) a (42bit + 64B)-size swap buffer used to migrate the tag and data block between the regions. In total, there is 16.06 KB additional storage, which is only 0.39% overhead for a 4MB cache. Note that the area overhead for the additional tags of PBOR_SZ is modeled using CACTI.

4.6 Summary

This work proposed the use of eDRAM as a replacement for SRAM in the last level cache. Low leakage of eDRAM comes at the cost of less retention time, and thus requires periodic refreshes. The work proposed to identify blocks that are privately cached by higher level caches and avoid refreshing them in the LLC. This saves considerable number of refreshes as well as refresh energy. The MESI cache coherence protocol is modified to identify the private blocks. An additional logic block is proposed that coordinates with the refresh circuitry and controls which addresses get refreshed.

From experiments, it was observed that the number of refreshes was reduced by 55% over basic refresh scheme. As the cache banks are stalled for lesser duration compared to the basic scheme we get improvement in performance by 4% on average. Compared to SRAM based cache we save overall total energy by 62%.

Thus, if one can manage refreshes carefully, eDRAM based LLC can become a valid replacement of SRAM.

CHAPTER 5

CORIDOR: Coherence based technique to Reduce Restores caused by Read Disturbance Error in STT-RAM Caches

In this chapter, we are going to discuss about the crucial reliability issue created by Read-Disturbance Error(RDE) in volatile-STTRAM. RDE presents this threat after scaling of STT-RAM. Towards this, our policies dynamically reduce the number of times a lines need to be restore after reads. In that the restore is similar to a refresh and falls in-line with our thesis objective of reducing refresh operations. The main theme is to identify read intensive blocks using coherence messages and temporal locality and move them to an SRAM buffer. This guarantees that several of the future reads will get serviced form SRAM and will not lead to RDE related restores in the STTRAM main cache.

5.1 Introduction

STT-RAM has high density, high write endurance and low read latency [98]. These features make it a promising candidate for architecting last-level caches. With ongoing feature-size scaling, the write current has reduced exponentially. The read current has been decreasing at a much smaller rate [37]. This is because at current values below $20\mu A$, the data cannot be correctly sensed [99], leading to the decision failure issues. In fact, at feature sizes at or below 32nm, the read current value becomes sufficiently close to the write current that a read current may disturb the stored data. This is referred to as "read disturbance error" (RDE). In STT-RAM, the read/write paths are shared. The read current's induction effect creates disturbance among the MTJ cell, which leads to the flipping of bit orientations [100]. In fact, at small feature sizes, RDE can surpass the capabilities of even strong error-correcting codes such as 5EC6ED, which corrects five errors and detects six errors [101]. Since STT-RAM write-optimization techniques such as early-write termination and write-read-verify introduce additional read operations, they increase the occurrence of RDEs greatly [102]. Due to these factors, more than writability [103, 104], it is readability that is likely to be the crucial bottleneck in STT-RAM performance [105, 106, 101, 107].

A solution to the RDE issue is to perform a write operation immediately after a read operation. This operation is referred to as a restore operation, and the overall scheme is called "high-current restore required" (HCRR) [99]. However, this scheme incurs significant overhead, especially for the read-intensive workloads. A read/write operation stalls the bank, and this translates to substantial latency and energy penalties. Given this, intelligent and effective techniques are vital for managing the RDE issue for improving the STT-RAM performance and energy efficiency.

In order to mitigate the RDE issue, we need to refresh/restore the data immediately after a read. We cannot wait until some time in the future (called the retention time in DRAM) to initiate a restore. This immediate need to refresh creates further challenges in designing a solution to mitigate them because every solution will affect the hit time of the cache. Another challenge is that the number of times that we need to refresh is proportional to the number of reads performed in the cache and not the data's retention time. This adds significantly to the overhead, given that the percentage and frequency of reads to a cache are very high. The solutions of RDE must ensure that the cache is stalled for as little time as possible, and at the same time, the data is not lost. Towards designing methods to overcome RDE and its impact on performance and energy, we make the following observations and use them in our proposals.

Firstly: The cache coherence protocols in shared memory multicores guarantee the correctness of shared data. If we can exploit the protocol level transitions to identify avenues to avoid the restore operations for RDE, it can save the number of restore operations.

Secondly: Caches are designed on the property of spatial and temporal locality. The temporal locality is displayed in the frequency of accesses to the same block as well as in the type of access. Specifically, the block that is 'read' now is more likely to incur a 'read' operation shortly instead of a write operation. This temporal locality in reads can be helpful to identify blocks that are read-intensive. After identifying such read-intensive blocks, we can extend the concept of cache and relocate the read-intensive blocks to a small lookup buffer type of storage. This can help in serving the future reads from the buffer instead of reading from the cache. The buffer has to be faster in lookup and should not itself suffer from the RDE issue. Therefore, the buffer must be designed using SRAM technology and must be small in capacity. The temporal locality for a particular block is applicable over a sequence of accesses, and after a burst of accesses is completed, the block may be dormant for some duration in the execution before being accessed or evicted. This burst of accesses is evident from the average number of consecutive reads incurred by a block (cf. Fig 5.3). Once the burst of read-accesses is over, the location of this block can be used for storing newer read-intensive blocks. In particular, once a block gets a sufficient number of read accesses in close succession, the access frequency reduces over the execution time, and such blocks can be removed from the buffer. This observation can be used to design a replacement policy for the buffer. The buffer can be searched in parallel to the main cache, and hence, it does not cause any performance bottleneck.

Lastly: Looking at the content of data is also helpful. Specifically, if the data value is zero, then the refresh/restore is unnecessary. If we can identify such blocks and keep a note of their content, then the number of restores can be saved.

Contributions: In this paper, we propose three architectural techniques for mitigating RDE in STT-RAM last-level caches. Finally, we evaluate a fourth technique that combines all these techniques.

1. SKIP_WH works on the observation that on a write-hit, the data block is requested from the upper-level cache with exclusive permission. Hence, at some point in time, the block owner will update the block in LLC, and the existing copy of data will be overwritten. Hence, on a write-hit, we need not perform a restore operation (Section 5.3.1).

2. RD_BUF works on the observation that due to temporal locality, some blocks see a significant number of read operations. We use a small SRAM buffer and migrate such read-intensive blocks from STT-RAM LLC to this SRAM buffer. In the future, reads to these blocks are served from the SRAM buffer, which avoids the restore operation to these blocks (Section 5.3.2).

3. SKIP_ZERO uses a “zero-indicator bit” which is set or reset on a write operation, depending on whether the value being written is zero or not (respectively). Based on this, our technique avoids the restore operations whenever a block has zero

data value. This saves both read energy and the need to restore the block (Section 5.3.3).

4. CORIDOR policy that synergistically combines all these policies (Section 5.3.4). Our cycle-accurate simulations show that compared to HCRR policy, CORIDOR policy saves 31.6% energy and brings the relative CPI (cycle-per-instruction) to 0.64× (Section 5.5.1-5.5.2). By contrast, an ideal RDE-free STT-RAM saves 42.76% energy and brings the relative CPI to 0.62×. Thus, CORIDOR entirely removes the performance impact of RDE and also reaches a significant fraction of energy saved by the ideal RDE-free cache.

The chapter is organized as follows: section 5.2 presents the background related to Read Disturbance Error in volatile-STTRAM cell. The section 5.3 presents the proposed RDE mitigation techniques. The experimental methodology is illustrated in section 5.4. Results and analysis are discussed in section 5.5. Section 5.6 reports the overhead analysis. Finally, we summarize this chapter in section5.7.

5.2 Background on RDE phenomenon

When a large current is applied to an STT-RAM cell for a short duration, the value can be written and read reliably. However, if the current is reduced, longer pulse duration is required to complete the operation. Both these factors affect the reliability of the cell, in that the reads become destructive [99]. The low read current leads to magnetic disturbance in the MTJ, and this can lead to flipping the stored data [100], thus causing the read disturbance error (RDE). Process variation further exacerbates the issue of RDE [108].

Consider a bitcell having an MTJ of area A_j. The electric current I_M equals "critical write current" (I_C) for P-to-AP switching [109]. Let MTJ oxide thickness be denoted as t_{MgO}. Let $t_{RDE-min}$ be the minimum thickness of MgO required to achieve error-free read operation. The MTJ is in P state with $t_{MgO} = t_{RDE-min}$ and resistance R_P. When $t_{MgO} < t_{RDE-min}$, current I_M rises above I_C and due to this, in a single read cycle, the data is written to this bitcell. This causes an RDE. Similar phenomenon happens in AP state. The value of $t_{RDE-min}$ depends on A_j. For a bitcell, the probability of RDE P_i is given by,

$$P_i = \lim_{\delta \to 0} \sum_j P(Q - \delta \leq Q \leq Q + \delta) \cdot P(t_{MgO} \leq t_{RDE-min})$$

where Q = A_j. This probability can be computed using Monte Carlo simulations.

Finding the exact probability of an RDE would require detailed modeling of circuit-level parameters and effects. Since this is beyond the scope of our architecture-level study, we assume a 100% probability of RDE occurrence in this paper. In other words, we assume that an RDE happens on each read operation.

5.3 Proposed RDE Mitigation Techniques

In this section, we will illustrate three of our proposed techniques: (i)SKIP_WH, (ii)RD_BUF ,and (iii)SKIP_ZERO

5.3.1 Skip Restore on Write-Hit (SKIP_WH)

Exploiting the properties of coherence protocol, we avoid restoring certain blocks that will get updated in the future by the upper-level cache (ULC). In particular, if the ULC fetches a block for writing, then such a block need not be restored in the LLC. This can be identified with the help of the coherence protocol.

The LLC receives read and write requests ULC under the given coherence protocol. The requests that result in a cache-hit are mainly of three kinds: (i) *read-hit*: request to load a block for reading, (ii) *write-hit*: request to load a block for writing, and (iii) *write-back*: data is written back to the LLC after getting evicted from the ULC. The requests of type (i) and (ii) incur a block read operation in the LLC. In an STT-RAM-based LLC, all these reads need a restore operation. For the read requests, we need to restore the block in the LLC after serving the request.

Key Idea: We make an important observation that on a write-hit, the data block is requested from the ULC with exclusive permission. Hence, for the write-requests (type (ii)), the block will be modified soon by the ULC and eventually written back to the LLC. Therefore, for such requests, a restore operation is unnecessary. Our first proposal, termed SKIP_WH, skips restoring the blocks on a write-hit (type (ii)) request. Note that the MESI protocol does not invalidate the block on a write-hit. Hence, the read on a write-hit incurs a restore.

Figure 5.1 shows the fraction of write hits in the overall hits (summation of both read and write hits). Evidently, different benchmarks have a significant fraction of write-hits. SKIP_WH avoids the restore operations on these write-hit operations.

Notice that SKIP_WH does not create any correctness issue. Suppose there is a request to the LLC from the coherence controller to read the block to another process/processor. Note that the block under consideration is with the ULC, and our policy

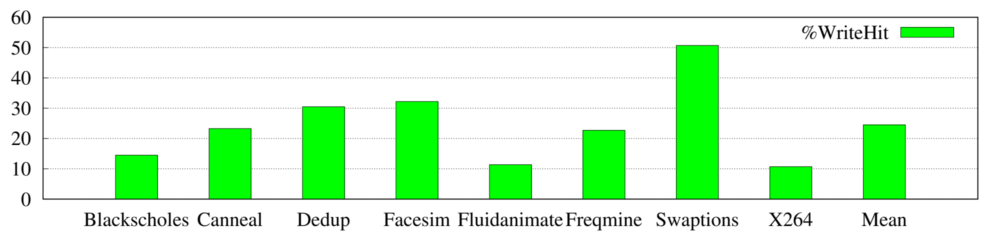


Fig. 5.1 The percentage of write hits in different benchmarks

does not restore it in the LLC. As part of the coherence protocol, the LLC always fetches the up-to-date copy from the ULC before forwarding to the requestor. This is because the copy in LLC is assumed to be stale. Thus, the coherence protocol guarantees to return the correct data to the requestor. Our technique seeks to avoid restore of such prospective stale blocks.

Figure 5.2 shows the flowchart of the proposal. On a cache miss, the block is loaded from memory, and the request is served. Initial loading of the block incurs a write to the STT-RAM, and such writes do not count under restore operations. On a read-hit in the LLC, the block is sent to the ULC, and a restore is performed by writing the block to the LLC. For a write-hit, the block is sent to ULC, and our policy does not perform the restore in LLC.

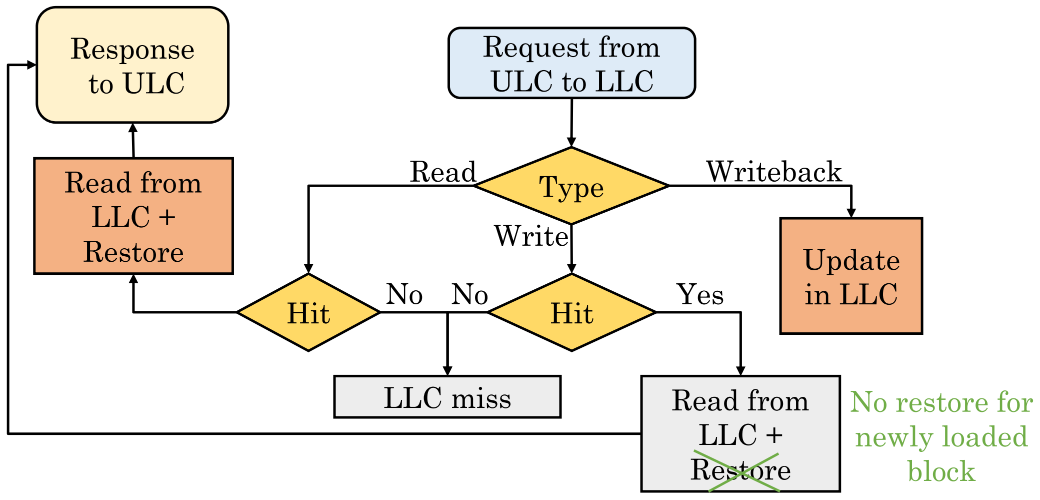


Fig. 5.2 Flow-chart of SKIP_WH

5.3.2 Migrating Read Intensive Blocks to an SRAM based Read-Buffer (RD_BUF)

Key idea: It is well-known that due to the temporal locality, the cache blocks are accessed repeatedly before eviction. During their lifetime in the cache, these blocks also show temporal locality in their read accesses, in that the blocks receive several read accesses between the write accesses or block eviction. Figure 5.3 shows the average value of consecutive reads (CRead) [110] seen by the blocks. On average, in most benchmarks, each block receives 4 to 8 consecutive read accesses. Intuitively, once a block receives a few reads, it is likely to get more reads soon. In other words, the higher the value of CReads for any application, the higher is the need for its restore operations.

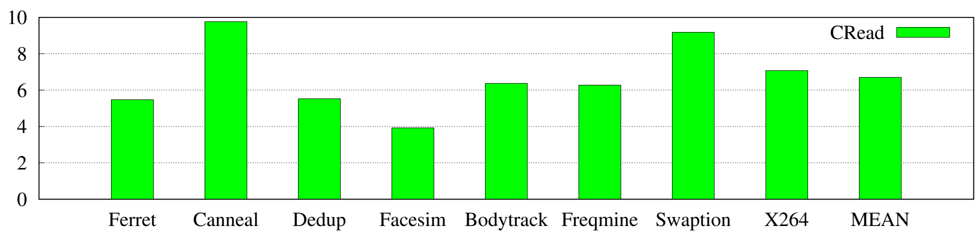


Fig. 5.3 The average consecutive reads (CReads) on the blocks in a bank for different benchmarks.

Our second proposal exploits this idea to detect the read-intensive blocks and migrate them to a small SRAM-based read-buffer. Future reads to these blocks are served directly from the read-buffer. This avoids the need to restore them in the STT-RAM LLC. As the read buffer is small associative storage, certain blocks get evicted from the buffer and get placed in their original location in the LLC. Note that the LLC maintains the tag for such entries and tracks the migrated blocks' location. All the reads that hit in the read-buffer contribute to saving the restore operations in the LLC.

Architecture

Figure 5.4(a)-(b) show the architectural changes required for implementing RD_BUF. In the LLC, we keep two fields in addition to the regular meta-data: a location bit (Loc) and a read counter (RC). The former is used to identify the block's position, whereas the latter is used to count the number of consecutive reads incurred by the block. The LLC is associated with a small, fully associative storage called the read-buffer, made using SRAM technology. This read buffer is limited in size to enable faster service times. Small SRAM also helps in controlling the leakage power consumption due to the read buffer. Every entry in this buffer keeps the tag and data block along with a read counter (RCB). This counter is used for deciding the victim block from the read buffer.

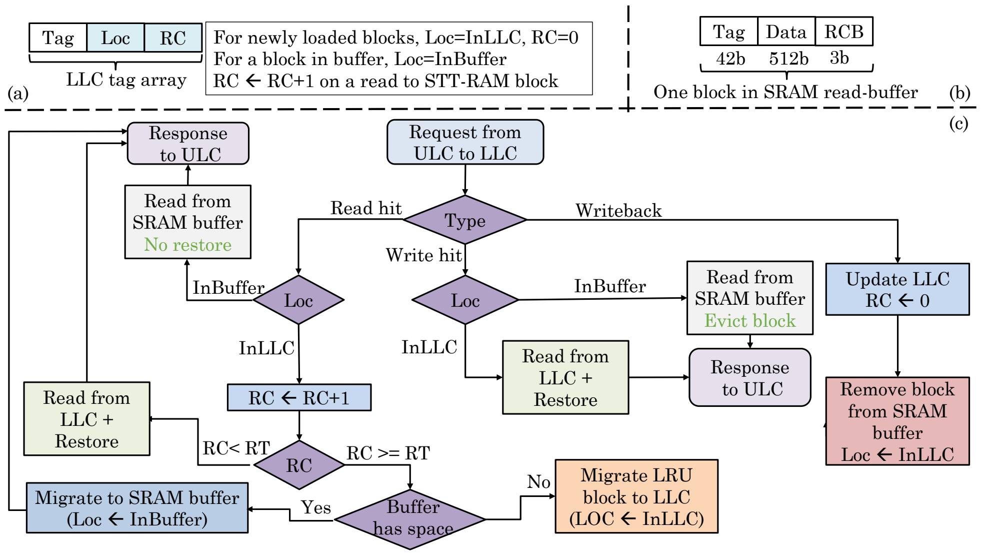


Fig. 5.4 (a) Modified tag array (b) One entry of SRAM buffer (c) Flow-chart of RD_BUF

Working

The proposal RD_BUF moves the read-intensive blocks to the read-buffer, and future reads to such blocks are served directly from the read buffer. Identification of read-intensive blocks and deciding the eviction policy for the read-buffer are discussed first.

Migration to read-buffer: From Fig. 5.3, it is evident that, on average, a block receives four to eight consecutive reads. To keep track of the number of successive reads, we maintain a read counter (RC) with each LLC block. This counter is incremented on each read access and is reset on block allocation, writes, and writebacks. We use the concept of a read threshold (RT) to infer the read intensity of the blocks. If the $RC > RT$ then the block is declared to be read-intensive and becomes eligible to be moved to the read buffer.

The value of RT is set such that a block is moved to read-buffer before it exhausts all the reads in the LLC. Assume that the RT value is 4. If the CRead value for a block is 6, it sees four reads while in the LLC and subsequent two reads while in the read-buffer. Thus, migrating it to read-buffer saves two restore operations. The values for RT is found empirically and set to 4 in our experiments. Detailed analysis using different values for RT is also presented in Section 5.5.3. Once a block migrates to the read buffer, the Loc bit is updated to indicate its current location. In particular, the location value of `InLLC` and `InBuffer` show that the block is stored in LLC and buffer, respectively.

Eviction from read-buffer: The read-buffer being a small associative memory, has limited capacity. Several blocks that are identified as read-intensive are migrated to the read-buffer. For making room for the newer blocks, certain older blocks need to be evicted from the read-buffer. We use a modified LRU-based policy to decide the victim blocks to evict. The evicted block is moved back to the LLC at its original position, and its location bit is updated to indicate this. Once this block moves back to LLC, the associated read counter, RCB, is reset.

A read counter (RCB) is kept with each block in the read-buffer. The RCB is set to zero when the block is migrated in. Also, the RCB is incremented with each read-hit to the particular block. The block having the least value of RCB is a good candidate for eviction. However, the RCB value of recently migrated blocks is usually very small. To account for the duration of stay for blocks in the read-buffer and their read-intensity during their stay in the read-buffer, we use a combination of the timestamp and RCB for deciding the eviction policy. In particular, we determine the victim block from among the top 25% blocks from the LRU stack. From these top 25% blocks, the block having the least RCB value is evicted from the read-buffer.

For every request coming to the LLC from the upper-level cache (ULC), we have the following cases to handle (refer Figure 5.4(c)):

- Read hit: Check the location bit of the block. If `Loc=InBuffer`, then serve the request directly from the buffer and increment RCB. If `Loc=InLLC`, increment the associated read counter (*RC*) and serve the read request. If the read counter is less than the threshold ($RC < RT$), perform a restore operation for the block. Otherwise, for $RC \geq RT$, migrate the block to the read-buffer and reset RC to 0 and `Loc` to `InBuffer`. Note that we do not restore the block in LLC.

- Write-hit: A write-hit implies that we have to serve the block to the ULC where the processor will modify it. From the LLC perspective, the write-hit also amounts to a block read. Hence, if the block is `InLLC`, it is restored, and RC is set to 0. If the block is `InBuffer`, the block is serviced to the ULC and evicted from the read-buffer to make room for new blocks. This latter action is done because the ULC will update the block and the copy in the read-buffer is stale.

- Writeback: If the block is being written back from the ULC, write its data in the LLC irrespective of the block's location. If the block was in read-buffer, free that entry from the read-buffer. Reset the read counter and also update the location bit `Loc=InLLC`.

- Cache miss: Load the block from the main memory by evicting an LRU victim from the LLC. If the victim is located in the read-buffer, it is evicted, and its contents are written back (if the block is dirty). The newly loaded block is written in the LLC, and the data is also forwarded to the ULC. For the new block, set `Loc=InLLC, RC=0`.

Working Example of RD_BUF Policy.

Figure 5.5 shows the working example of RD_BUF policy for a 16-way STT-RAM LLC.

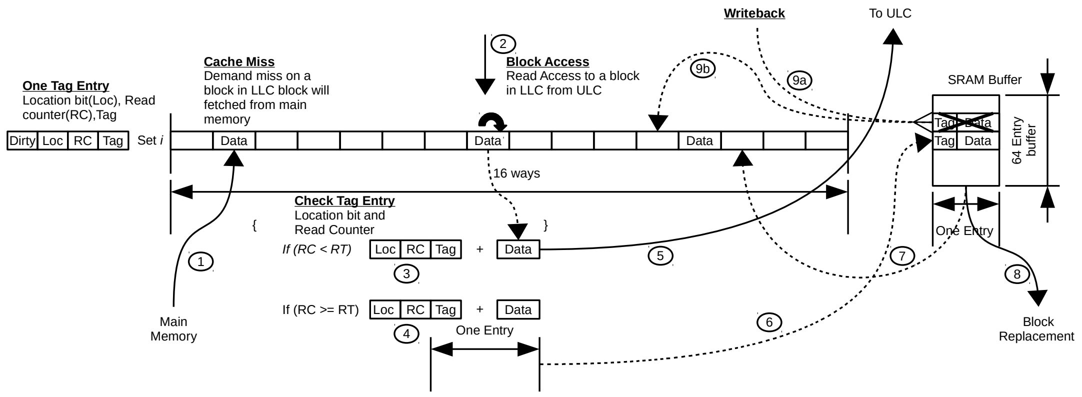


Fig. 5.5 Working Example of RD_BUF policy.

- Cach Miss: For a cache miss, the block is fetched from main memory (1). This block is served directly to ULC and not considered for restore operations.

- Read hit: On a read hit, the *RC* value in the tag entry of the block is checked (3). If $RC < RT$, the request is served to the ULC (5), as done in the baseline LLC and a restore operation is performed (2). If $RC \geq RT$, the request is served (4), and a copy of tag with data is migrated to read buffer (6). Future requests to this data are served from the SRAM buffer, which avoids the need for restore operations. However, if there is no space in the buffer, then a victim is chosen using the approximate LRU policy and is migrated back to the STT-RAM LLC (7). Further, when a replacement operation is performed in the STT-RAM LLC and the block is present in the SRAM buffer, the block is evicted (8).

- Writeback: If a writeback from ULC is targeted to a block that is currently present in the buffer (9a), then this entry is deleted from the buffer, and the writeback is redirected to the LLC (9b). However, if the writeback is targeted to a block present in the LLC, the block is updated normally, and the counters are reset.

Comparison with hybrid caches

All the existing hybrid SRAM-STTRAM cache designs are designed to mitigate the write endurance issue or reducing the write-energy overhead of STTRAM. These techniques seek to reduce the number of write operations to STT-RAM by migrating write-intensive cache blocks to SRAM [111, 112]. In the experimental evaluation, we present the comparison with two hybrid cache designs [111, 112]. We now summarize them briefly.

Wu et al. [111] propose RWHCA (read-write aware hybrid cache architecture), which has a small write region made of SRAM and a large read region designed with STTRAM. Each cache block stores a saturation counter. They discuss a data migration policy to keep the read/write-intensive data items in corresponding regions. On a load miss and a store miss, the block is loaded in read-region and write-region, respectively, and the saturation counter is initialized to 11. On a read from the read-region or a write to the write-region, the saturation counter is incremented. Otherwise, the counter is decreased, and if the MSB of the counter reaches zero, it indicates successive hits in the wrong region. Hence, the LRU block in the opposite region is swapped with the block seeing a hit and the counters of both the blocks are reinitialized.

Ahn et al. [112] observe that the data referenced by the same load/store instructions have similar characteristics. Hence, the number of writes to a cache block correlates with the instruction that loads the block into the cache. Their scheme tracks the instructions that usually load write-intensive blocks. Based on this, their technique, termed WI (write intensity-based technique), predicts the write intensity of the upcoming blocks. Then, in an SRAM-STTRAM hybrid cache, it loads write-intensive blocks in the SRAM ways and other blocks in the STTRAM ways. This leads to write-energy saving.

Recent works have shown that with ongoing feature-size scaling, it is readability and not writability that will bottleneck the scaling of STT-RAM. This is because, with continuous feature-size scaling, the read-disturbance issue is becoming severe. Since read-operations lie on the critical access path, RDE error has a significant impact on the performance.

In the context of mitigation of RDE, hybrid SRAM-STTRAM caches are ineffective, as confirmed by our results. This is because the traditional hybrid SRAM-STTRAM caches move write-intensive data to SRAM and read-intensive data to STTRAM. Moving read-intensive blocks to STTRAM would aggravate the RDE issue since read-intensive blocks would incur many reads and hence, a high number of RDEs. Further, hybrid SRAM-STTRAM caches suffer from challenges of fabrication due to different characteristics of SRAM and STTRAM.

Our design uses a separate SRAM buffer, and hence, it is easier to design. Further, assuming a 16-way SRAM-STTRAM hybrid cache, where one way is designed using SRAM, the total capacity of SRAM is 256KB. By contrast, our SRAM-buffer has a total capacity of 16KB (64 entries per 1MB bank, each entry of size 64B). Thus, our design can work with the smaller relative capacity of SRAM since SRAM memory has a high leakage power consumption.

5.3.3 Skip restore on Zero Data Blocks (SKIP_ZERO)

It is well-known that a significant fraction of data blocks in the applications has a zero value [113, 85]. For example, Ekman and Stenstorm [83] have found that for their workloads, 30% of blocks have all 64-bytes as zero. Fig. 5.6 shows the percentage of zero-data blocks in our benchmarks. Evidently, the frequency of zero-data blocks is different in the benchmarks, and on average, 44.4% blocks are zero data blocks (ZDB). Based on this information, our next proposal is to skip the restore operations on the ZDBs. Whenever a block having zero content is read, we do not restore its contents. This saves both read energy and the need to restore the block. In this way, the zero-detection approach saves both access time and dynamic energy. If the data content is not zero, then a read operation happens normally, and the block is restored. We further propose another optimization, where a writeback to a block with zero data content will not be written to the LLC. This optimization would further reduce the number of writes performed in the LLC.

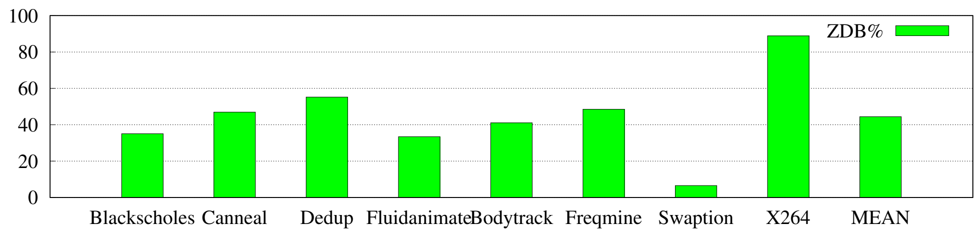


Fig. 5.6 The percentage of zero data block (ZDB) in different benchmarks

Architecture: Figure 5.7(a) shows the changes to the tag array. Here, the LLC tag array is associated with one zero indicator bit (ZIB) to indicate whether the data block has zero content or not. If this bit is set, then the block has zero value content. We use a comparator unit called the "zero detection logic" (ZDL) to detect the data value. This is shown in Figure 5.7(b). Whenever a block is loaded in the LLC from the main memory or written back from the ULC, it is passed through the ZDL. If the block has a zero value, then the block is not written in the LLC, but the ZIB is set. When a block is read from the LLC, which has the ZIB set, we do not perform an actual read operation on the data block from the LLC. Instead, we use a zero block reconstruction logic (in particular, a register storing zero value) to send a block with zero content. Note that the ZIB bit information is maintained with the tag array. As the tag-array is made of SRAM, the ZIB bit is free from RDE.

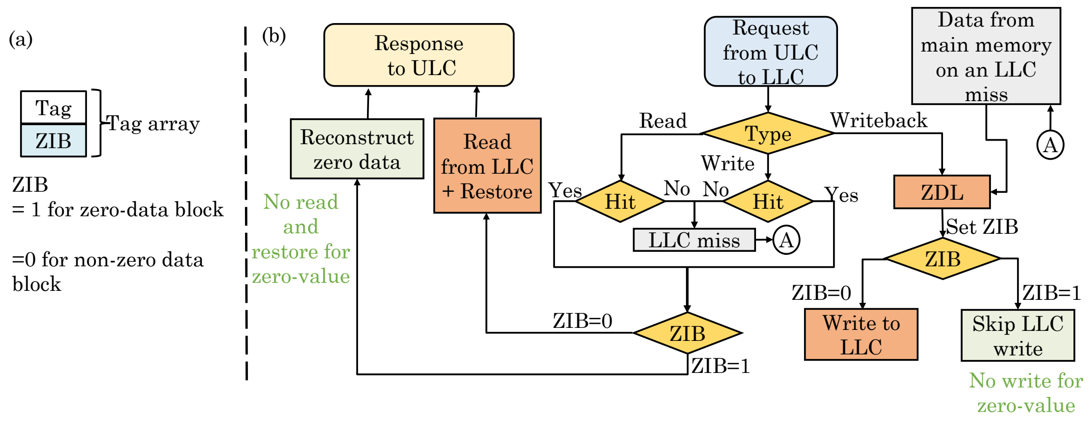


Fig. 5.7 (a) Modification to the tag array (b) Flow-chart for the SKIP_ZERO policy.

Working: Figure 5.7(b) shows the flow-chart for the SKIP_ZERO policy. For every request coming to the LLC from the ULC, we have the following cases to handle:

1. Read hit: Check the ZIB bit of the block. If `ZIB=1`, this is read access to a zero data block, and hence, the read is served by reconstruction of the data block with zero values. This avoids unnecessary restore operation on the data block. If `ZIB=0`, the read operation happens normally, and after this, a restore operation is performed.

2. Write hit: From the LLC perspective, the write-hit also amounts to a block read. Therefore, the same actions as those taken on a read-hit take place in this case.

3. Writeback and cache miss: A write needs to be performed to the LLC on a writeback from ULC to LLC and on a cache miss when the data comes from the main memory to the LLC. In both these cases, the ZDL checks the data contents and sets the ZIB. If the contents are zero (`ZIB=1`), write to LLC is skipped. However, if the contents are non-zero, a write happens normally.

Limitation: The policy is mainly dependent on the existence of Zero Data blocks in the application. As shown in fig. 5.6, we get an average of 42.5% zero data traffic at the last level. As shown in Table 5.1, this traffic is mainly divided across blocks loaded from main memory (column-3) and blocks written back from ULC (column-4). This percentage distribution of blocks leads to savings in write energy for our optimization; however, it does not directly impact the restore savings. In fact, among all the zero data content blocks, SKIP_ZERO saves restore operations only on the read hits to such blocks. The percentage read-hits to ZDBs are shown in column-5 of Table 5.1, and this percentage value is much lower than the percentage of total ZDBs (column-2). Note that SKIP_ZERO also reduces the number of writes on ZDBs; however, the restores are avoided only read hits to the ZDBs. Therefore, augmenting SKIP_ZERO with our other proposals (SKIP_WH and RD_BUF) is highly beneficial. Our next combined policy attempts to do this.

Table 5.1 Division of written zero data blocks in last level in SKIP_ZERO policy.

Benchmarks	ZDB Written in LLC(%)	ZDB loaded on Miss in LLC(%)	ZDB written on write-back in LLC(%)	ZDB Hit in LLC(%)
Blackscholes	35.05	15.23	19.85	10.8
Canneal	46.93	32.10	14.83	0.003
Dedup	55.22	41.40	13.82	22.172
Fluidanimate	33.4	10.02	23.38	50.47
Bodytrack	41.05	25.50	15.55	0.348
Freqmine	48.51	41.00	7.51	2.627
Swaption	6.54	6.17	0.37	18.225
X264	88.86	75.03	13.83	46.633

5.3.4 CORIDOR: Combining policies SKIP_WH, RD_BUF and SKIP_ZERO

In our fourth policy, called CORIDOR, we combine our earlier three proposals. Specifically, we skip the restores for the write-hits, migrate read-intensive blocks to read-buffer, and perform zero-value-based optimizations, as discussed before. For the cases where more than one (out of SKIP_WH, RD_BUF and SKIP_ZERO) policy may act, we first apply SKIP_ZERO, then SKIP_WH, and then RD_BUF. In other words, on a read or write operation, if the block is zero, read/write/restore operations are avoided.

It is noteworthy that the SKIP_ZERO policy avoids the migration of zero-data blocks to the buffer, which reduces the traffic between LLC and buffer, and contention for

Table 5.2 System Configuration and workloads ($\star$- CPU-Intensive, $\diamond$- Memory-Intensive)

Components	**Parameters**
Processor	2Ghz, Quad-core, X86
L1 Cache	Private SRAM based 32KB, 8 way, split I/D caches, 64B block, LRU
L2 Cache	Shared, STT-RAM based 4MB, 4 banks, 16-way, 64B block, LRU Per-bank: Hit/miss/write energy: 0.28/0.13/1.092 nJ, latency: 3.37/1.38/10.70 ns, leakage: 60mW
SRAM read-buffer	16-entry, fully-associative, approx-LRU, read/write energy: 0.0.007nJ/0.0007nJ, leakage: 2.93mW
	32-entry, fully-associative, approx-LRU, read/write energy: 0.009nJ/0.009nJ, leakage: 5.48mW
	64-entry, fully-associative, approx-LRU, read/write energy: 0.014nJ/0.014nJ, leakage: 10.44mW
	128-entry, fully-associative, approx-LRU, read/write energy: 0.021nJ/0.021nJ, leakage: 19.30mW
Workloads	Blackscholes (Black)*, Canneal (Cann)$^\diamond$, Dedup$^\diamond$, Fluidanimate (Fluid)*, Bodytrack (Body)*, Freqmine (Freq)$^\diamond$, Swaptions (Swap)*, X264*

the buffer space. Thus, the SKIP_ZERO policy helps in mitigating the overheads of the RD_BUF policy. On a write-hit, SKIP_WH policy avoids only restore operations, whereas SKIP_ZERO policy, if applicable, can avoid both read and restore operations.

5.4 Experimental Methodology

We use Gem5 [114] cycle-accurate full-system simulator to perform our experiments. Gem5 uses the Ruby memory module and uses the MESI cache coherence protocol. The architectural parameters are given in Table 5.2. We use 4 threads in a quad-core setup. The L1 cache is a private cache, and the STT-RAM-based L2 cache is the shared last level cache. We use a fully-associative SRAM read-buffer with 64 entries. The parameters of the STT-RAM cache are obtained using NVSim, and those of the SRAM buffer are obtained using CACTI. As for workloads, we use PARSEC [115] benchmarks with medium input sizes. Table 5.2 shows the classification of the workloads.

5.5 Results and Analysis

The results are presented on different metrics: Energy savings, Perfromance (CPI) and Speed-up. We have conducted various experiments over the architectural parameters. We present a detailed sensitivity analysis of RD_BUF policy over different Read-Threshold,and Read buffer size.

We show the results for the following techniques:

- High current restore required (**HCRR**): The HCRR scheme uses normal current ($20\mu A$) to read the STT-RAM cell and performs a restore (i.e., write) operation after every read operation to correct the RDE [105, 101]. It has been used in real prototypes [99]. With HCRR [99], each read operation also leads to a write operation, and thus, HCRR causes a significant increase in write traffic compared to the RDE-free STT-RAM cache. Since STT-RAM writes have high energy/latency overhead, HCRR can degrade efficiency and create bandwidth issues. As the number of cores accessing the LLC increases, the restore operations would make the LLC ports unavailable for normal accesses. Thus, HCRR has crucial disadvantages. We take HCRR as the baseline scheme due to its deployment in the product systems.
- RDE-free STT-RAM (**Ideal**): This scheme assumes an ideal, RDE-free STT-RAM.
- **BDI** [116]: Cache block compression technique that uses a base word and stores the block as the difference with respect to the base word.
- **RWHCA** [111]:
- **WI** [112]: RWHCA and WI are summarized in Section 5.3.2. For these policies, we assume the STTRAM partition to be RDE prone.
- **P1** (Proposed policy): Skipping restore on write Hit (SKIP_WH)
- **P2** (Proposed policy): Migrating read-intensive blocks to read-buffer (RD_BUF).
- **P3** (Proposed policy): Skipping restore of zero data blocks (SKIP_ZERO)
- **P4** (Proposed policy): This includes a combination of SKIP_WH, RD_BUF and SKIP_ZERO

In this chapter, the comparison analysis of proposed techniques with existing techniques. This would give a fair comparative analysis of Volatile-STTRAM with Non-Volatile STTRAM. In addition, we have shown comparative analusis over one of the compression technique(i.e. BDI-compression).

5.5.1 Energy Saving

Total Energy: Fig. 5.8 shows the normalized total energy savings over the baseline HCRR. On average, we get 15.41% energy savings in our SKIP_WH policy. Since SKIP_WH policy leverages the write hit operations, it is especially beneficial for write-intensive applications. The RD_BUF policy saves 21.01% energy by serving read requests from the SRAM buffer. The SKIP_ZERO policy saves 21.27% energy. The savings provided by this policy depends on the frequency with which read-hits happen to zero-value blocks. This factor varies across applications. The final combined policy, CORIDOR, gives 31.62% savings in total energy over baseline. Further, the RDE-free STT-RAM cache (Ideal) achieves an energy saving of 42.76%. Note that, in x264, the combined policy gives higher savings than RDE-free cache because the combined policy also uses optimization of not writing zero-blocks to the LLC, which is not adopted by the RDE-free cache. From Fig. 5.6, we note that the percentage of ZDBs is highest for x264 and dedup. Correspondingly, SKIP_ZERO policy provides the highest savings for these two workloads.

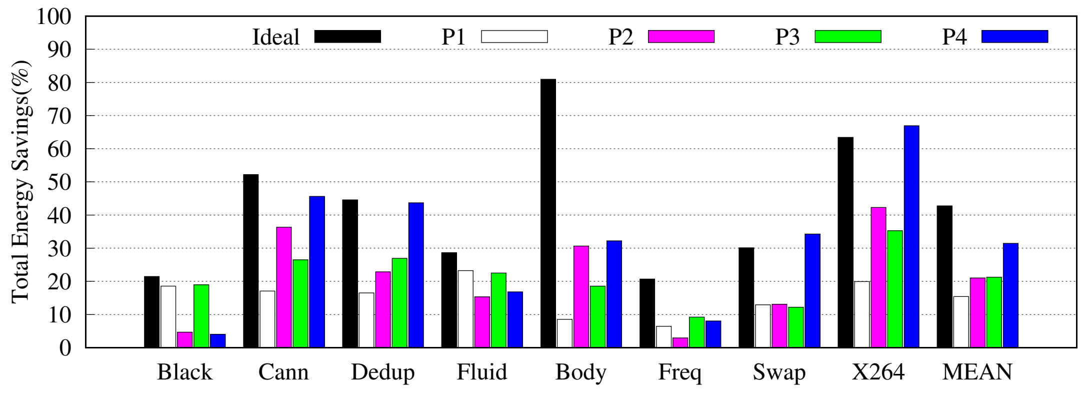


Fig. 5.8 Total energy savings for different benchmarks over the HCRR policy (higher is better)

Restore Energy: Fig. 5.9 shows the savings in the energy of the restore operations over the baseline HCRR. SKIP_WH saves 17.24% energy on average. RD_BUF policy saves 22.14% restore energy. The SKIP_ZERO policy saves 14.95% energy, whereas

CORIDOR saves 51.15% restore energy. Both SKIP_WH and SKIP_ZERO help in reducing the migration overhead of the RD_BUF policy.

Fig. 5.9 also shows the comparison with hybrid cache policies RWHCA [111] and WI [112]. RWHCA saves 17% and WI saves 34% restore energy compared to baseline HCRR. We present a detailed comparison with hybrid caches in Section 5.5.5.

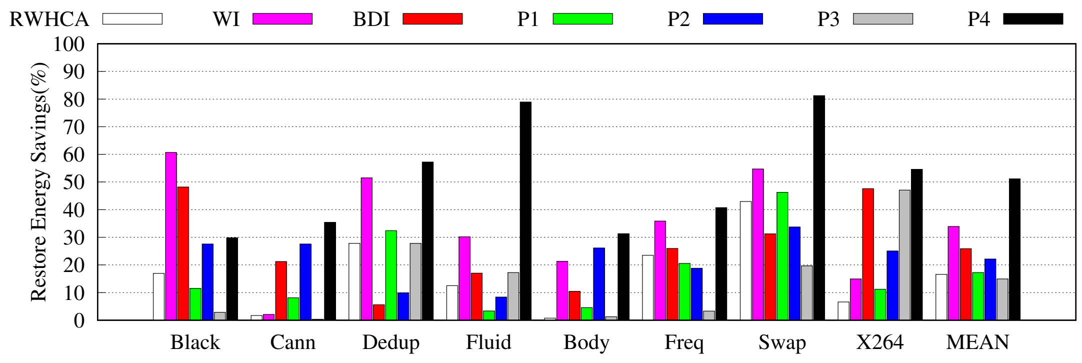


Fig. 5.9 Restore energy savings for different benchmarks over HCRR (higher is better) (RDE-free cache saves 100% restores and hence, is not shown)

5.5.2 Performance

Fig. 5.10 shows the normalized cycle per instructions (CPI) with respect to baseline. A restore operation after a read operation stalls the cache access. By virtue of reducing the number of restore operations, CORIDOR brings a significant reduction in each bank's total stall duration. The SKIP_WH brings relative CPI to 0.87×. RD_BUF brings the relative CPI to 0.64× since it serves many requests from the SRAM buffer and SRAM read latency is lower than the STT-RAM read latency. The SKIP_ZERO policy brings it to 0.85×, and the CORIDOR(P4) policy brings the relative CPI to 0.64×. The ideal RDE-free LLC reaches a relative CPI value of 0.62×. *Noticeably, our CORIDOR(P4) policy achieves nearly the same performance as an ideal RDE-free STT-RAM cache.*

5.5.3 Sensitivity of RD_BUF to Read-Threshold (RT)

Impact on read hits in the read-buffer: The read threshold decides how eagerly we migrate read-intensive data blocks to the buffer. A lower value of the read threshold makes the RD_BUF policy proactive in migrating blocks to the SRAM buffer. An RT value higher than CRead would be ineffective in detecting the read-intensive blocks. We did not choose RT=2 since a too small value of RT would lead to a high amount

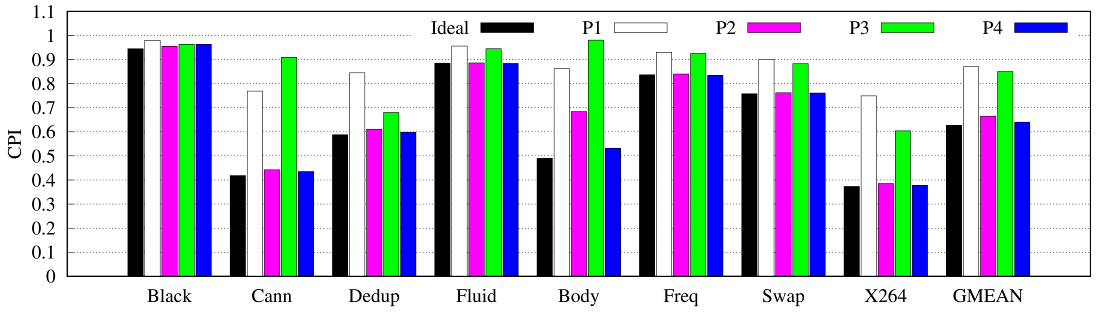


Fig. 5.10 Relative CPI values with different policies, normalized to HCRR (lower is better)

of migration overhead. The read-buffer uses an approximate-LRU replacement policy, whereby the LRU-victim block is removed from the 25% of the least recently accessed entries in the buffer. This avoids starvation in the buffer.

As shown in Fig. 5.11, For RT values of 3, 4, 5 and 8, the average values of buffer-hit percentages are 10.3%, 12.2%, 12.8%, and 13.7%, respectively. In general, the buffer-hit percentage tends to saturate after RT=4, such that using RT=8 provides only marginal improvement. In fact, at RT=8, the main STT-RAM cache sees a higher number of reads and restore operations. Evidently, higher values of RT are not beneficial. Therefore, an RT value of 4 (or 5) can be taken for all the benchmarks. Notice that for swaptions, for RT=4, nearly 28% hits happen from the buffer.

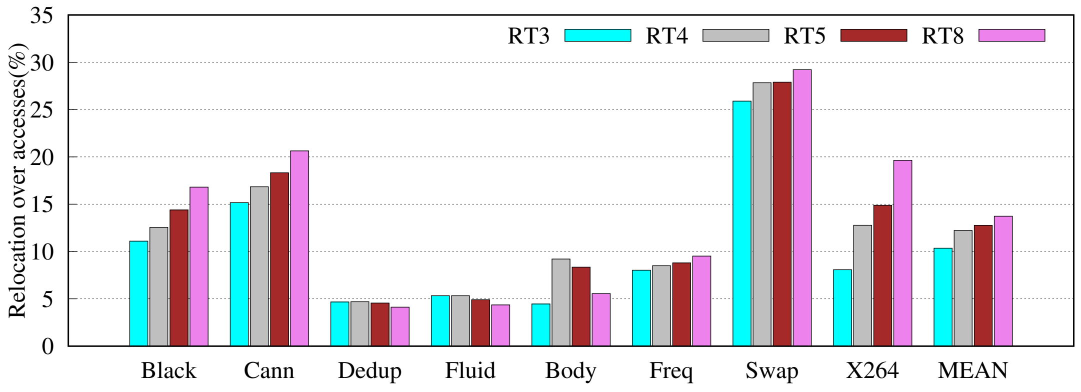


Fig. 5.11 Percentage of read-hits served by read-buffer for different RT values for RD_BUF policy (higher is better).

Impact on migrations to the read-buffer: Fig. 5.12 shows the migrations for different read threshold (RT) values. At RT=3, 4, 5 and 8, the percentage of migrations over total accesses is 12.7%, 8.9%, 6.9%, and 4.2%. With the increasing value of RT, the migration overhead is reduced. This is because with increasing RT, there is an increasing

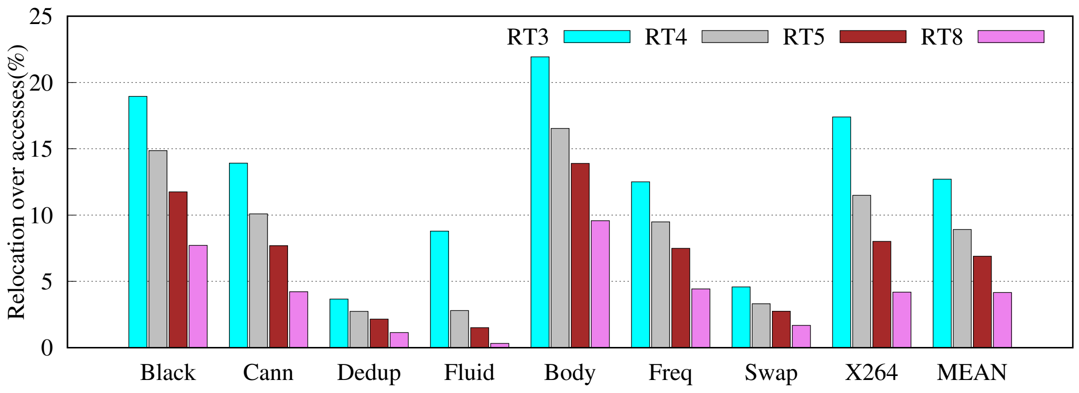


Fig. 5.12 Percentage of migration to the read-buffer over total accesses for RD_BUF policy (higher is better).

delay in declaring a block as read-intensive. Since we keep a read counter with each LLC block, using a higher value of RT increases the number of bits required for storing this counter. For smaller values of RT, the number of migrations to the read-buffer is high. However, this also increases the number of reverse migrations to the LLC and reduces the number of read-hits captured while the block is in the read-buffer (refer Fig. 5.11). Given these tradeoffs, an RT value of 4 or 5 can be considered optimal.

Impact on performance and energy: Figure 5.13 shows the CPI results of RD_BUF policy for different values of RT. Firstly, RD_BUF saves energy for all the values of RT. At RT values of 3, 4 and 5, the normalized CPI values are 0.67, 0.68 and 0.70, respectively. However, at RT=8, the normalized CPI becomes 0.84. Thus, the performance degrades drastically at RT=8, because a late declaration of read-intensive block results in more reads (and restores) from the cache and fewer reads from the read buffer. For large RT, the STT-RAM cache serves more reads (and restores), adding to the latency overhead. This affects the performance. Hence, the performance tends to move closer to that of HCRR. At very small RT values (RT=2), there are more reverse migrations leading to latency overhead. An RT value of 3 to 5 balances these tradeoffs, and hence, it is considered beneficial.

As shown in Figure 5.14, RD_BUF policy provides energy saving of 21.15% and 21.01% at RT=3 and RT=4, respectively. For a higher value of RT, more reads and restores take place within the STT-RAM before they are migrated to the read buffer. This reduces the energy-saving at RT=5 and RT=8. In fact, the energy saving is lowest at RT=8, which is due to the inability of the read-buffer to absorb a majority of the reads.

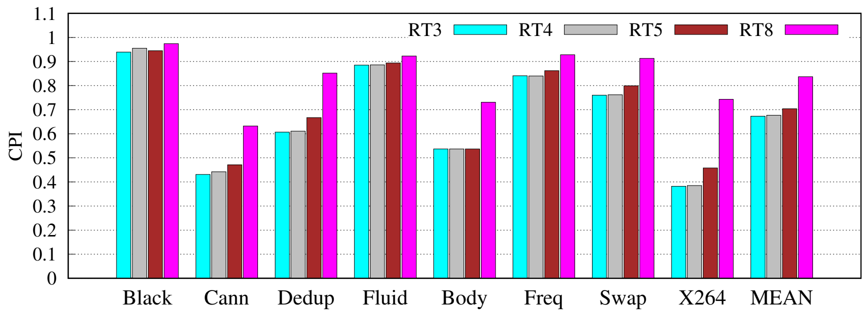


Fig. 5.13 Relative CPI values over different read-threshold values, normalized to HCRR for RD_BUF policy(lower is better).

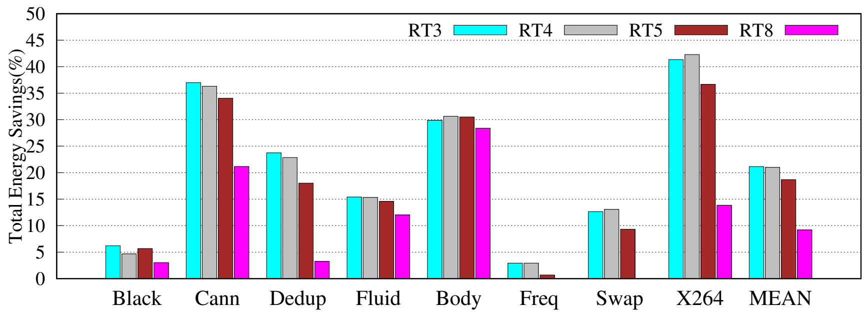


Fig. 5.14 Total energy savings for different benchmarks over various RT values normalized to HCRR policy for RD_BUF (higher is better)

5.5.4 Sensitivity to Read-Buffer size

The size of the read-buffer plays a crucial role in determining the overall performance and overheads. We experimented with four buffer sizes: 16, 32, 64, and 128 entries. As shown in Fig. 5.15, a smaller buffer size reduces the percentage of read-hits from the buffer. This increases the traffic between LLC and the read-buffer and thus, degrades the energy efficiency. In particular, the reads served from the read buffer for the above sizes are respectively 27%, 41%, 58%, and 73%. If the buffer size is very small (e.g., 16 or 32 entries), it results in more pre-mature block evictions from the buffer. This is evident from Fig. 5.16, which shows the percentage of reverse migrations over total accesses incurred by the different sizes of the read-buffer. Evidently, the 64-entry buffer performs better than a 16-entry and a 32-entry buffer, while the 128-entry buffer is the best. The number of pre-mature evictions (i.e., reverse migrations) reduces from 71% in 16-entry to 58%, 41%, and 16% respectively for 32, 64, and 128-entry buffers. However, it is noteworthy that a smaller SRAM buffer incurs lower area and leakage energy overhead compared to a large buffer. Therefore, increasing the read buffer size beyond a point may provide diminishing returns.

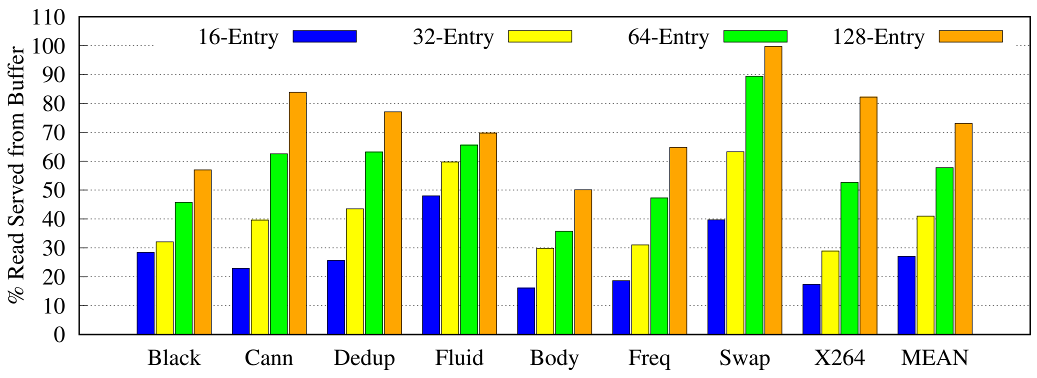


Fig. 5.15 Percentage of reads served by the read buffer for different buffer sizes for RD_BUF policy.

Figure 5.17 shows the performance normalized over HCRR. Intuitively, a larger SRAM buffer can serve more reads, and hence, it improves the performance. Specifically, the performance improvement with buffers of size 16, 32, 64, and 128 are 20%, 28%, 32%, and 42%, respectively. This performance improvement comes at the cost of increased leakage energy consumption of the larger SRAM buffer. Figure 5.18 shows the energy savings for different buffer sizes. Specifically, at buffer sizes of 16, 32, 64, and 128, the energy savings are 20%, 21%, 20% and 15%, respectively. Evidently, the performance advantage of the 128-entry buffer is offset by its area and leakage power

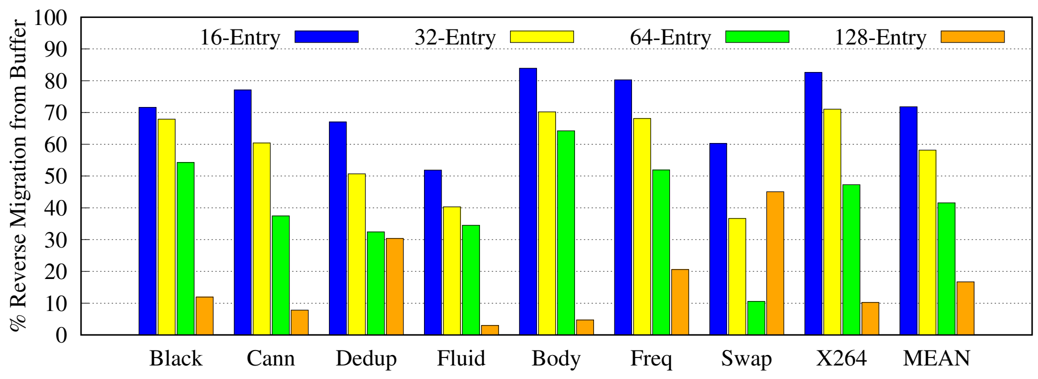


Fig. 5.16 Percentage of reverse migrations (block eviction) from the read buffer for different buffer sizes for RD_BUF policy.

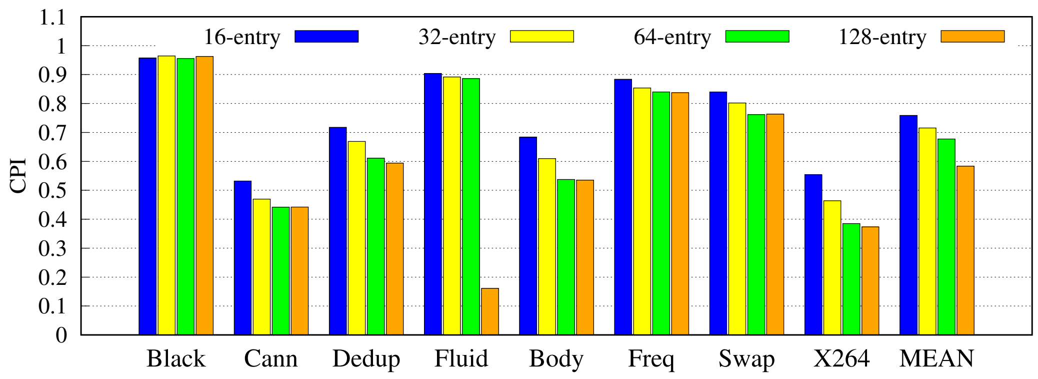


Fig. 5.17 Relative CPI values over different buffer sizes, normalized to HCRR for RD_BUF policy (lower is better).

overheads. Thus, a 64-entry buffer balances these tradeoffs, and hence, we use a 64-entry buffer.

5.5.5 Comparison with hybrid cache management techniques and a cache-compression technique

CORIDOR uses a separate SRAM read-buffer to absorb the read accesses coming to the read-intensive blocks and thus, reduce the number of restores incurred by RDE-prone STT-RAM caches. By contrast, the SRAM-STTRAM hybrid cache uses SRAM as part of the cache [117]. Given the low density and high leakage power consumption of SRAM, way-level hybrid caches use a few SRAM ways and several STTRAM ways.

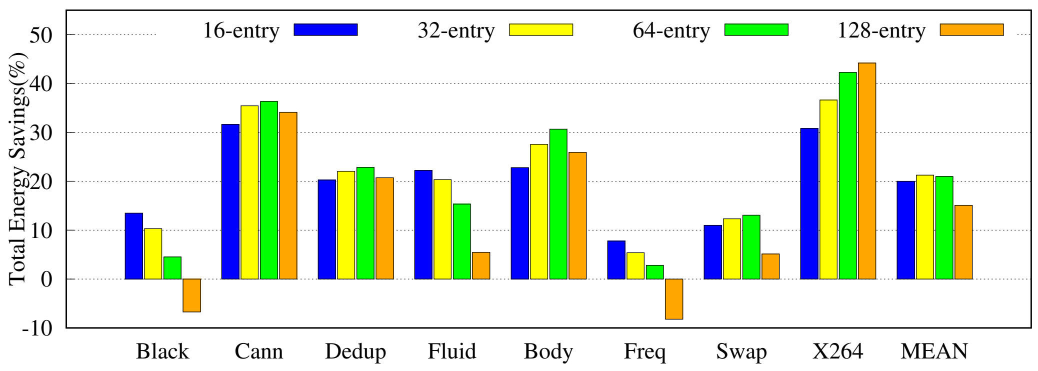


Fig. 5.18 Total energy savings for different benchmarks over various Buffer Sizes normalized to HCRR policy for RD_BUF (higher is better)

The hybrid cache management techniques seek to move write-intensive blocks to the SRAM partition to avoid the longer write latency and energy of STT-RAM partition.

We now present the comparative evaluation results between CORIDOR and two well-known hybrid cache management techniques, viz., RWHCA [111] and WI [112]. Both these techniques seek to migrate write-intensive blocks to SRAM ways. We assume that the number of ways of SRAM and STT-RAM are in the ratio of 1 is to 4. As the focus of our work is on mitigating RDEs, we assume that the STT-RAM partition is RDE-prone and hence, all the reads to STT-RAM require a restore operation. Thus, RWHCA and WI are also incurring RDE based restore overheads, therefore we terms them as RWHCA-with-RDE and WI-with-RDE in the tables below. We show the effectiveness of the SRAM partition in reducing the number of restores as well as the impact on performance and energy consumption. Since the hyrbid cache uses a very large amount of SRAM compared to our 64-entry SRAM read-buffer, we normalize the hybrid cache results to the baseline pure-SRAM cache instead of the STT-RAM based cache. All cache management methods using STT-RAM aim to achieve performance closer to SRAM and save maximum energy. We therefore show the performance and total energy normalised with baseline pure-SRAM based cache.

Table 5.3 shows the relative degradation in performance of various policies with respect to the pure-SRAM baseline cache. It can be observed that the hybrid cache management techniques bring large degradation in the performance by virtue of moving read-intensive data to the STTRAM partition. Specifically, on average, RHWCA and WI bring 28.7% and 24.7% degradation in the performance. By contrast, CORIDOR degrades by 8% over SRAM as we avoid restores and it reaches closer to ideal-STT which degrades performance by 5.8% over SRAM. Thus, CORIDOR's approach of avoiding restore operations avoids nearly all the performance loss arising from the use of STTRAM.

A key advantage of our read-buffer is that it requires smaller SRAM capacity than that required in the SRAM-STTRAM hybrid cache. Also, a separate SRAM buffer is easier to design than a SRAM-STTRAM hybrid cache. Depending on the need, the SRAM buffer can be easily deactivated.

Table 5.3 Percentage performance degradation of various policies w.r.t SRAM.

Policy	**Black**	**Cann**	**Dedup**	**Fluid**	**Body**	**Freq**	**Swap**	**X264**	**MEAN**
Ideal-STT	5.82	7.04	4.37	0.63	10.25	5.72	0.97	12.67	5.86
RWHCA (with RDE)	3.74	60.41	30.92	3.25	48.41	6.27	10.67	91.73	28.7
WI (with RDE)	1.58	60.21	20.71	2.59	38.32	5.28	8.46	83.76	24.71
CORIDOR	7.83	11.45	6.03	0.57	19.83	5.42	1.32	14.44	8.19

We also evaluate the efficacy of a cache-compression technique in mitigating restore operations. Specifically, we evaluate "base-delta immediate" (BDI) [116], which is a well-known cache compression technique. To separately compress both narrow and wide values, it uses two base words where one base is always set to zero. It compresses the block by storing only the difference of each word with the base word. Compression reduces the number of bits that are read/written and hence, it reduces the number of bits restored. This saves both restore energy and the total energy. The original BDI proposal [116] used cache compression to increase the effective cache capacity and thus, achieve better performance. Since cache capacity increase is orthogonal to our work, we used BDI only for saving restore operations and not for increasing the cache capacity. Hence, we do not present performance results with the BDI technique.

Table 5.4 shows the restore energy savings compared to the baseline HCRR policy. By virtue of using compression, BDI reduces the number of bits that need to be restored and this translates into restore energy saving. Compared to the RWHCA policy, the WI policy relocates a larger number of frequently accessed (write as well as read) blocks to the SRAM partition. Hence, WI is able to save more restore operations than RWHCA. Also note that, both hybrid cache polices are able to save on restore energy because one-fourth of the cache is not made of STTRAM and hence this part is not prone to RDE issue. Whereas our proposal uses complete cache of RDE prone STTRAM and yet it is able to save highest amount of restore energy among all policies wrt HCRR.

To compare the total energy, we use baseline as SRAM, because hybrid cache uses considerable portion of SRAM. Hybrid cache saves total energy wrt SRAM, mainly due to savings in leakage energy on behalf of the STT-RAM partition. An Ideal-STTRAM will save 51.8% energy over baseline SRAM as it uses complete cache made of RDE free STTRAM. The other RDE prone designs save total energy due to savings in leakage as

Table 5.4 Percentage of Restore Energy Savings by various policies w.r.t HCRR.

Policy	**Black**	**Cann**	**Dedup**	**Fluid**	**Body**	**Freq**	**Swap**	**X264**	**MEAN**
RWHCA (with RDE)	16.94	1.7	27.8	12.51	0.74	23.5	42.96	6.59	16.59
WI (with RDE)	60.69	2.07	51.51	30.19	21.29	35.85	54.73	14.96	33.9
BDI (with RDE)	48.2	21.24	5.6	17.04	10.4	25.97	31.2	47.57	25.9
CORIDOR	29.83	35.37	57.24	78.94	31.31	40.74	81.24	54.56	51.15

well as restores, however each value is lesser compared to the Ideal case. In particular, the policies: HCRR, RWHCA, WI, BDI and CORIDOR save energy by 23.5%, 34%, 35.5%, 13% and 44% respectively, over SRAM. Hybrid policies save less energy compared to CORIDOR because they have a large SRAM partition compared to a small read buffer used in our proposal.

In the context of mitigation of RDE, hybrid SRAM-STTRAM caches are in effective, as confirmed by our results. This is because the traditional hybrid SRAM-STTRAM caches move write-intensive data to SRAM and read-intensive data to STTRAM. Moving read-intensive blocks to STTRAM would aggravate the RDE issue since read-intensive blocks would incur many reads and hence, a high number of RDEs. Further, hybrid SRAM-STTRAM caches suffer from challenges of fabrication due to different characteristics of SRAM and STTRAM. Our design uses a separate SRAM buffer, and hence, it is easier to design. Further, assuming a 16-way SRAM-STTRAM hybrid cache, where one way is designed using SRAM, the total capacity of SRAM is 256KB. By contrast, our SRAM-buffer has a total capacity of 16KB (64 entries per 1MB bank, each entry of size 64B). Thus, our design can work with the smaller relative capacity of SRAM since SRAM memory has a high leakage power consumption.

5.5.6 Comparison with Selective Restore technique for a private cache hierarchy

Wang et al. [101] proposed a technique, named “selective restore”, for a 2-level cache hierarchy which uses both levels as private caches. Their technique skips the restore operations for blocks that are likely to get write operations in the future. We implemented the “selective restore” technique, our proposal CORIDOR and the baseline HCRR in a private cache setup to perform a comparative analysis. Figures 5.19 and 5.20 show the comparative results.

Compared to HCRR, selective-restore saves 13.69% energy whereas CORIDOR saves 23.89% energy. Also, the normalized CPI with selective-restore and CORIDOR are 0.84 and 0.64, respectively. Thus, CORIDOR provides higher performance than

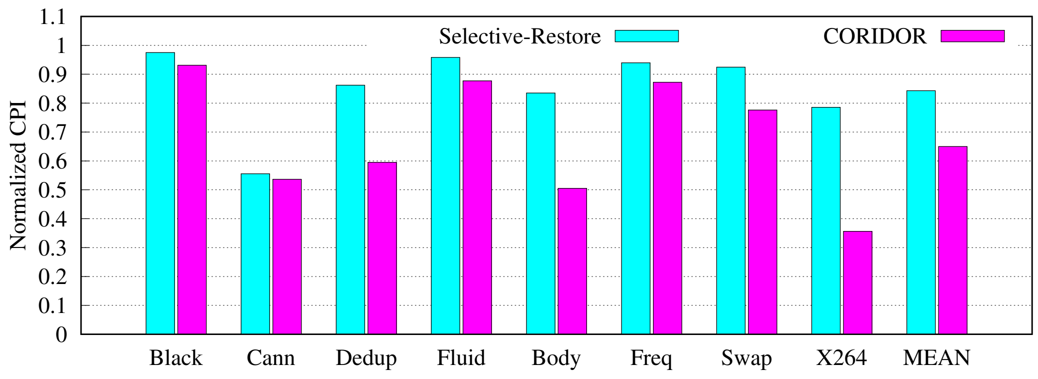


Fig. 5.19 Relative CPI of "selective restore" and CORIDOR normalized to HCRR policy (lower is better).

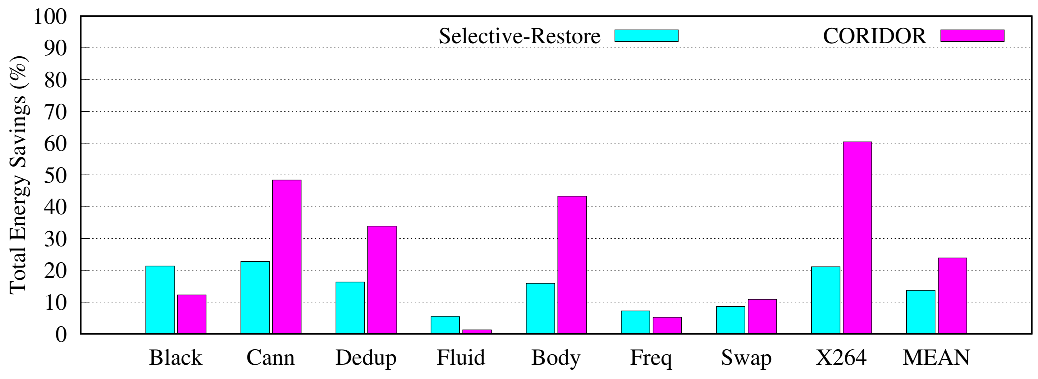


Fig. 5.20 Total energy savings of "selective restore" and CORIDOR normalized to HCRR policy (higher is better).

selective-restore. CORIDOR takes into account coherence messages and intelligently declares blocks as read intensive to migrate them to a read buffer. This, along with zero-block detection, helps in more effective mitigation of restore operations. This further translates into higher performance and energy savings.

5.6 Overhead Analysis

Latency overhead: In our proposed CORIDOR policy, read-intensive blocks are migrated to the read-buffer. This migration saves the restore for the corresponding read for the block. However, it incurs a write in the SRAM-based read-buffer. During this migration, if the read-buffer is full, we need to evict a victim. Eviction of the victim from the read-buffer relocates the block to the LLC. This incurs one STT-RAM write operation. During this relocation of blocks, the LLC is stalled, which accounts for the latency overheads. However, this overhead is compensated by the savings in the restore operations for subsequent reads to read intensive blocks, as these reads will be served from the read buffer. The relocation needs SRAM write and read operations (4 cycles) and an STT-RAM write operation (21 cycles). These overheads are taken into account in our experimental evaluation.

Note that the occasions of relocation are very few compared to the read-hit enjoyed by the read-buffer, which saves the restores in the LLC. Figure 5.21 shows the energy and latency values for restore operations and migrations incurred by our proposed policy with respect to the baseline HCRR policy. Notice that while the HCRR policy incurs a restore operation on each read operation (i.e., a restore overhead of 100%), the P4 policy needs a restore only for 48.8% reads, with an additional 9.6% writes for migrations. This is shown in Figure 5.21. Thus, there is an overall saving in latency in our proposed policy. Clearly, the overhead of migrations gets compensated by the savings in restore operations.

Energy overhead: Our technique incurs energy overhead for migrations from STT-RAM to read-buffer, reverse-migrations from SRAM to STT-RAM when the buffer is full, and the leakage energy of the SRAM read-buffer. A migration requires STT-RAM read and SRAM write. A reverse migration requires SRAM read and STT-RAM write. However, the savings in energy due to restore operations is much larger compared to this small overhead. In particular, the restore energy of 100% in HCRR is reduced to 48.8% in the proposed policy, with an additional 7.8% energy incurred as the migration overhead. This is shown in Figure 5.21. Thus, there are overall energy savings in our

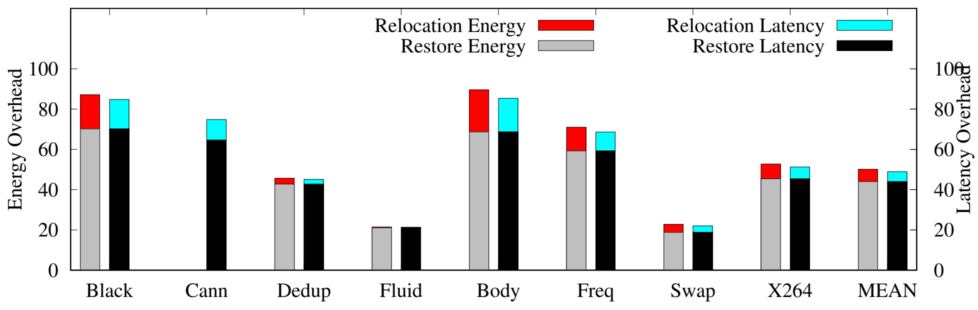


Fig. 5.21 Percentage of energy overhead and latency overhead of CORIDOR(P4) policy over the baseline (HCRR).

proposed policy. Clearly, the overhead of migrations gets compensated by the savings in restore operations.

Storage overhead: A 1MB 16-way LLC bank with 64B block has 1024 sets. We assume 42b wide tags. For each 1MB LLC bank, our technique incurs the following overhead: (i) a location bit, a zero-indicator bit (ZIB), and a 2-bit read counter for each data block (ii) a (42b + 64B)-size swap buffer used to migrate the tag and data block between LLC and read-buffer. (iii) an SRAM read buffer with 64 entries, each of size (42b+64B). In total, there is 12.4KB additional storage, which is only 1.21% overhead for each 1MB bank.

5.7 Summary

High density and low leakage make STT-RAM-based caches a lucrative option over power-hungry SRAM designs. However, at lower technology nodes, the STT-RAM designs suffer from RDE in which the reads cause the data to be corrupted. To mitigate this issue, after each read, the block has to be re-written or restored to preserve the data content. This additional restore operation, a write operation, leads to stalling of cache bank and consumes dynamic energy. This work proposed techniques to mitigate the RDE in STT-RAM-based LLCs by using properties of the coherence protocol, read-intensity of blocks, and identification of blocks with zero data content.

The first proposal, SKIP_WH, exploited the cache coherence protocol and skipped the restoration of blocks fetched for modification by the ULC (i.e., blocks incurring write hits). Our second proposal, RD_BUF, identified the read intensity of each block by associating a read counter and comparing it against a read threshold. Such blocks were relocated to an SRAM-based read buffer, and subsequent read accesses to such blocks were serviced from the read buffer. As these reads were done from the buffer, the restore operation to the STT-RAM LLC was unnecessary. The third proposal, SKIP_ZERO, identified blocks that have zero data content and associated this information in the meta-data. Such blocks were never written, and on a read operation, they were reconstructed before sending to the requester. Blocks having zero data content were also never restored after being accessed. All three proposals use different properties of the block access pattern to avoid restore operations. Depending on the application characteristics, a specific solution may work for one over the other. These proposals are complementary to each other and can be combined to give bigger benefits in restore savings. We proposed to combine all three proposals into the final policy 'CORIDOR'.

The proposals were compared with baseline policy HCRR that performed a restore on every read and with an ideal RDE-free STT-RAM cache. The CORIDOR policy gave 31.6% savings in total energy over HCRR, whereas an ideal RDE-free STT-RAM cache gives 42.7% savings. Thus, our proposal achieves three-fourths of the energy-saving achieved by the ideal cache. The proposal also saved 51% energy incurred by the restore-operations by HCRR. Savings in restore operations also reduced the stall time for the cache banks. Further, it reduced execution time to 0.64 times the HCRR, which is almost the same as that of the ideal STT-RAM cache.

We performed a sensitivity analysis on the different values of read-threshold (RT) and read-buffer sizes. It was seen that there is a trade-off between having a higher read threshold and the effective read hits captured by the read buffer. For lower RT values, the migrations to and from the read-buffer increase, thus leading to more overheads for block relocation. An optimal value of RT will move the block to read buffer such that the block incurs most of its read hits in the read buffer before getting relocated back to the LLC. It was experimentally found that the value of RT as 4 gave the best results and that higher values gave diminishing returns in terms of read-hits. It was also observed that the size of the read-buffer should be small enough to enable faster access latency and big enough to reduce capacity misses. A buffer size with 64 entries was found to give a decent hit rate.

The work has thus demonstrated that the properties of coherence and temporal locality of read-intensive blocks can be used in conjunction to avoid the restore operations resulting from the RDE issue of STT-RAM. This is a step towards designing STT-RAM-based, error-free, energy-efficient replacements for SRAM-based last-level caches.

CHAPTER 6

CAPMIG: Coherence based Refresh Optimization in Multi-retention STT-RAM caches

The issue of refreshing arises in low-retention STTRAM caches. In order to reduce the write energy, the volume of the cell is reduced which leads to reduction in the longevity of the stored content. After a certain time interval the content leaks leading to deletion of information. Solution to this is to refresh the expiring data cells within their retention interval. Multi-retention STTRAM come with variety of retention intervals and lead to significant number of refreshes to data. However, data blocks in a cache usually get written/updated leading to an automatic refresh. If we are able to place blocks in a retention window same as its write behaviour, we will be able to avoid refreshes to such blocks. Towards this aim, we propose to use multiple retention intervals based STTRAM cache and use coherence messages to decide where to place an incoming cache block. This will reduce the number of refreshes compared to random block placement.

6.1 Introduction

STT-RAM uses magnetic-tunnel junction (MTJ) which stores the value using the magnetic orientation of the ferro-magnetic material making them non-volatile over a longer duration. If the surface area of the cell is reduced, it can operate faster using lesser write latency and energy [76] with the consequence of lower retention time. As the retention time reduces, to maintain the data integrity we need to either write-back the data block or refresh the contents. A refresh includes the block being written to a buffer and then re-written into the cache [78], popularly called a dynamic refresh scheme (DRS) [76]. Refreshing incurs extra performance and energy overheads due to bank stalling as well as additional refresh energy [118]. Prior research to solve this problem have either used hybrid SRAM-STTRAM caches [119] or multi-retention STTRAM caches [120] where the cache is divided into partitions, each using a different retention time. The problem of refresh or writeback is addressed by migrating the block to higher retention partition or to the SRAM partition. As the hybrid caches using SRAM still incur relatively higher leakage, in this chapter we focus on multi-retention caches.

Existing approaches on multi-retention STTRAM, like HALLS [78] trains an application using banks of different retention times and choose the best retention time for the remaining execution. However in case the applications profile changes over execution, then the blocks need to be relocated to appropriate partitions depending on their current access profile. Such decisions also require overhead of counters associated with the blocks to keep track of their write access patterns as well as bulk migration of blocks when they change their retention behaviour. Also note that once the retention regions are decided, and if certain blocks cross their retention interval, then such blocks are written back to the next level memory. Whereas, certain other research contributions advice to refresh the expired blocks. In this chapter we choose the latter and refresh the blocks upon expiry. However, our aim is to place the block in the best retention region so that we need not refresh it often. Towards designing methods to identify the correct retention region for blocks without bookkeeping their access patterns, we use the following observations:

Firstly: Cache coherence protocols in shared memory systems help us identify the type of block requests. We can use this information to predict the best retention time for certain types of requests. Secondly: once a type of retention region is selected, the block can remain in that region for a longer duration on account of spatial and temporal locality. In that, a block which gets a ‘write’ is likely to get further write requests before it converts into a read intensive block.

The chapter makes the following main contributions:

- For a multi-retention STTRAM caches we propose a placement method using coherence protocol for newly loaded blocks.

- We propose a migration policy based on coherence protocol to relocate blocks within the retention regions.

- In addition, we augment our policy with zero data detection logic and avoid refresh for such data blocks.
- We perform experimental evaluation on full system simulator gem5 and show that our proposal give signification improvement in saving the number of refreshes.

6.2 Background and Related Work

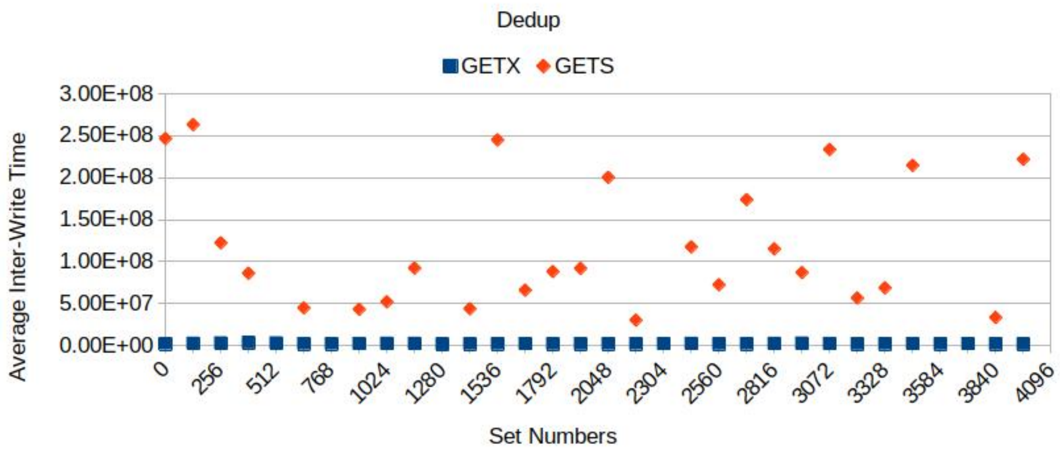


Fig. 6.1 Percentage of read-hits served by read-buffer for different RT values for RD_BUF policy (higher is better).

6.2.1 Multi-Retention STTRAM

The volume (cell area) of the MTJ device is reduced in order to increase writability by reducing the retention period of the STT-RAM cache cells. The lowering of size of the free layer of the magnetic tunnel junction (MTJ), which is the storage element for STT-RAM, decreases the energy required to write the cell, resulting in shorter retention periods [120]. A simple DRAM-style refresh approach is also available to verify the data's correctness. STT-RAM caches with shorter retention periods have better performance and energy efficiency. However, any possible accuracy difficulties that may arise as a result of random bit-flips have been overlooked. The use of single-bit error correction simplifies the creation of a refresh-like operation that reads the contents of a line and then sends it back after error correction. As previously noted, the non-

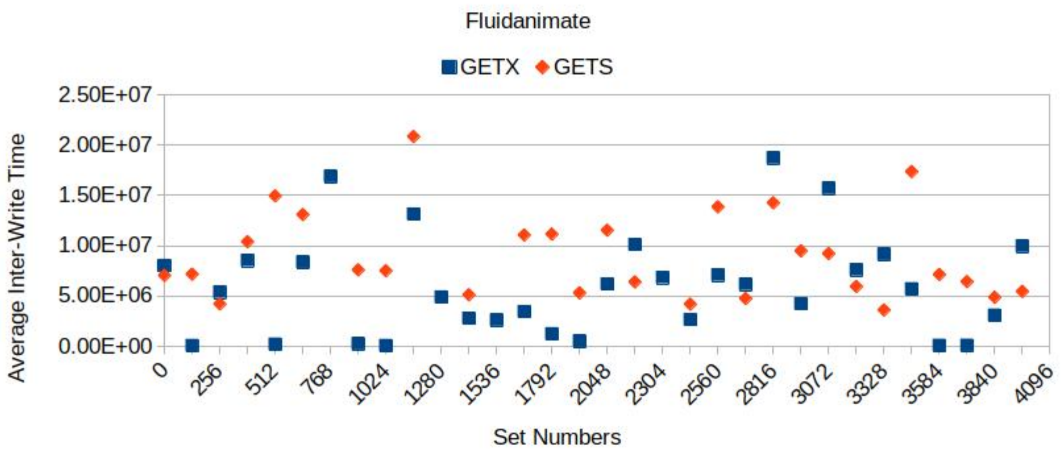


Fig. 6.2 Percentage of read-hits served by read-buffer for different RT values for RD_BUF policy (higher is better).

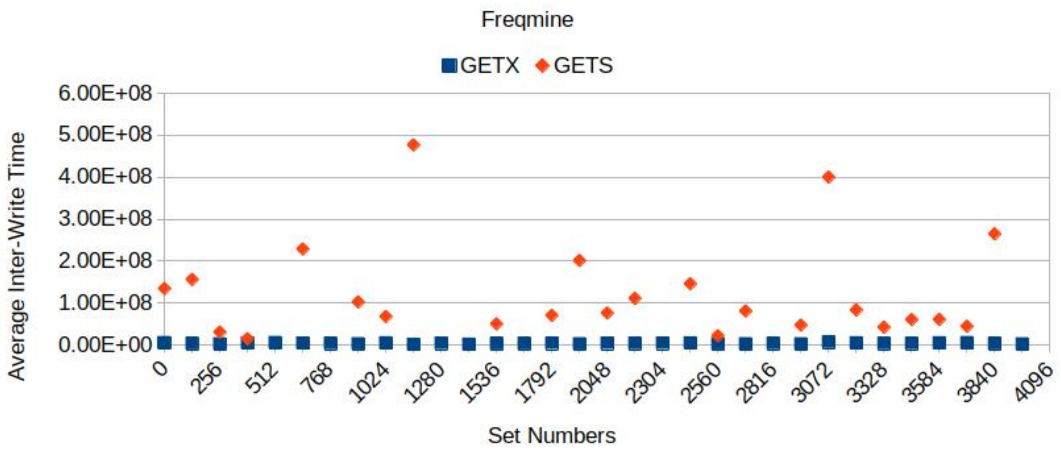


Fig. 6.3 Percentage of migration to the read-buffer over total accesses for RD_BUF policy (higher is better).

destructive nature of STT-RAM readings eliminates the requirement for a writeback if no defect is discovered.

Shortening the retention time STT-RAM in which cell minimizes its write dynamic energy and latency of STT-RAM by lowering data retention time. This can be accomplished by lowering the free layer's surface area. Two variants of data retention time decreased STT-RAM is shown in reference [76]: 26.5 s and 3.24 s. STT-RAM cells are used in all cache levels in work in References [120] [76] because they have lower write dynamic energy and latency. However, in order to keep the data in their STT-RAMs, they must perform periodic refresh operations. DRAM-style Refresh was utilized in Reference [120] (refresh all cache lines). First, our idea focuses on large LLCs where dense STT-RAM-based caches plays important role. It has higher write costs than SRAM caches, which can be reduced by relaxed-STTRAM caches. Second, this work aims to lower the system's energy usage while also improving its performance. In particular write operations consumes energy in STT-RAM-based caches, a more aggressive cache partitioning strategy is required. This inspires us to investigate a STT-RAM-based LLC design with low retention levels. In order to leverage the best features of relaxed STTRAM, we use multiple retention levels in a single cache. As we have evidance of existance of hot and cold cache blocks. Thus, variablilty in data block lifetime inside LLC which changes throughout the execution. Thus gives a justified reason to use multiple retention level partitions in LLC. However, data block varies this lifetime visiblity depending on the type of application.

We presented a hybrid memory system with multiple ranges of high and low retention STT-RAM sections to meet both enhanced performance requirements at the same time. As illustrated in Fig. 6.6, we use a 16-way L2 cache where Way 0-7 of the 16-way cache is implemented with a low retention STT-RAM design, whereas ways 8-15 are built with a high retention STT-RAM design. For a quicker write response and minimal refresh operation, write intensive blocks are largely assigned/rearraged within way 0-7, while read intensive blocks are retained in the other ways.

6.2.2 Refresh Schemes

A multi-retention cache provides options for cells with different retention times in a single cache. Retention time over the cells maintained by a periodic refresh similar to the DRAM style. Each refresh operations consists of four physical operations to be performed while refreshing a block: 1) STTRAM Read, 2) Refresh Buffer Write, 3) Refresh Buffer Read, and 4) STTRAM Write. Refresh consumes a significant amount of energy and time. We allowed different applications to switch the regions on different instances, depending on the frequency of the write blocks approaching the last level.

Jog et al. [121] proposed CacheRevive, where they refresh first 8 most recently used blocks using dedicated hardware. Proposed technique copy data to a refresh buffer and copy back to the cache line. This would always be suffering from high latency overhead. Mirror Cache [122] is another approach by taking advantage of the high density of STTRAM. It augmented an auxiliary cache with the main cache. At the time of expiry, data migrates to the auxiliary cache, and subsequent accesses will be served from the extended part of the cache. Thus, it saves refresh operations over the cache lines. However, there would be a hardware overhead of the auxiliary segment. These techniques are suggested for first-level cache, while we concentrate on last-level cache in this chapter. Apart from dynamic techniques, there are some static methods with low scalability. These are compiler-assisted techniques for refresh-optimization [123] [124].

In the context of relaxed STT-RAM, the technique of Qiu et al. [124] proposed refresh mitigation technique for main memory in embedded system which can traverse loops in a new direction such that data lifespan can be reduced. The technique reduces lifespan of Memory write instruction through optimal loop scheduling vector. Thus, reduces dynamic energy and minimize overall refresh overhead.

Rabiee et al. [125] present a lifetime altering technique for upper level STTRAM caches. They eliminate redundant write operations by considering dead blocks in L1 cache. They proposed a modified L1 cache coherency protocol which reduces 60% write operations from the different sources. For this they augmented two states(M' and S') in MESI protocol. The proposed architecture also includes a 8-entry buffer which gives saving of upto 57% with respect to baseline.

6.2.3 Exploiting presence of Zero-Values

Chang [86] propose a "value-based technique" for reducing the average cache power consumption while accessing the cache. They propose a "zero-sensitive cell" that senses bit value to reduce zero values' access power. To save leakage power, Islam et al. [126] propose an architecture that considers zero loads in the applications. It uses an auxiliary zero-value cache in parallel with the L1 cache and keeps zero-value blocks in this auxiliary cache in a compressed form. Similarly, Dusser [85], provide a zero augmented cache that works in parallel with the normal cache. Its advantage over the zero-value cache is that it can be used with any cache level and not just L1 cache. Zhang et al. [127] utilize compression for reducing write traffic and mitigating "process variation" in PCM-based "main memory". We have enough literature in order to validate the presence of zero-content over the blocks from main memory. At the end of the retention time we have to refresh blocks and refresh operation on zero blocks can be avoided. In order to improvise our CAPMIG policy, we proposed policy CAPMIG with Skipping Zero(CAPMIG_SZ) an augmentation of restricting refreshing over zero blocks.

6.3 Motivation

6.3.1 Observing the Temporal Locality for Read and Writes

On account of temporal locality, the cache blocks get accessed repeatedly before eviction. During the lifetime of the block in the cache, the block shows temporal locality in its accesses. This applies to both read as well as write accesses. Figure 6.4 shows the average value of consecutive reads (CReads) [128] and consecutive writes (Cwrites) incurred by the blocks for various benchmarks. In particular, the value of CReads gives the number of consecutive read accesses incurred by the block before it incurs a write access or gets evicted. On average, in most benchmarks, each block receives 2 to 10 consecutive read accesses and 2 to 7 consecutive write accesses. We use this information to infer that once a block is placed in a particular region based on its access request (i.e. either a read or write), it will incur the same type of access over a certain period in the future.

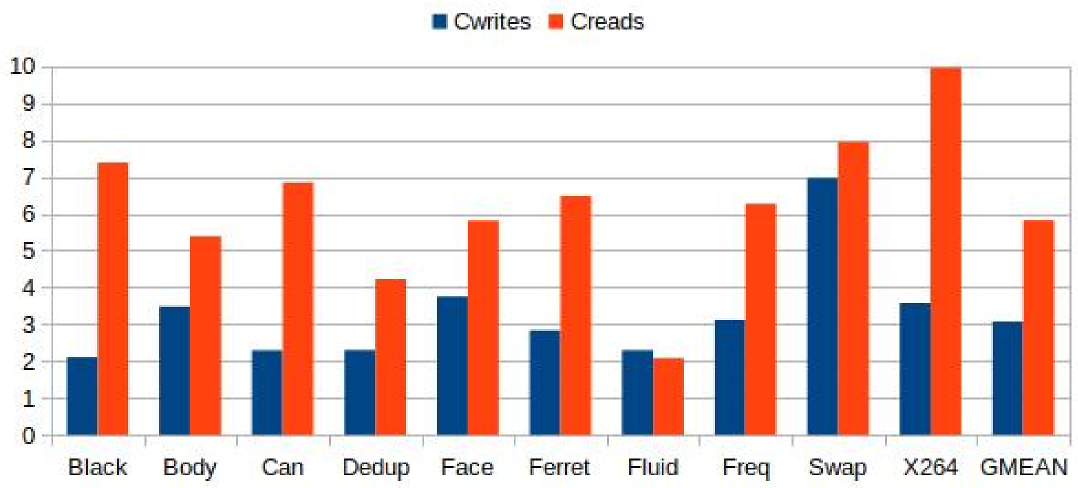


Fig. 6.4 The average consecutive reads (CReads) and consecutive writes (CWrites) on the blocks in a bank for different benchmarks.

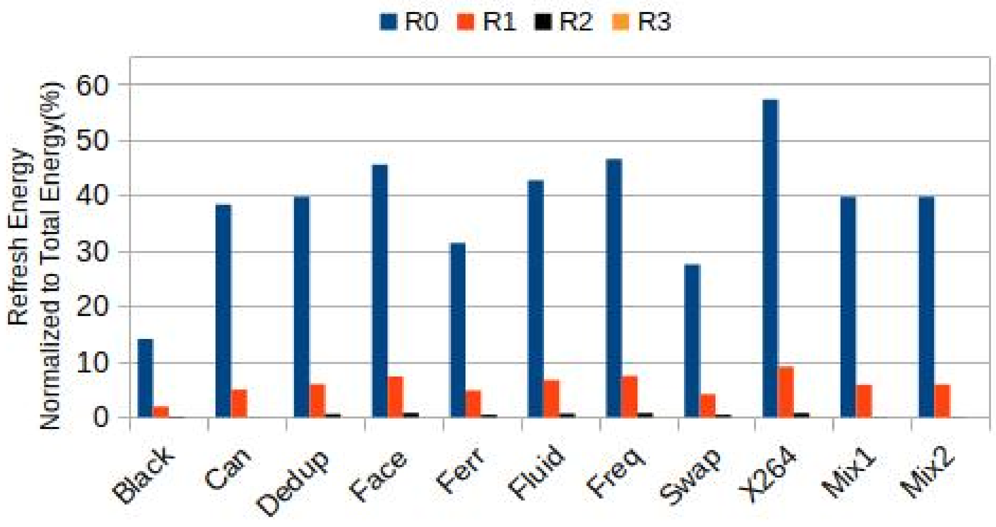


Fig. 6.5 Refresh energy percentage for different retention regions

6.3.2 Interval between Consecutive Writes

If a block is written/updated, it is treated as a refresh and hence the block need not be immediately refreshed. We note that a block loaded for writing incurs several writes before it incurs a read or gets evicted. Similarly, a block loaded on a read request will incur several reads before incurring a write or getting evicted. Figure 6.4 shows the time interval between consecutive writes to blocks across various sets (and benchmarks). The figure shows the access intervals for blocks loaded on a write request (GETX) and on read requests (GETS). It is evident from the figure that blocks loaded on write-requests incur writes more frequently compared to the blocks loaded on read requests. Blocks loaded on reads (GETS) incur a write much later after getting loaded. We use this information to infer that the access behaviour of block is dependent on the type of request on which the block gets loaded. We can use this information to decide the best retention time for block placement at the time of loading. In particular, we note that blocks loaded for writing get frequent writes and hence can be loaded in low retention regions while blocks loaded for reads (including instruction blocks) can be loaded in higher retention regions.

6.3.3 Comparing the Refresh Overhead

In a multi-retention STT-RAM cache, the blocks need to be refreshed at the end of retention time. A randomly placed block using basic LRU policy and without considering the type of access pattern will incur several more refreshes in addition to the write energy and latency. Whereas, a non-volatile STT-RAM cache needs no refreshing. Figure 6.5 shows the energy needed to refresh as percentage of total energy. It shows energy of various regions and as is evident the Low Retention Region (LRT) has the highest overhead. Thus, it is imperative to save the refreshes in order to save overall dynamic energy.

6.4 Proposed Policy: CAPMIG

This section describes our proposed block placement technique for multi-retention STTRAM Last Level Cache that reduces dynamic refreshes. The method observes the coherence messages keep track of the request type and corresponding state in the protocol. This technique prevents a block from being refreshed excessively by assigning it to the best possible region inside the cache.

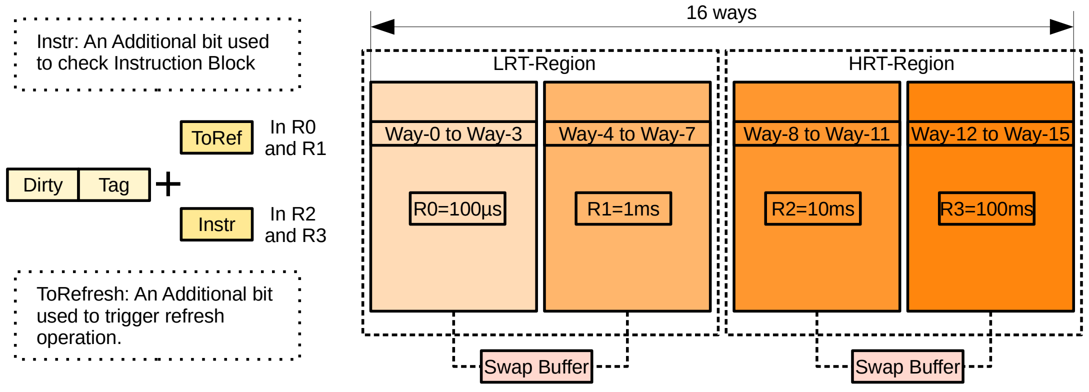


Fig. 6.6 CAPMIG: Proposed multi-retention based LLC Cache Architecture

6.4.1 Architecture

We proposed multi-retention Hybrid cache architecture for a 16-way low-level STTRAM cache. According to Figure 6.6, way-0 to way-3 is implemented with Low Retention Time STT-RAM design and way-12 to way-15 are implemented with High Retention Time STTRAM design.

In order to track the expiry status of the retention time of a cache block, we associated a timer with every block. There are 3 instances when these counters are updated: i) Initially when block allocated to LLC , ii) On each writeback, when new data updated on the block, and iii) When blocks are written after migration from another region. The timer of the blocks is lesser than the retention time of the corresponding block. When a timer reaches its final state, it triggers the refresh operation for the particular block. Note that the multi-retention baseline also uses such timers. The retention time of a cache block refers to the amount of time that data in the cache entry is valid after the most recent write. With the help of expiry counters, we keep track of blocks that are about to expire. To do so, we set the clock period to be N times shorter than the retention duration, where N is lesser than the cache retention time by a factor of N. The cache entry is invalidated and written back to main memory when the counter reaches the final state (if dirty). We have four states where we need two bits for the expiration counters.

In our cache architecture, within each cache set, we divide the cache ways into four equal-sized groups with retention times 100μs, 1ms, 10ms and 100ms respectively. Each cache block has an associated *toRefresh* bit (a.k.a *To-Ref*) that keeps track of the blocks exclusively held by the Upper Level Cache(ULC). We triggers refresh operation only for the blocks for which To-Refresh bit is set. Similarly, to place instructions exclusively in the R3 region, we use an *isIntr* bit to identify instruction blocks.

6.4.2 Block Placement

As discussed in the architecture sub-section, we have different retention regions. The leftmost (R0) is the lowest retention time (LRT) region and the rightmost (R3) the highest retention time (HRT) region. We make the following observations:

- Block residing in R0 will expire quickly due to low retention. If the blocks get frequently updated by the application then the occasions to refresh them are reduced. In particular, if a block gets written before it reaches its retention interval, then we need not refresh that block.
- Block residing in R3 will have more time before they expire. Hence even if the block is not updated at frequent intervals, the occasions of refreshing them is much less compared to the case of R0.

Exploiting the properties of coherence protocol and the above observations we arrive at the following block placement policy depending on the type of request generated by the upper level cache (ULC). The policy is described based on each type of cache request. However, at an abstract level we propose that: *blocks which are more read intensive are kept in high retention regions: R2 and R3; whereas blocks which are prone to receive writes are kept in low retention regions R0 and R1.*

For a request coming from ULC to LLC on block *B*, it could result in either a cache-hit or a miss. We list below the set of actions taken depending on the outcome for each type of request.

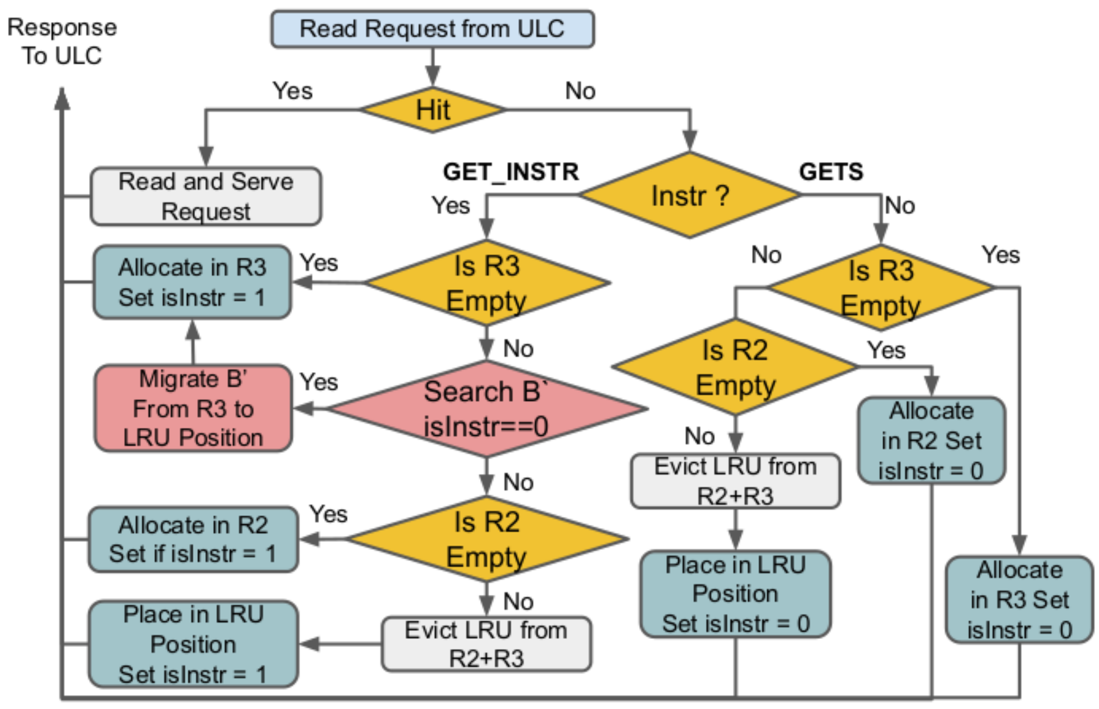

Response To ULC
Read Request from ULC
Yes
Hit
No
Read and Serve Request
GET_INSTR
Instr ?
GETS
Yes
No
Allocate in R3 Set isInstr = 1
Yes
Is R3 Empty
No
Is R3 Empty
Yes
No
Migrate B' From R3 to LRU Position
Yes
Search B` isInstr==0
Is R2 Empty
Yes
Allocate in R2 Set isInstr = 0
No
No
Evict LRU from R2+R3
Allocate in R2 Set if isInstr = 1
Yes
Is R2 Empty
Place in LRU Position Set isInstr = 0
Allocate in R3 Set isInstr = 0
No
Place in LRU Position Set isInstr = 1
Evict LRU from R2+R3

Fig. 6.7 Flow Chart of Operations performed on Read Request from ULC.

Cache Miss

Cache miss can occur on a read request for instruction, read request for data block, or write request for data-block. Each case is discussed below:

- GET-INSTR (Read-miss)
  Blocks which are loaded using GET-INSTR request are instruction blocks. These blocks will never receive any writes and hence they will need to be refreshed upon expiry. In order to reduce the number of times they need to be refreshed, we place them in the highest retention time region. The block is loaded in R3 if there is empty space. If there is no space in R3 we invoke block migration and/or LRU eviction. This is discussed later when we describe the block migration policy.

  We give priority for instruction type blocks to stay within R3 and to keep track of which block is loaded as an instruction we keep one bit with each block: *isInstr* which is true for instruction type blocks. Figure 6.7 shows the set of actions taken on a read-miss. Note that this figure also includes the actions for read-miss on data blocks.

- GETS (Read-miss)
  Blocks which are loaded using GETS are used for reading data items. Such blocks will incur reads for sometime, before incurring any writes in future. However, on account of temporal locality, we predict that GETS type blocks will have more reads in the beginning. Therefore, we choose to load such blocks in high retention region R3. If there is no space in R3 then we place the block in R2. If there is no space in R2 then we evict the LRU from the combined region R2+R3 and place the block in the LRU position. The bit indicating isInstr is set of false for data blocks. This is shown in figure 6.7.

- GETX (Write-miss)
  Blocks which are loaded using GETX request are mainly data blocks loaded for writing. These blocks will incur several writes during their lifetime in the cache. As these blocks will get frequently updated by the application and each write is similar to a refresh, it is beneficial to load them in lowest retention region (preferably R0). We also make note that the block which is loaded on GETX will be modified by the ULC and the corresponding copy in LLC is stale. In that, this copy will get updated on a writeback and hence we need not refresh as long as it is cached in the ULC. For doing this we keep an additional bit with each cache block called *toRefresh* set of false, indicating that the particular block need not be refreshed upon expiry. We change this bit to true once the block is written back to the LLC from the ULC. By default this bit is set to true for all blocks in the cache.

  The block (B) is loaded in R0 if there is empty space. If there is no space in R0 we invoke block migration and/or LRU eviction. This is discussed later when we describe the block migration policy. Figure 6.8 shows the set of actions taken on a write-miss.

Cache Hit

Cache hit can occur on a read-access, write-access, or write-back of data from ULC. Each case is discussed below:

- GETS (Read-Hit)
  Read-hits are treated normally. In that the data block is sent to the ULC. This has no impact on the refresh schedule of the block.

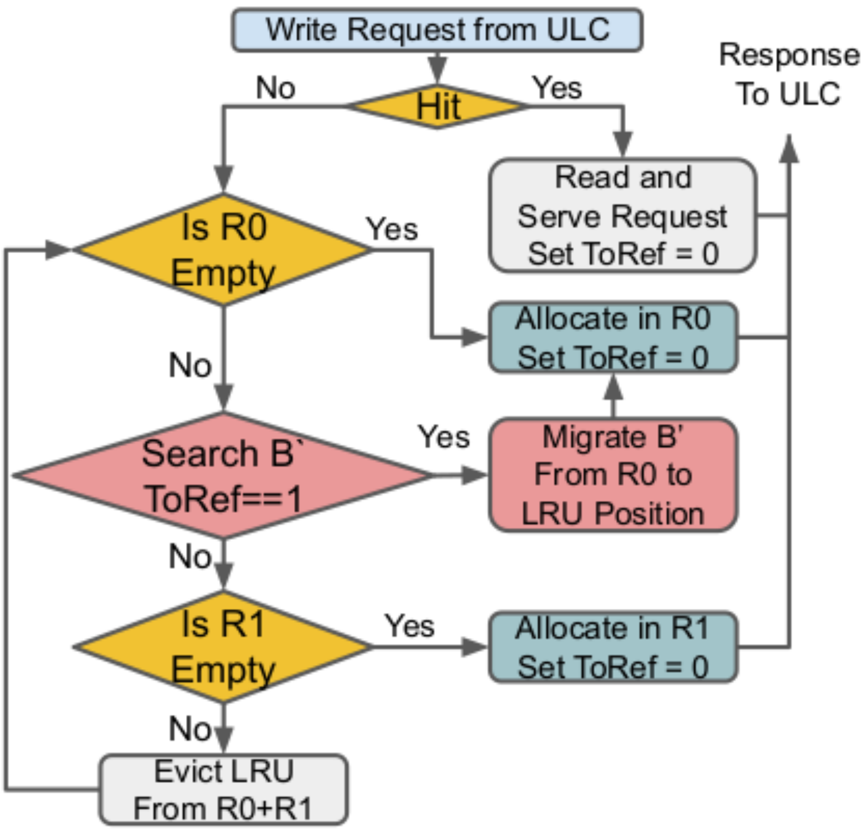

Write Request from ULC
Hit
No
Yes
Response To ULC
Read and Serve Request Set ToRef = 0
Is R0 Empty
Yes
Allocate in R0 Set ToRef = 0
No
Search B` ToRef==1
Yes
Migrate B' From R0 to LRU Position
No
Is R1 Empty
Yes
Allocate in R1 Set ToRef = 0
No
Evict LRU From R0+R1

Fig. 6.8 Flow chart of operations performed on Write Request

- PUTX (Writeback)
  When a dirty block gets written back from the ULC to the LLC we receive a request of the type PUTX. In this the data is written to the LLC and the corresponding toRefresh bit is set to true. Normally this block will be located in R0 or R1; however it is also possible that it is found in region R2/R3 as it might have been loaded earlier in R2/R3 due to a read-miss. If the block is located in R0, then we invoke migration as discussed in the next sub-section. Figure 6.9 shows the set of actions taken on a write-back.

- GETX (Write-Hit)
  When we get a write request for an already loaded block, we return the block to the ULC and set its toRefresh bit to false, (cf. figure 6.8). This will save any refreshes to the block until it gets written back from the ULC. Note that the block can be in any region when it gets a write-hit. Our policy of loading (write-miss) blocks in R0 is only during the initial load from the main memory. However,

blocks residing in R2 and R3 (that were loaded by GETS (read-requests)) can also incur write-hits in the future.

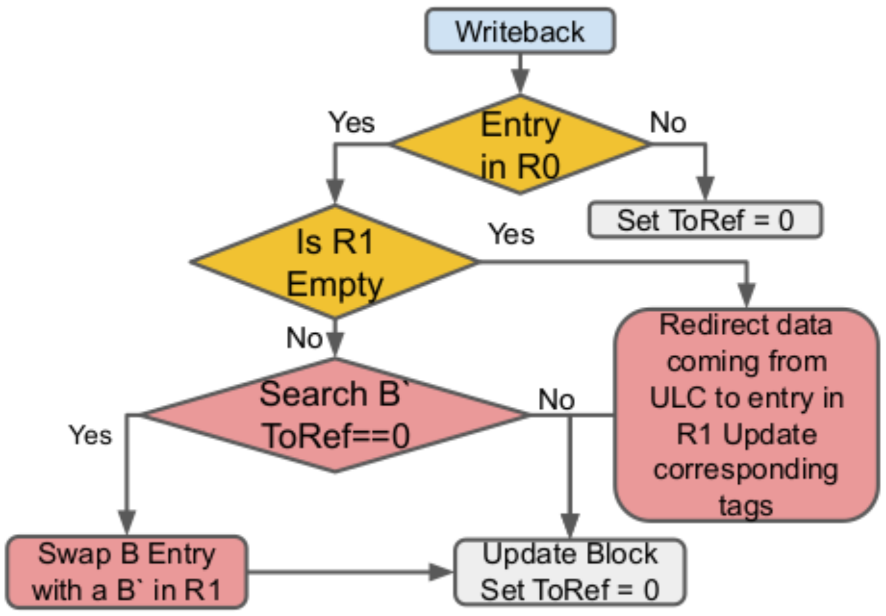

Writeback
Entry in R0
Yes
No
Set ToRef = 0
Is R1 Empty
Yes
No
Redirect data coming from ULC to entry in R1 Update corresponding tags
Search B` ToRef==0
Yes
No
Swap B Entry with a B` in R1
Update Block Set ToRef = 0

Fig. 6.9 Flow Chart of operations performed on writeback.

6.4.3 Block Migration

During the residency of the block in the LLC, there will be occasions when the location of the block needs to change from one region to another. This is mainly done to place the block in the correct region or to make room for another better candidate. Below we discuss the proposed methodology. Similar to the treatment under block placement, we discuss block migration for various types of requests and whether they are cache hit or miss.

Cache Miss

- GET-INSTR (Read-miss)
  For loading the new instruction block B our first choice is to load it in R3. Please refer to Figure 6.7. In case there is no space in R3 and there is a block B' in R3 which is not of type instruction (i.e., it has isInstr=0) we choose to migrate B' to R2 and load B in R3 in place of B'. In case all blocks in R3 are of type instruction, then load block B in R2 if there is space. If there is no space in R2, then evict the LRU block from the combined partition R2+R3 and load B at LRU.

- GETX (write-miss)
  For loading the new block B on write-miss our first choice is to load it in R0. Please refer to Figure 6.8. In case there is no space in R0 and there is a block B' which has its toRefresh=1, we choose to migrate B' to R1 and load B in R0 in the place of B'. In case all blocks in R0 have toRefresh=0, then load block B in R1 if there is space. If there is no space in R1, then evict the LRU block from the combined partition R0+R1 and load B at LRU.

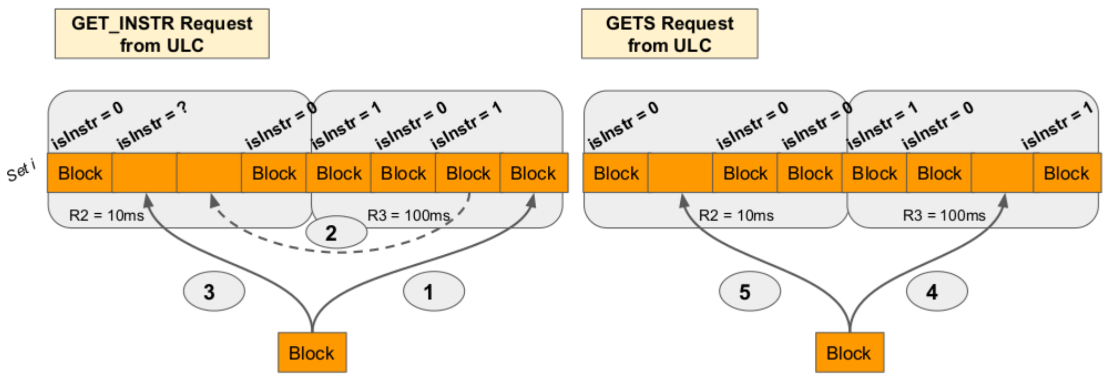


Fig. 6.10 Operations performed on GETS and GET_INSTR request from ULC.

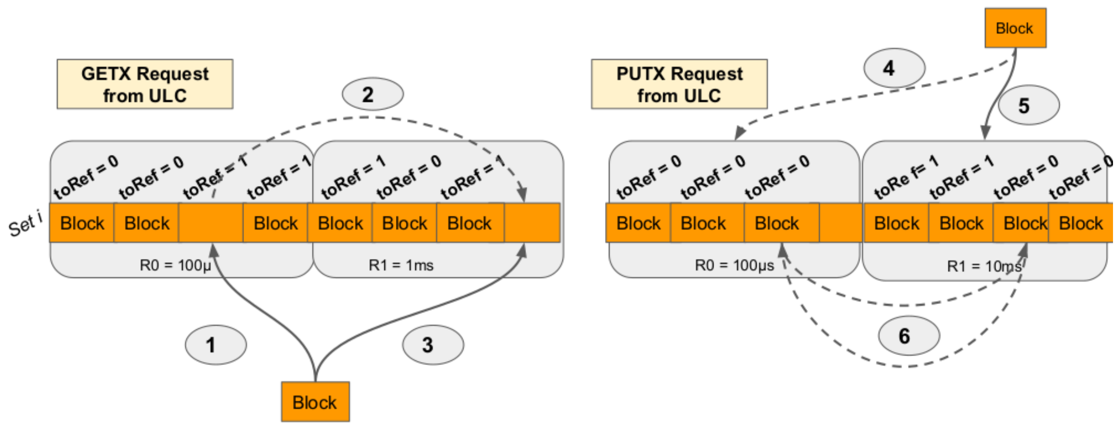


Fig. 6.11 Operations performed on GETX and PUTX request from ULC.

Cache Hit

- PUTX (Writeback)
  Please refer to Figure 6.9. If the block B being written back is located in R0 we try to perform the writeback of B in the region R1. This is done for following two

reasons: (i) as B is written back, it will have to be refreshed and hence we prefer to place it in a higher retention region; and (ii) it will enable some candidate block from R1 to get promoted to R0 if it has its toRefresh bit set to false.

If there is a block B' in R1 with toRefresh=0 then we migrate B' in place of B in R0 and writeback B in R1. Effectively swapping the positions of B and B'. If there is no B' with toRefresh=0 then we writeback B in its original location in R0.

6.4.4 Working Example

For a cache miss, the block is fetched from main memory. This block to be placed in a particular region before being served to ULC. Below we give two illustrations: one on a read-miss and second on a write-miss.

- Read Miss
  This includes GETS and GET INSTR request from ULC. According to figure 6.10, newly fetched blocks from main memory.

  Case-1: Instruction Blocks are placed in R3-region (shown by arrow-1) and a copy is served to the ULC. If no space available in R3-region, one block randomly selected with is-Instr=0 from R3 and migrated to R2-region (shown by dotted arrow-2).

  Case-2: For Instruction blocks, In case R3 contains all the instructions blocks we allocate block to R2-region (shown by arrow -3).

  Case-3: On GETS request, priority would be given in R3-region on allocation (shown by arrow 4). If no space in R3-region we allocate block to R2-region (shown by arrow-5).

- Write miss
  This includes GETX request from ULC, a block to be fetched from main memory. There would be following cases:

  Case-1: As shown in figure 6.11, on GETX, we place block at R0-region (shown with arrow-1). If no space available in R0-region, one block randomly selected with To-Ref = 1 from R0-region (shown with dotted arrow-2) and moved to R1-region.

  Case-2: In case R0 contains all the blocks with To-Ref==0 then we allocate block in R1-region (shown with arrow-3) .

Case-3: On PUTX request from ULC to R0-region (shown by dotted arrow-4), we perform writeback in R1-region (shown by arrow-5) if there is a block in R1 with To-Ref=0. In this case we swap this block with the position of the original block in R0 (shown with dotted arrow-6).

In case for a particular block, no space available in corresponding regions a combined LRU from R0+R1 and from R2+R3 would be evicted from LRT and HRT region respectively.

6.4.5 CAPMIG_SZ: Skip refreshing on Zero Data Blocks

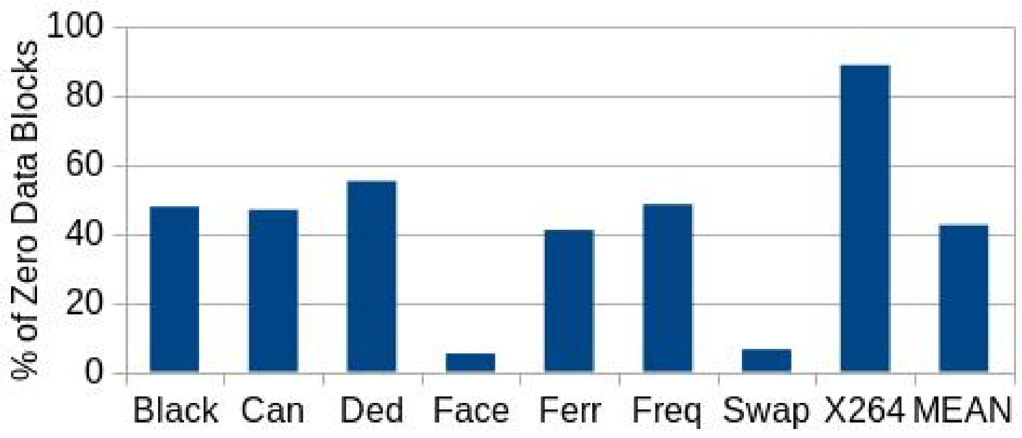


Fig. 6.12 The percentage of zero data block (ZDB) in different benchmarks

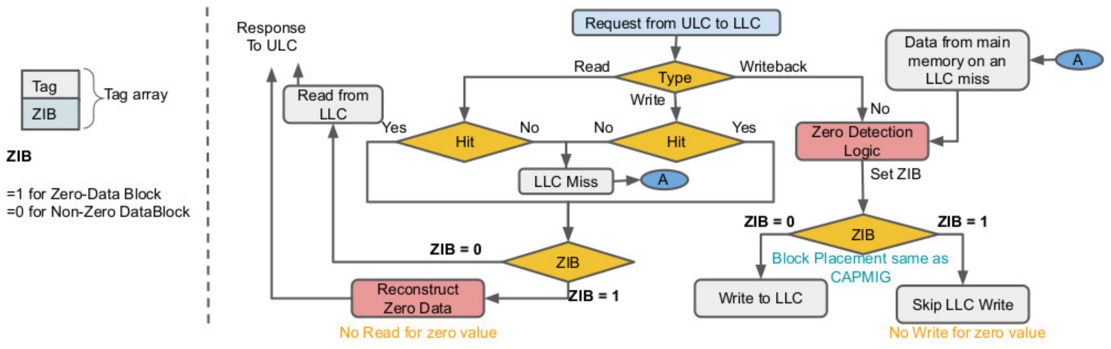


Fig. 6.13 Flow-chart for the CAPMIG_SZ policy. We can see additional modification to the tag array(at bottom).

One of the major observations done over the different benchmarks and literature survey, that we have a considerable amount of data blocks with zero bits as content. Ekman and Stenstorm [17] have found that for their workloads, 30% of blocks have all 64-bytes as zero. Figure 6.12 shows the percentage of zero-data blocks in our benchmarks. Evidently, the frequency of zero-data blocks is different in the benchmarks, and on average, 44.4% blocks are zero data blocks (ZDB). Based on this information, our next proposal is to skip the refresh operations on the ZDBs.

Based on this information, we have performed an augmentation over CAPMIG; Skipping zero over CAPMIG policy. Our next proposal is to skip the restore operations on the Zero-Data Blocks (ZDB). Whenever a block having zero content is read, we do not refresh its contents. This saves both read energy and the need to restore the block. In this way, the zero-detection approach saves both access time and dynamic energy. If the data content is not zero, then a read operation happens normally, and the block is refreshed periodically. We further propose another optimization, where a writeback to a block with zero data content will not be written to the LLC. This optimization would further reduce the number of writes performed in the LLC.

Architecture: Figure 6.13 shows the changes to the tag array. Here, the LLC tag array is associated with one zero indicator bit (ZIB) to indicate whether the data block has zero content or not. If this bit is set, then the block has zero value content. We use a comparator unit called the "zero detection logic" (ZDL) to detect the data value. Whenever a block is loaded in the LLC from the main memory or written back from the ULC, it is passed through the ZDL. If the block has a zero value, then the block is not written in the LLC, but the ZIB is set. When a block is read from the LLC, which has the ZIB set, we do not perform an actual read operation on the data block from the LLC. Instead, we use a zero block reconstruction logic (in particular, a register storing zero value) to send a block with zero content. Note that the ZIB bit information is maintained with the tag array.

Block Placement: As we discussed in section 6.3 , We are exploiting the features of coherency protocol and placing blocks depending on the type of request generated by ULC. At the time of placement, our proposed ZDL architecture ensures setting ZIB bit. We make the following updates:

- For blocks with ZIB=1 indicates blocks are Zero data blocks. Thus, we save write energy while blocks placement. We do not write actual blocks; update only tag bits. We skip refresh operations for such blocks.

- For blocks with ZIB=0 indicates it's a Non-Zero data block. Block placement as per the conditions mentioned in section 6.3.

Table 6.1 System Configuration

System Component	**Configurations**
Processor	X86 Quad-core @ 2Ghz
L1 SRAM Cache	32KB Private , 4 way, split I/D caches, 64B block, LRU, MESI , write back policy
L2 STT-RAM Cache	4MB(4 bank each bank of 1MB), 64B cache block, 16-way, LRU, MESI Protocol. Retention Times: 100μs,1ms, 10ms, 100ms (with ways, 0-3,4-7,8-11,12-15 of cache set respectively)
Multi-Threaded Benchmarks	Blackscholes (Black), Canneal (Can), Dedup (Ded), Facesim (Face) , Ferret (Ferr), Fluidanimate (Fluid), Freqmine (Freq), Swaptions (Swap), X264

Working: Figure 6.13 shows the flow-chart for the CAPMIG_SZ policy. For every request coming to the LLC from the ULC, apart from CAPMIG we also maintain the following criteria for CAPMIG_SZ on:

(1) Read hit: Check the ZIB bit of the block. If ZIB=1, read access to a zero data block; hence, the read is served by reconstruction of the data block with zero values. This avoids unnecessary read operation on the data block. If ZIB=0, the read operation takes place, and after this, periodic refresh operations is performed according to the corresponding region.

(2) Write hit: The write-hit also amounts to a block read from the LLC perspective. Therefore, the same actions as those taken on a read-hit take place in this case.

(3) Writeback and cache miss: A write needs to be performed to the LLC on a writeback from ULC to LLC and on a cache miss when the data comes from the main memory to the LLC. In both these cases, the ZDL checks the data contents and sets the ZIB. If the contents are zero (ZIB=1), write to LLC is skipped. However, if the contents are non-zero, a write happens normally.

Migrations: In order to place the block in the correct region or to make room for another better candidate. We did migrations in our proposed methodology. We do block migration for various types of requests and whether they are cache hit or miss. The procedure to be followed would be the same as CAPMIG. We get a benefit over write energy. For blocks with ZIB=1, we do not require write energy; this will save read count and a significant amount of dynamic energy during migration.

6.5 Experimental Evaluation

6.5.1 Setup

We evaluate our proposal on a full system simulator GEM5 [87]. The memory module is simulated using Ruby module inside GEM5. The MESI CMP based cache coherence protocol is used and modified to implement the proposed refresh optimization proposal. This works based on data-width only and hence, can be integrated with any cache coherence scheme. Table 6.1 shows the architecture parameters used in our setup.We use L1 as private caches and Multi-Retention STT-RAM based L2 as the last level cache.

The LLC performs a block level refresh. We have 4 different retention intervals varying from $100\mu s$ to $100ms$. The parameters of STT-RAM cache are obtained using NVSim [129]. We model the different relaxed retention times and energy parameters depicted in table 6.2 [78]. As for workloads, we use PARSEC [115] benchmarks with medium input sizes. We have done simulation on medium variant of the 4-threaded PARSEC benchmark applications to verify our proposed policy. These benchmarks include large variety of multi-threaded applications such as mining, multi-media, animation, computer vision, etc. as shown in Table 6.1. All the design overheads and effects of protocol changes are taken into consideration during simulations. We compare the proposed policy with the Pure STT-RAM based cache with no refresh issues (i.e. a non-volatile STT-RAM) We show the results for the following techniques:

- **SRAM**: This is a baseline SRAM cache which needs no refreshing and has the best access latency. However, such caches suffer from high leakage energy.
- **NV-STT**: This scheme is an ideal, Refresh-Free STT-RAM cache, which needs no refresh as it retains data for considerable interval. However, it takes longer latency and energy to write the data; and at the same time it is slower in reads compared to the volatile STT-RAM design.
- **DRS**[76]: Dynamic Refreshing Scheme (DRS) [76] is an ideal multi-retention cache which needs periodic refresh scenario. This setup places the block randomly using LRU replacement policy, irrespective of the retention time of the location.
- **CACHE REVIVE [121]** This policy refreshes only Most Recently Used(MRU) blocks by copying the blocks with diminishing retention time in the refresh buffer and then moving back to main cache.

Table 6.2 Timing and Energy parameters for Pure STTRAM and Retention-relaxed STT-RAM cache configurations at different retention time

Cache Config.	1MB, 64B line size 16way					
Mem. Device	SRAM	NV-STT	retention-relaxed STTRAM [78]			
Retention Time	-	-	100 μs	1 ms	10 ms	100 ms
Write Ener. (per access)	0.338 nJ	7.68 pJ	0.392 nJ	0.404 nJ	0.419 nJ	0.438 nJ
Hit Ener. (per access)	5.318 nJ	5.794 nJ	5.794 nJ	5.794 nJ	5.794 nJ	5.794 nJ
Leakage Pow. (in mW)	3234.91	391 mW	2200.032 mW			
Read Lat. (in cycles)	2	7	2	2	2	2
Write Lat. (in cycles)	2	21	3	4	6	7
Full Form	Mem.=Memory, Lat. = Latency, Pow.= Power, Ener.= Energy					

- **CAPMIG**: This is our proposed policy with customized placement and migration in multi-retention cache.
- **CAPMIG_SZ**: This includes a combination of CAPMIG and skipping the Zero-Data Block policy.

6.5.2 Results and Analysis

Refresh Count: Our proposal identifies private blocks loaded in ULC and avoid refreshing them in the LLC. At the same time the block placement helps to avoid refreshes of blocks that get written-back. We get a significant reduction in R0-region as we are trying to place blocks which are held exclusively by the ULC. Figure 6.14 shows the reduction in refreshes achieved by our proposed methodology against the baseline, DRS, which refreshes every cache line due to random placement of blocks. Whereas, we place blocks to best possible location according to read and write locality. We get a reduction in the range of 4% - 74% with an average reduction of 34.5% in CAPMIG. The improvisation of skip zero CAPMIG_SZ, gives 41% reduction in refresh count w.r.t DRS. The existing Cache Revive policy is not shown towards refresh count and refresh energy because this policy is mainly for writebacks and occasionally refreshes the blocks (just before expiry). The writebacks generated by Cache Revive will affect the performance and miss rate. This effect is shown in the relevant subsections.

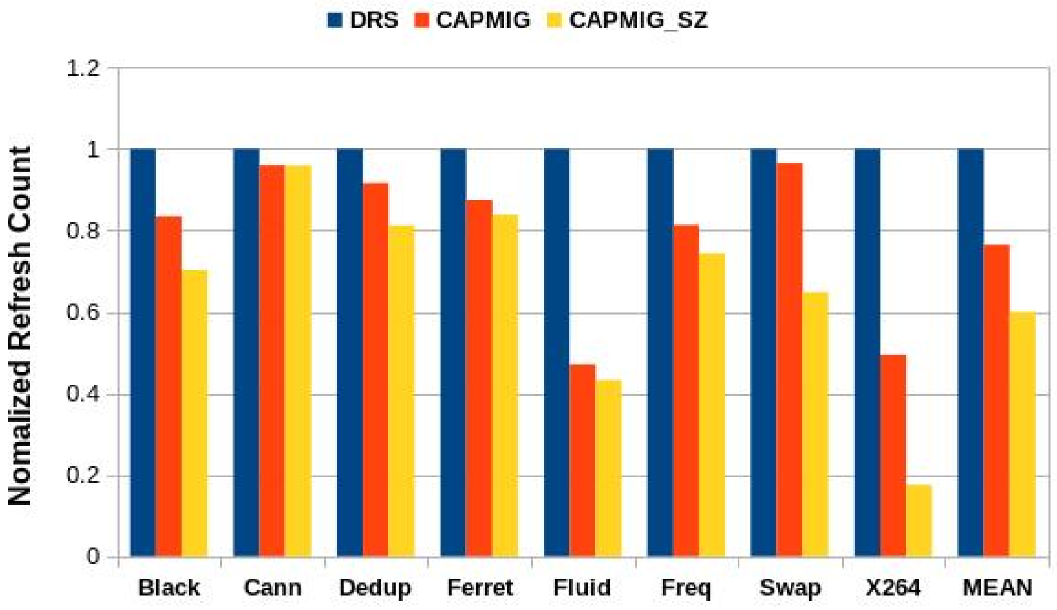


Fig. 6.14 Refresh count of CAPMIG policy normalized w.r.t. random placement policy.(lesser is better)

Regions comparison A detailed breakup of region-wise reduction in refresh counts is shown in figure 6.16. As can be seen, we get considerable reductions in every region. We also get maximum reduction in R0-region as our policy makes sure that write intensive blocks are loaded in R0 and we also skip the refreshes for GETX block which are held exclusively with the ULC. The number of refreshes incurred by the HRT are lesser compared to the LRT and hence the savings are also comparatively lesser. The block placement policy makes sure that: *the HRT incur only the minimum required number of refreshes and the major savings comes from the LRT.*

Performance: Figure 6.15 shows the normalized CPI for all techniques wrt SRAM cache. We use SRAM as baseline here to show the degradation in performance on account of higher access latency. Due to longer access latency non-volatile STT-RAM degrades CPI by 7% over SRAM. The access latency in multi-retention based cache is lower in comparison to pure STT-RAM and therefore the DRS scheme using multi-retention STT-RAM degrades CPI by 4.7%. Our proposed policy CAPMIG improves on the CPI over DRS and achieves CPI within 4.7% of SRAM. Overall we go closer to CPI of SRAM and also save energy compared to DRS. This will give us better EDP. If

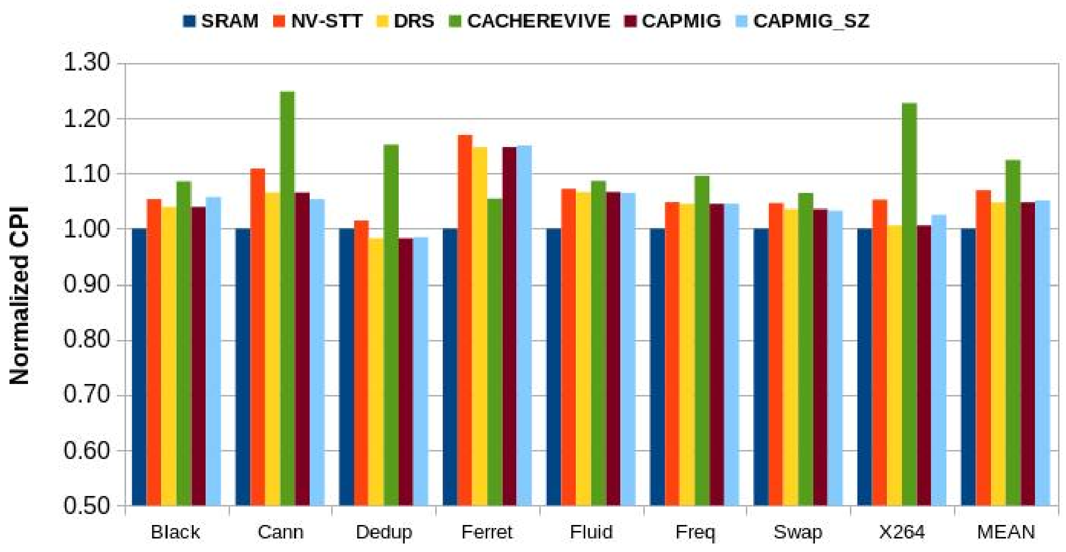


Fig. 6.15 Performance of various policies normalised with baseline SRAM (lesser is better).

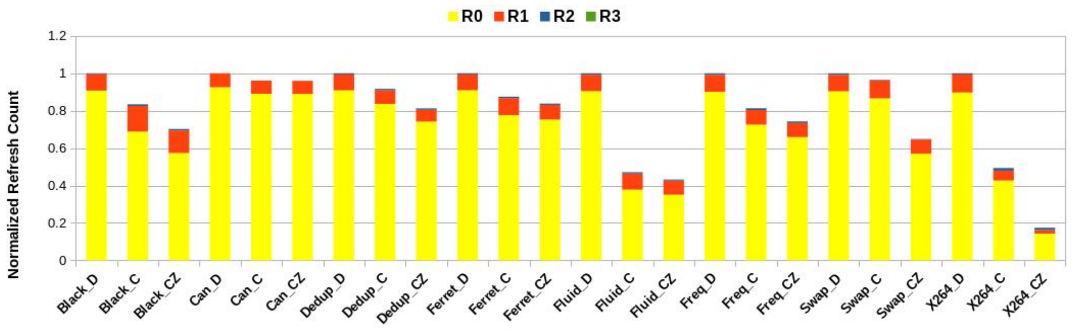


Fig. 6.16 Region wise normalised Refresh Counts for proposed policy CAPMIG (_C) and CAPMIG_SZ (_CZ)w.r.t DRS with random allocation. (_D).

we compare the existing policy CACHE-REVIVE, it results in 13% degradation over SRAM whereas our final policy improves on CPI over existing policy.

Block placement in CAPMIG affects their lifetime in the cache which in turn affect the miss-rate. This is because the selection of victim is based on the region and not the global LRU. The existing policy CacheRevive writebacks the blocks on expiry if they cannot be refreshed. This too results in increase in miss-rate. Figure 6.18 shows in normalised miss rate wrt SRAM. As expected CacheRevive increases miss rate considerably. Whereas, our proposal is able to control the same within 5% degradation.

Energy Consumption:

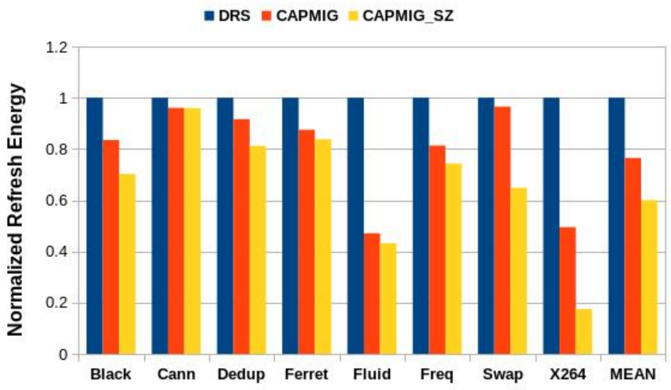


Fig. 6.17 Normalized Refresh Energy for proposed policy w.r.t DRS(lesser is better)

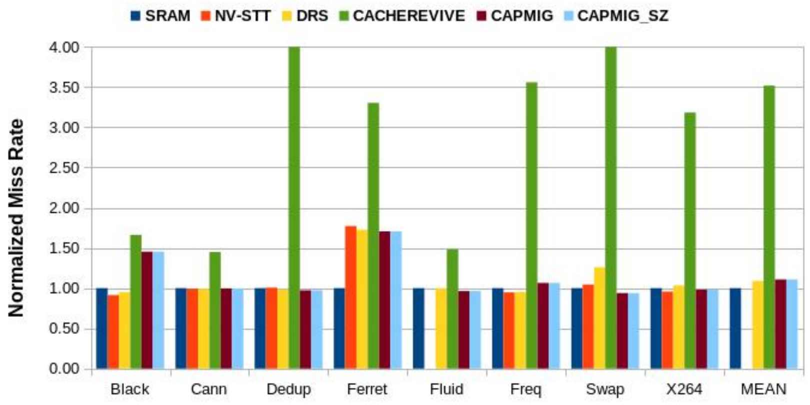


Fig. 6.18 Miss Rate of proposed policy normalized w.r.t SRAM

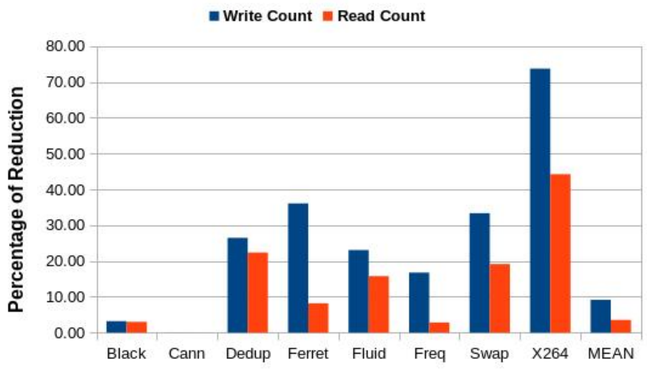


Fig. 6.19 Percentage Reduciton in number of Reads and Writes in CAPMIG_SZ wrt CAPMIG

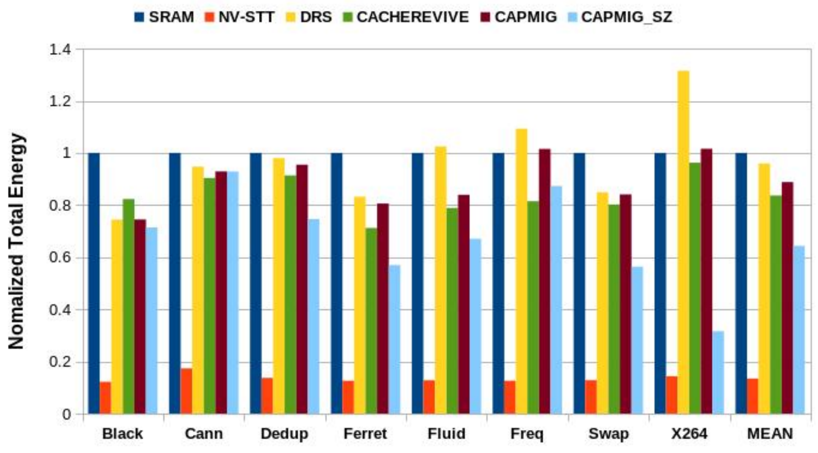


Fig. 6.20 Total Energy normalized to baseline

Figure 6.17 shows the normalised refresh energy values for DRS and CAPMIG. As can be see our saving of refreshes also saves energy. In particular we get 41% savings over DRS.

We use NVM to take advantage of low leakage energy. The total energy (static + dynamic) is shown in Figure 6.20. Baseline SRAM consumes the maximum. The proposed policies that optimise on refreshes save energy. In particular, CAPMIG outperforms SRAM by 12% on overall total energy and whereas NV-STT and DRS save 86% and 5% energy respectively wrt SRAM.

The frequent reallocation of our algorithm causes an energy overhead compared to DRS. However, our combined eviction policy, redirecting writeback to the R1 region and block placement in correct region help in overall energy savings.

The improvisation of skipping zero policy over the CAPMIG gives savings of 31% over the baseline. It gives higher savings due to optimization of not writing zero-blocks to the LLC, which is not adopted by the CAPMIG. We note that the percentage of ZDBs is highest for x264. Correspondingly, CAPMIG_SZ policy provides the highest savings for X264 workload.

The CAPMIG_SZ policy leverages advantage by existence of Zero Data blocks in the application. As shown in fig. 6.12, we get an average of 42.5% zero data traffic at the last level. As shown in Table 6.3, this traffic is mainly divided across blocks loaded from main memory (column-3) and blocks written back from ULC (column-4). This percentage distribution of blocks leads to savings in write energy for our optimization; also, it directly impact the refresh energy savings by skipping unnecessary refresh of Zero blocks. Figure 6.19 shows the reduction of Read and Write count in percentage due to skipping zero in CAPMIG_SZ.

Table 6.3 Division of written zero data blocks in last level cache.

Bench.	ZDB Written in LLC(%)	ZDB loaded on Miss in LLC(%)	ZDB written on writeback in LLC(%)	ZDB Hit in LLC(%)
Black	35.05	15.23	19.85	10.8
Can	46.93	32.10	14.83	0.003
Ded	55.22	41.40	13.82	22.172
Fluid	33.4	10.02	23.38	50.47
Body	41.05	25.50	15.55	0.348
Freq	48.51	41.00	7.51	2.627
Swap	6.54	6.17	0.37	18.225
X264	88.86	75.03	13.83	46.633

6.5.3 Comparison with Related Works

Prior work has shown significant reduction in the processor's power [130]; thus, replacing SRAM with STT-RAM can considerably improve the energy efficiency. A significant reduction in leakage power, reduced the total energy consumption in DRS as compared to the SRAM.

In DRS retention time kept constant throughout the execution of the system. Technically it performs refresh which require a temporary buffer to hold data during refresh operations. We have also implemented another existing approach Cache Revive [121], where author used a temporary buffer of 1900 slots. We have scaled down the buffers size as per our corresponding configuration. The Cache Revive policy is writeback based, in that beyond a certain number of blocks that get refreshed, most of the expiring blocks get written back to the next level. This results in increased miss-rate and affects the performance. We have shown the effect on both these metrics in the results section.

HALLs [78] proposes an iteration based approach in order to select the best possible configuration of the cache and retention time interval, whereas our approach is focused on block placement in order to reduce refresh energy. The HALLS, proposes refresh free approach by per-block counter lifetime tracking. If counter lapse, the cache block will be evicted and if they are dirty first written to the main memory before eviction. We are not performing comparison with HALLs as approach focuses specifically on configuration and Retention time selection. Besides that it is a pure writeback based strategy rather than refreshing a block. As our proposal is refresh based and HALLS is writeback based, we have not quantitatively compared with HALLS.

Mirror cache [122], is an another energy-efficient block refreshing approach. This policy relaxes the retention time in order to reduce write latency. This scheme is made for the L1 i.e Upper Level Cache,where cache blocks mirrored in a different auxiliary segment of cache. Generally, ULCs are low capacity caches and mirroring could be possible since overhead will be less. This approach is not scalable for the Last Level Cache(LLC), therefore we have not compared CAPMIG with Mirror Cache policy.

DASCA [131] proposed policy predicts dead block writes that are written on the cache. Dead blocks refers to those that are loaded in cache and but not reused thereafter. A similar idea was proposed by Wang [132] for both NV-STT and NV-RRAM based Hybrid Cache. They predict access pattern for direct block placement and migration. [130] Previous work has explained that how inherent retention time is relaxed in order to make Retention-Relaxed STTRAM [120]. LARS [118] is an approach of adaptable runtime cache depends on the different applications. It proposes an optimal tuning algorithm that determines the retention based on energy consumption at runtime. Sun et al. [76], used retention relaxed STTRAM to make a hybrid version of the cache with two retention levels. We have a very limited existing policy as an option for a fair comparison with our proposed policy. As per our knowledge, no previous work targets block placement strategy as key concept to minimize the refresh operations in multi-retention LLC.

6.5.4 Overhead Analysis

Latency Overhead: As explained in section 6.3, in order to make space in R3 we migrate B' to R2 region. This migration helps in prioritizing Instruction in R3 region. However, it incurs a read from R3 region (2 cycles) followed by write in the R2 region (6 cycles). Similarly on write miss, in order to make space in R0 we migrate B' to R1 region. This migration helps in keeping freshly fetched GETX blocks in R0 region. It incurs a read from R0 region (2 cycles) followed by write (4 cycles) in the R1 region. During this migration of blocks, the LLC is stalled, which accounts for the latency overheads. Besides above migration we have another infrequent relocation on PUTX. We try to perform writeback in R1 region for blocks previously in R0 region. It incurs a tag migration from R0 to R1 region and write redirection. In case no space in R1, one B' promoted to R0 region incurring read from R1 region (2 cycles) and write in R0 region (3 cycles). These overheads are taken into account in our experimental evaluation. However, it be noted that in spite of these overheads, we get overall better performance. In particular CAPMIG_SZ improvises the results after reduction of

Energy Overhead: Our technique incurs energy overhead for migrations between regions. A migration requires read and write of the corresponding source and destination regions (mentioned in Table-II). A redirection on PUTX from R0 to R1 requires tag read and tag write only as writeback will be redirected to R1 after tag migration.

Storage Overhead: We use SRAM-based saturating expiration counters in each cache block for the proposed multi-retention system. Each expiration counter has four states and each cache block takes two bits. We employ two data buffers of 64 bytes each and two tag buffers of 42 bits each for block reallocation.Our technique incurs the following overhead: (i) one To-Refresh bit, one Instruction bit, and one zero indicator bit(ZIB) (ii) a (42bit + 64B)-size swap buffer used to migrate the tag and data block between the regions. In total, there is 16.06 KB additional storage, which is only 0.39% overhead for a 4MB cache. Note that the area overhead for the additional tags of CAPMIG_SZ is modeled using NVSIM [129].

6.5.5 Overhead comparison with existing policies

CACHE-REVIVE includes three types of overhead:1) Static overhead due to buffer slots, 2)Migration overhead from STTRAM to buffer and reviving from buffer to STTRAM, 3) Increase in Writeback percentage.

Cache-revive performs operations over blocks with elapse time 3/4th retention time interval. If a block belongs to the first half of the MRU slots, a particular block will be migrated to the temporary buffer if space is available, otherwise evicted. If the block is dirty, it will be written back to the main memory. If the block belongs to the other half, in that case, the block is either evicted or written back to the main memory. CAPMIG does not perform forced eviction from the Cache; However, it suffers from few migrations among the regions based on the *to-ref* and *is-instr* bit.

CACHE-REVIVE policy contains overhead due to early evictions and writeback. We can see in figure 6.18, Cache-revive increases the miss rate with respect to the Pure SRAM cache. It also includes area overhead due to the temporary buffer slots. There is no extra storage involved in CAPMIG.

6.6 Summary

In order to fulfill the demand for diverse applications, the multi-retention caches have immense potential to replace other pure non-volatile memories. It reduces the access cost over the cache with high retention as a longer retention is of no use for the last level cache, in that the blocks get updated or evicted before the worst case retention. High density, reduced access latency and choosing location according to the re-usability of the block are enough fascinating features to go for the multi-retention caches. However, reduction in access cost has a trade-off with refresh penalty. To make them good replacement candidates, we must optimize on the refreshing aspect. Towards achieving this goal, this chapter presented coherence based method to decide the best region to place the blocks. As the blocks loaded on a write are likely to receive more writes we place them in the Low Retention Region (LRT). Similarly, blocks loaded on reads and instruction blocks are placed in the High Retention Region (HRT). We also proposed certain conditions under which the blocks are moved across regions. All these contribute to savings in number of refreshes by 41% leading to 36% improvement in total energy compared to baseline. The policies also result in performance improvement over the baseline refresh policy and stays within 5% of the SRAM.

The proposed policy uses the request type to decide the placement and does not require additional overheads related to identification of write intensity of blocks and bulk migrations across regions. A simple, one-time decision and occasional block migrations give us significant savings. Thus use of coherence information can help in making multi-retention caches usable with minimised block migration and refresh overheads.

CHAPTER 7

Conclusion

This research work was motivated by the recent trend of employing emerging memory technologies in the cache hierarchy. SRAM based caches are prone to high leakage energy leading to use of alternatives like embedded-DRAM and non-volatile STTRAM based caches. These alternatives come with higher density and low leakage. However they have certain drawbacks like either requiring frequent data refreshes or consuming high latency and power during writes. The STTRAM write latency-energy drawback can be overcome by designing them in such a way that the cells can be written quickly and/or their dimensions can be reduced. This design implies that the retention time of data reduces requiring it to be refreshed. Another obstacle is that the read and write current become almost similar causing a read to potentially destroy the written value. In order to maintain data correctness, we need to restore the values every time the value is fetched.

- We have aimed to address the reliability of data over the volatile caches by ensuring the retention of data blocks. Towards this we have designed various strategies with minimal overhead.
- **The thesis aimed** at reducing the number of times we need to refresh or restore the data values in eDRAM and volatile-STTRAM based caches. Towards this we designed policies by looking at the content of the data and determining whether it needs to e refreshed and we also exploited coherence protocol based messages to intelligently decide when to refresh and where to place the block so that it need not be refreshed often.

- **Our content and coherence based strategies** have significantly reduced the refresh energy without hampering the performance. All policies compared fairly well with existing state-of-the-art works.
- New contributions to knowledge were made in respect of volatile large last level caches. Whereas the objective of SRAM cache has traditionally been to reduce long disparity between processor and main memory; this work demonstrates that this performace gap can reduced by eDRAM and volatile-STTRAM based caches.
- Several new observations support the proposed use of the strategies that was developed and investigated in this work; Zero data detection method was used to optimize and enhance result in order to reduce overhead; SRAM Read buffer in our work CORIDOR played a cruicial role in order to deal Read Disturbance issue.

7.1 Summary of Contributions

- **Refresh Optimisation through Dynamic Reconfiguration Based Policy in eDRAM:** embedded-DRAM caches need to refresh the data at regular intervals. This work concentrated on looking at the content of the block to decide whether to refresh it or not. In particular, the blocks having zero data values need not be refreshed. We identify such blocks by using a zero detection logic and keep a bit with each block identifying whether it is a zero block. Whenever the refresh is triggered for eDRAM, all the blocks with identified zero content will not e refreshed. Note that such blocks have invalid data and when a read request arrives for this data we cannot return the stored content. In that we need to reconstruct the zero block before serving the read. For this we use an additional (zero value) register whose content is forwarded on reads to zero data blocks. In order to ease the refresh selection, we relocate blocks with zero data to a separate logical partition within the cache. Further we extended our proposal to dynamically change the size of the zero value partition depending on the working set and content of the running application. In this proposal we achieved refresh reduction of around 38% with performance gain of 6-9% and 12-14% energy savings.
- **Coherence based refresh optimisation in eDRAM caches:** Apart from seeing the content, if we check the access request we can identify certain blocks which are getting updated in upper level caches and need not be refreshed at the LLC

level. For this we make use of the coherence protocol messages. Blocks that are requested for writing by the ULC will be identified to be not refreshed in the LLC. Only after the block getting written back from the ULC we will flag it for refreshing. This policy reduces refresh by 55% with performance gain of 4% and energy savings of 62%.

- **CORIDOR: Coherence based technique to reduce restores cased by read-disturbance error in STTRAM caches:** In the deep sub-micron region, STT-RAM suffers from read-disturbance error (RDE), whereby a read operation disturbs the stored data. Mitigation of RDE requires restore operations, which imposes latency and energy penalties. Hence, RDE presents a crucial threat to the scaling of STT-RAM. Towards this we offered three techniques to reduce the restore overhead. First, we make use of coherence protocol to identify blocks that will get updated at a higher level cache in the near future. For such blocks we avoid the restore operations upon reads. Second, we identify read-intensive blocks using a lightweight mechanism and then migrate these blocks to a small SRAM buffer. On a future read to these blocks, the restore operation is avoided as it gets serviced from the SRAM buffer. The challenge here was to identify the correct threshold to declare a block read intensive and also manage replacement policies for the limited size SRAM buffer. Third proposal was based on content of the data block. In that for data blocks having zero value, a write operation is avoided, and only a flag is set. Based on this flag, both read and restore operations to this block are avoided. Further, we combined these three techniques to design our final policy, named CORIDOR. Compared to a baseline policy, which performs restore operation after each read, CORIDOR achieved savings of 51% energy in restore operations while reducing execution time to 0.64 times. The total energy savings were 31%.

- **CAPMIG: Coherence based refresh optimisation in multi-retention STTRAM caches:** Multi-Retention STTRAM caches have been considered as a good alternative over standard STTRAM caches by reducing their retention time which reduces the write latency. As the retention time is reduced we need to refresh the data when the retention time expires. This leads to considerable amount of latency and energy. To mitigate this if the block gets updated by the ULC then the refresh is automatically done. If we can manage to place blocks in the appropriate retention zone depending on their access patterns then we can avoid forceful refresh at the end of retention interval, as the cache access will guarantee block refresh. Towards achieving this, we proposed coherence based policy to decide the retention partition in which a block gets placed. A block loaded on a write access is likely to get more writes in future and is therefore loaded in the lowest retention region. Similarly, instruction blocks are loaded in highest retention time region. This helps in reducing the number of refreshes incurred by the blocks. During runtime the blocks may change their access patterns, requiring a change in their retention region. We also proposes a migration policy to relocate the blocks to appropriate regions during runtime. Identification of zero data value blocks and not refreshing them is an additional augmentation to our proposal. The contributions save refreshes by 41% leading to 36% improvement in total energy compared to baseline. The policies also result in performance improvement over the baseline refresh policy and stays within 5% of the SRAM.

7.2 Scope for Future Work

- eDRAM caches are proliferating in embedded devices as well as on accelerators. It is seem that machine learning accelerators prefer to use eDRAM on account of their density and low leakage. We can extend our refresh optimisation methods to this domain where the data demands are very high and the data have low temporal locality. The access patterns in this domain may not be similar to our experimental setup and hence we see scope of extending our proposals.
- Similar to RDE, STTRAM also suffers write disturbance errors. We can extend our proposals to mitigate these type of errors.
- eDRAM caches can be combined with SRAM partitions to form a hbyrid cache. Policies can be proposed to move refresh prone data to SRAM partition so that we can reduce number of refreshes in eDRAM.

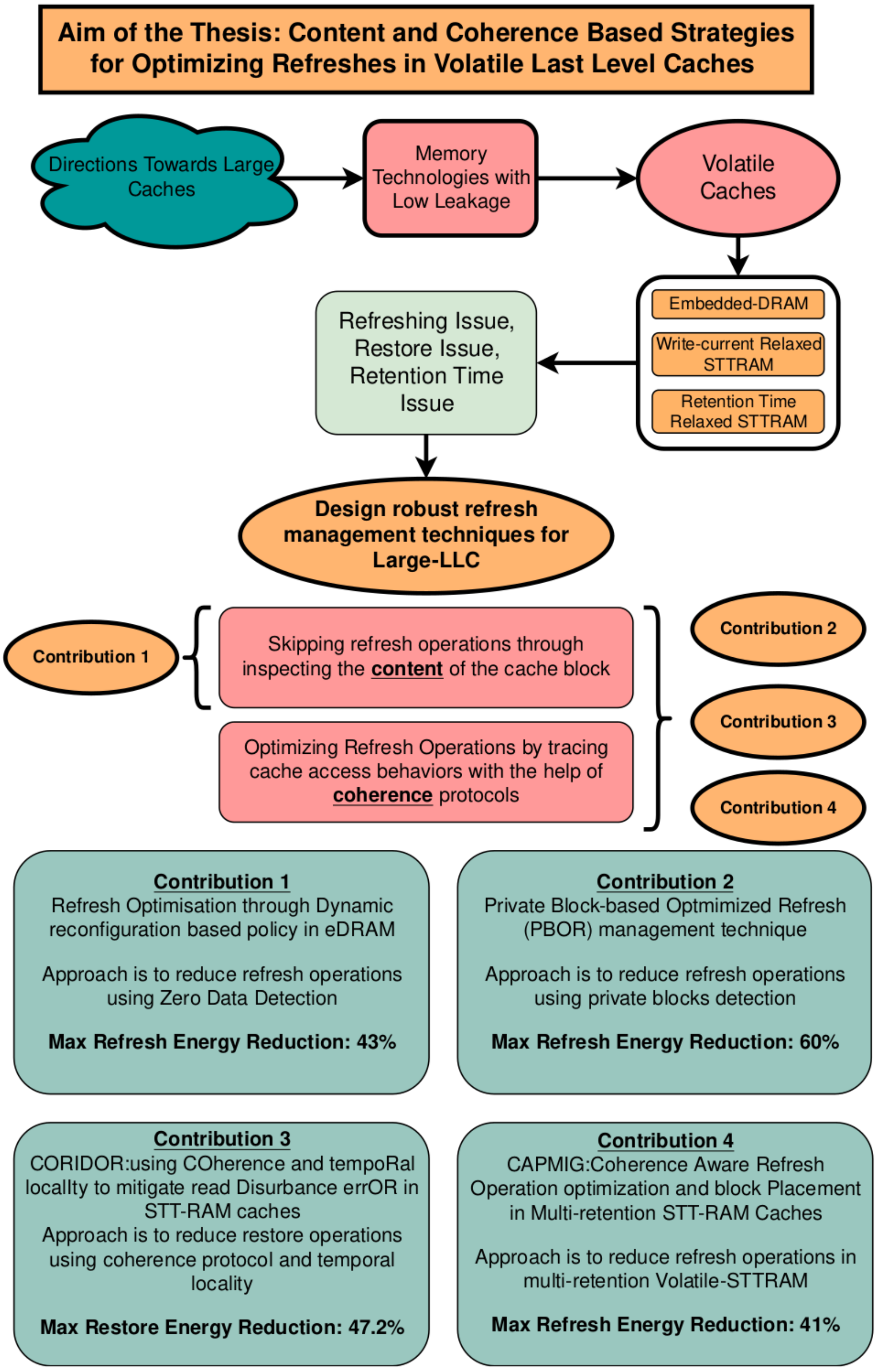

Aim of the Thesis: Content and Coherence Based Strategies for Optimizing Refreshes in Volatile Last Level Caches
Directions Towards Large Caches
Memory Technologies with Low Leakage
Volatile Caches
Embedded-DRAM
Write-current Relaxed STTRAM
Retention Time Relaxed STTRAM
Refreshing Issue, Restore Issue, Retention Time Issue
Design robust refresh management techniques for Large-LLC
Contribution 1
Skipping refresh operations through inspecting the content of the cache block
Optimizing Refresh Operations by tracing cache access behaviors with the help of coherence protocols
Contribution 2
Contribution 3
Contribution 4
Contribution 1
Refresh Optimisation through Dynamic reconfiguration based policy in eDRAM
Approach is to reduce refresh operations using Zero Data Detection
Max Refresh Energy Reduction: 43%
Contribution 2
Private Block-based Optmimized Refresh (PBOR) management technique
Approach is to reduce refresh operations using private blocks detection
Max Refresh Energy Reduction: 60%
Contribution 3
CORIDOR:using COherence and tempoRal localIty to mitigate read Disurbance errOR in STT-RAM caches
Approach is to reduce restore operations using coherence protocol and temporal locality
Max Restore Energy Reduction: 47.2%
Contribution 4
CAPMIG:Coherence Aware Refresh Operation optimization and block Placement in Multi-retention STT-RAM Caches
Approach is to reduce refresh operations in multi-retention Volatile-STTRAM
Max Refresh Energy Reduction: 41%

Fig. 7.1 Summarizes the contributions of this thesis.

References

[1] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. 1

[2] John Barth, Don Plass, Erik Nelson, Charlie Hwang, Gregory Fredeman, Michael Sperling, Abraham Mathews, Toshiaki Kirihata, William R. Reohr, Kavita Nair, and Nianzheng Cao. A 45 nm soi embedded dram macro for the power™ processor 32 mbyte on-chip l3 cache. *IEEE Journal of Solid-State Circuits*, 46(1):64–75, 2011. 2, 8

[3] Sally A. McKee and Robert W. Wisniewski. *Memory Wall*, pages 1110–1116. Springer US, Boston, MA, 2011. 4

[4] Wei Zang and Ann Gordon-Ross. A survey on cache tuning from a power/energy perspective. *ACM Comput. Surv.*, 45(3), jul 2013. 7

[5] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *Digital Tech. J.*, 9(1):49–62, jan 1997. 7

[6] Blaine Stackhouse, Sal Bhimji, Chris Bostak, Dave Bradley, Brian Cherkauer, Jayen Desai, Erin Francom, Mike Gowan, Paul Gronowski, Dan Krueger, Charles Morganti, and Steve Troyer. A 65 nm 2-billion transistor quad-core itanium processor. *IEEE Journal of Solid-State Circuits*, 44(1):18–31, 2009. 7

[7] Sparsh Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 4(1):33–43, 2014.

[8] Subhash Saini, Andrey Naraikin, Rupak Biswas, David Barkai, and Timothy Sandstrom. Early performance evaluation of a "nehalem" cluster using scientific

and engineering applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery. 7

[9] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. Design and optimization of large size and low overhead off-chip caches. *IEEE Transactions on Computers*, 53(7):843–855, 2004. 7

[10] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, may 2011. 7

[11] Julien Ryckaert, Pieter Weckx, and Shairfe Muhammad Salahuddin. 3 - sram technology status and perspectives. In Andrea Redaelli and Fabio Pellizzer, editors, *Semiconductor Memories and Systems*, Woodhead Publishing Series in Electronic and Optical Materials, pages 55–86. Woodhead Publishing, 2022. 7

[12] S. Mittal et al. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015. 7, 8

[13] W.R. Reohr. Memories: Exploiting them and developing them. pages 303 – 310, 10 2006. 7

[14] P. G. Emma, W. R. Reohr, and M. Meterelliyoz. Rethinking refresh: Increasing availability and reducing power in dram for cache applications. *IEEE Micro*, 28(6):47–56, Nov 2008. 8

[15] I. Bhati et al. Dram refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 65(1):108–121, Jan 2016. 8

[16] S. Mittal et al. A survey of techniques for architecting dram caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863, June 2016. 8

[17] Valentina Salapura, Jose R. Brunheroto, Fernando Redigolo, and Alan Gara. Exploiting edram bandwidth with data prefetching: simulation and measurements. In *2007 25th International Conference on Computer Design*, pages 504–511, 2007. 8

[18] W. J. Starke, J. Stuecheli, D. M. Daly, J. S. Dodson, F. Auernhammer, P. M. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner. The cache and memory subsystems of the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):3:1–3:13, 2015. 8

[19] Andrea Calimera, Alberto Macii, Enrico Macii, and Massimo Poncino. Design techniques and architectures for low-leakage srams. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(9):1992–2007, 2012. 20

[20] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. *SIGARCH Comput. Archit. News*, 41(3):60–71, jun 2013. 22, 23

[21] S.Y. Hou, H. Hsia, C.H. Tsai, K.C. Ting, T.H. Yu, Y.W. Lee, F.C. Chen, W.C. Chiou, C.T. Wang, C.H. Wu, and Douglas Yu. Integrated deep trench capacitor in si interposer for cowos heterogeneous integration. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 19.5.1–19.5.4, 2019. 23

[22] K.C. Huang, Y.W. Ting, C.Y. Chang, K.C. Tu, K.C. Tzeng, H.C. Chu, C.Y. Pai, A. Katoch, W.H. Kuo, K.W. Chen, T.H. Hsieh, C.Y. Tsai, W.C. Chiang, H.F. Lee, A. Achyuthan, C.Y. Chen, H.W. Chin, M.J Wang, C.J. Wang, C.S. Tsai, C.M. Oconnell, S. Natarajan, S.G. Wuu, I.F. Wang, H.Y. Hwang, and L.C. Tran. A high-performance, high-density 28nm edram technology with high-k/metal-gate. In *2011 International Electron Devices Meeting*, pages 24.7.1–24.7.4, 2011. 23

[23] John Barth, William R. Reohr, Paul Parries, Gregory Fredeman, John Golz, Stanley E. Schuster, Richard E. Matick, Hillery Hunter, Charles C. Tanner, Joseph Harig, Hoki Kim, Babar A. Khan, John Griesemer, Robert P. Havreluk, Kenji Yanagisawa, Toshiaki Kirihata, and Subramanian S. Iyer. A 500 mhz random cycle, 1.5 ns latency, soi embedded dram macro featuring a three-transistor micro sense amplifier. *IEEE Journal of Solid-State Circuits*, 43(1):86–95, 2008. 23, 27

[24] Paul Keltcher, Stephen Richardson, and Stuart Siu. An equal area comparison of embedded dram and sram memory architectures for a chip multiprocessor. 2000. 23

[25] Esteve Amat, Antonio Calomarde, Francesc Moll, Ramon Canal, and Antonio Rubio. Feasibility of embedded dram cells on finfet technology. *IEEE Transactions on Computers*, 65(4):1068–1074, 2016. 23

[26] Anh Tuan Do, He Yi, Kiat Seng Yeo, and Tony T. Kim. Retention time characterization and optimization of logic-compatible embedded dram cells. In *2012 4th Asia Symposium on Quality Electronic Design (ASQED)*, pages 29–34, 2012. 23

[27] Ki Chul Chun, Pulkit Jain, Jung Hwa Lee, and Chris H. Kim. A 3t gain cell embedded dram utilizing preferential boosting for high density and low power on-die caches. *IEEE Journal of Solid-State Circuits*, 46(6):1495–1505, 2011. 24

[28] N. Butt, K. Mcstay, A. Cestero, H. Ho, W. Kong, S. Fang, R. Krishnan, B. Khan, A. Tessier, W. Davies, S. Lee, Y. Zhang, J. Johnson, S. Rombawa, R. Takalkar, A. Blauberg, K. V. Hawkins, J. Liu, S. Rosenblatt, P. Goyal, S. Gupta, J. Ervin, Z. Li, S. Galis, J. Barth, M. Yin, T. Weaver, J. H. Li, S. Narasimha, P. Parries, W. K. Henson, N. Robson, T. Kirihata, M. Chudzik, E. Maciejewski, P. Agnello, S. Stiffler, and S.S. Iyer. A 0.039um2 high performance edram cell based on 32nm

high-k/metal soi technology. In *2010 International Electron Devices Meeting*, pages 27.5.1–27.5.4, 2010. 24

[29] Dinesh Somasekhar, Yibin Ye, Paolo Aseron, Shih-Lien Lu, Muhammad M. Khellah, Jason Howard, Greg Ruhl, Tanay Karnik, Shekhar Borkar, Vivek K. De, and Ali Keshavarzi. 2 ghz 2 mb 2t gain cell memory macro with 128 gbytes/sec bandwidth in a 65 nm logic process technology. *IEEE Journal of Solid-State Circuits*, 44(1):174–185, 2009. 24

[30] Robert Giterman, Adam Teman, Pascal Meinerzhagen, Andreas Burg, and Alexander Fish. 4t gain-cell with internal-feedback for ultra-low retention power at scaled cmos nodes. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2177–2180, 2014. 24

[31] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2), May 2013. 25

[32] Zhitao Diao, Zhanjie Li, Shengyuang Wang, Yunfei Ding, Alex Panchula, Eugene Chen, Lien-Chang Wang, and Yiming Huai. Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. *Journal of Physics: Condensed Matter*, 19(16):165209, apr 2007. 25

[33] Wangyuan Zhang and Tao Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *MICRO*, pages 2–13, 2009. 25

[34] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009. 25

[35] Young-Bae Kim, Seung Ryul Lee, Dongsoo Lee, Chang Bum Lee, Man Chang, Ji Hyun Hur, Myoung-Jae Lee, Gyeong-Su Park, Chang Jung Kim, U-In Chung, In-Kyeong Yoo, and Kinam Kim. Bi-layered rram with unlimited endurance and extremely uniform switching. In *2011 Symposium on VLSI Technology - Digest of Technical Papers*, pages 52–53, 2011. 25

[36] Elham Cheshmikhani, Hamed Farbeh, and Hossein Asadi. Enhancing reliability of STT-MRAM caches by eliminating read disturbance accumulation. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 854–859. IEEE, 2019. 25, 51

[37] Rujia Wang, Lei Jiang, Youtao Zhang, Linzhang Wang, and Jun Yang. Selective restore: an energy efficient read disturbance mitigation scheme for future STT-MRAM. In *DAC*, 2015. 25, 50, 103

[38] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61, 2011. 26

[39] Kyle Kuan and Tosiron Adegbija. Lars: Logically adaptable retention time stt-ram cache for embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 461–466, 2018. 26, 52

[40] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 134–145, 2007. 28

[41] J. Kong et al. Towards refresh-optimized edram-based caches with a selective fine-grain round-robin refresh scheme. *Microprocessors and Microsystems*, 49:95 – 104, 2017. 28, 39

[42] A. Agrawal et al. Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies. In *HPCA-2013*, pages 400–411, Feb 2013. 39, 76

[43] A. Agrawal et al. Mosaic: Exploiting the spatial locality of process variation to reduce refresh energy in on-chip edram modules. In *HPCA-2014*, pages 84–95, Feb 2014. 39

[44] Mohammad Alizadeh, Adel Javanmard, Shang-Tse Chuang, Sundar Iyer, and Yi Lu. Versatile refresh: low complexity refresh scheduling for high-throughput multi-banked edram. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 247–258, 2012. 40, 56

[45] Z. Jaksic and R. Canal. Dram-based coherent caches and how to take advantage of the coherence protocol to reduce the refresh energy. In *DATE-2014*, March 2014. 40

[46] Alejandro Valero, Salvador Petit, Julio Sahuquillo, David R. Kaeli, and José Duato. A reuse-based refresh policy for energy-aware edram caches. *Microprocessors and Microsystems*, 39(1):37–48, 2015. 40

[47] Amit Kazimirsky and Shmuel Wimer. Opportunistic refreshing algorithm for edram memories. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(11):1921–1932, 2016. 41

[48] T. Kirihata, P. Parries, D.R. Hanson, Hoki Kim, J. Golz, G. Fredeman, R. Rajeevakumar, J. Griesemer, N. Robson, A. Cestero, B.A. Khan, Geng Wang, M. Wordeman, and S.S. Iyer. An 800-mhz embedded dram with a concurrent refresh mode. *IEEE Journal of Solid-State Circuits*, 40(6):1377–1387, 2005. 41

[49] S. Mittal. A cache reconfiguration approach for saving leakage and refresh energy in embedded DRAM caches. *CoRR*, abs/1309.7082, 2013. 41

[50] K Patel et al. Energy-efficient value-based selective refresh for embedded drams. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 466–476, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 41

[51] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. Improving energy efficiency of embedded dram caches for high-end computing systems. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 99–110, New York, NY, USA, 2014. Association for Computing Machinery. 42

[52] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 134–145, 2007. 42

[53] Young-Ho Gong, Jae Min Kim, Sung Kyu Lim, and Sung Woo Chung. Exploration of temperature-aware refresh schemes for 3d stacked edram caches. *Microprocessors and Microsystems*, 42:100–112, 2016. 43

[54] Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C.Hunter, and Lizy K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 375–384, 2010. 43, 44

[55] J. Liu et al. Raidr: Retention-aware intelligent dram refresh. In *ISCA-2012*, June 2012. 44

[56] Prashant J. Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. Refresh pausing in dram memory systems. *ACM Trans. Archit. Code Optim.*, 11(1), feb 2014. 44

[57] Alejandro Valero, Julio Sahuquillo, Salvador Petit, Vicente Lorente, Ramon Canal, Pedro López, and José Duato. An hybrid edram/sram macrocell to implement first-level data caches. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 213–221, 2009. 45

[58] Javier Lira, Carlos Molina, David Brooks, and Antonio Gonzalez. Implementing a hybrid sram / edram nuca architecture. In *2011 18th International Conference on High Performance Computing*, pages 1–10, 2011. 45

[59] Joonho Kong and Young-Ho Gong. An efficient trade-off between yield and energy for edram caches under process variations. *Microprocessors and Microsystems*, 55:1–12, 2017. 45

[60] M. M. Islam et al. Zero-value caches: Cancelling loads that return zero. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 237–245, Sept 2009. 46

[61] A.R. Alameldeen and D.A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 212–223, 2004. 46

[62] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, page 214–220, New York, NY, USA, 2000. Association for Computing Machinery. 46

[63] Fabian Oboril, Fazal Hameed, Rajendra Bishnoi, Ali Ahari, Helia Naeimi, and Mehdi Tahoori. Normally-off stt-mram cache with zero-byte compression for energy efficient last-level caches. ISLPED '16, page 236–241, New York, NY, USA, 2016. Association for Computing Machinery. 47

[64] Julien Dusser et al. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 46–55, New York, NY, USA, 2009. ACM. 47

[65] Jun Yang et al. Frequent value locality and its applications. *ACM Trans. Embed. Comput. Syst.*, 1(1):79–105, November 2002. 47

[66] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, page 214–220, New York, NY, USA, 2000. Association for Computing Machinery. 48

[67] Yen-Jen Chang, Feipei Lai, and Chia-Lin Yang. Zero-aware asymmetric sram cell for reducing cache power in writing zero. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):827–836, 2004. 48

[68] Jinwook Jung, Yohei Nakata, Masahiko Yoshimoto, and Hiroshi Kawaguchi. Energy-efficient spin-transfer torque ram cache exploiting additional all-zero-data flags. In *International Symposium on Quality Electronic Design (ISQED)*, pages 216–222, 2013. 48

[69] Yen-Jen Chang and Feipei Lai. Dynamic zero-sensitivity scheme for low-power cache memories. *IEEE Micro*, 25(4):20–32, 2005. 49

[70] Zhenyu Sun, Hai Li, and Wenqing Wu. A dual-mode architecture for fast-switching STT-RAM. In *ISLPED*, pages 45–50, 2012. 49

[71] Sparsh Mittal, Jeffrey Vetter, and Lei Jiang. Addressing read-disturbance issue in stt-ram by data compression and selective duplication. *IEEE Computer Architecture Letters*, 16(2):94–98, Jul-Dec 2017. 49

[72] Lei Jiang, Wujie Wen, Danghui Wang, and Lide Duan. Improving Read Performance of STT-MRAM based Main Memories through Smash Read and Flexible Read. *ASP-DAC*, 2016. 50

[73] Hang Zhang, Xuhao Chen, Nong Xiao, Fang Liu, and Zhiguang Chen. RED-SHIELD: Shielding read disturbance for STT-RAM based register files on GPUs. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pages 389–392, 2016. 50

[74] Fateme S Hosseini and Chengmo Yang. Compiler-Directed and Architecture-Independent Mitigation of Read Disturbance Errors in STT-RAM. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 222–227, 2019. 50

[75] Elham Cheshmikhani, Hamed Farbeh, and Hossein Asadi. 3RSeT: Read Disturbance Rate Reduction in STT-MRAM Caches by Selective Tag Comparison. *IEEE Transactions on Computers*, 2021. 51

[76] Zhenyu Sun, Xiuyuan Bi, Hai Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 329–338, 2011. 51, 133, 134, 137, 152, 159

[77] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps. In *DAC Design Automation Conference 2012*, pages 243–252, 2012. 51

[78] Kyle Kuan and Tosiron Adegbija. Halls: An energy-efficient highly adaptable last level stt-ram cache for multicore systems. *IEEE Transactions on Computers*, 68(11):1623–1634, 2019. 52, 134, 152, 153, 158

[79] Kyle Kuan and Tosiron Adegbija. Mirrorcache: An energy-efficient relaxed retention l1 sttram cache. GLSVLSI '19, page 299–302, New York, NY, USA, 2019. Association for Computing Machinery. 52

[80] Jungwoo Park, Myoungjun Lee, Soontae Kim, Minho Ju, and Jeongkyu Hong. Mh cache: A multi-retention stt-ram-based low-power last-level cache for mobile hardware rendering systems. 16(3), jul 2019. 52

[81] Farzane Rabiee, Mostafa Kajouyan, Newsha Estiri, Jordan Fluech, Mahdi Fazeli, and Ahmad Patooghy. Enduring non-volatile l1 cache using low-retention-time sttram cells. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 322–327, 2020. 52

[82] S. Mittal, J. S. Vetter, and D. Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015. 55

[83] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 74–85, 2005. 56, 89, 114

[84] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. *SIGARCH Comput. Archit. News*, 28(5):150–159, nov 2000. 56

[85] Julien Dusser et al. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 46–55, New York, NY, USA, 2009. ACM. 56, 114, 138

[86] Yen-Jen Chang, Chia-Lin Yang, and Feipei Lai. Value-conscious cache: Simple technique for reducing cache access power. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10016–, Washington, DC, USA, 2004. IEEE Computer Society. 56, 138

[87] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. 57, 72, 89, 95, 152

[88] M. T. Chang et al. Technology comparison for large last-level caches (l3cs): Low-leakage sram, low write-energy stt-ram, and refresh-optimized edram. In *HPCA-2013*, pages 143–154, Feb 2013. 73

[89] J. Barth, W. R. Reohr, P. Parries, G. Fredeman, J. Golz, S. E. Schuster, R. E. Matick, H. Hunter, C. C. Tanner, J. Harig, H. Kim, B. A. Khan, J. Griesemer, R. P. Havreluk, K. Yanagisawa, T. Kirihata, and S. S. Iyer. A 500 mhz random cycle, 1.5 ns latency, soi embedded dram macro featuring a three-transistor micro sense amplifier. *IEEE Journal of Solid-State Circuits*, 43(1):86–95, Jan 2008. 73, 95

[90] Joonho Kong, Young-Ho Gong, and Sung Woo Chung. Towards refresh-optimized edram-based caches with a selective fine-grain round-robin refresh scheme. *Microprocessors and Microsystems*, 49:95–104, 2017. 73

[91] Sandro Bartolini, Pierfrancesco Foglia, and Cosimo Antonio Prete. Exploring the relationship between architectures and management policies in the design of nuca-based chip multicore systems. *Future Generation Computer Systems*, 78:481–501, 2018. 73

[92] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A nuca substrate for flexible cmp cache sharing. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):1028–1040, 2007. 73

[93] Sparsh Mittal, Rujia Wang, and Jeffrey Vetter. Destiny: A comprehensive tool with 3d and multi-level cell memory modeling capability. *Journal of Low Power Electronics and Applications*, 7(3), 2017. 73

[94] C. Bienia et al. The parsec benchmark suite: Characterization and architectural implications. In *PACT-2008*, pages 72–81, Oct 2008. 73, 95

[95] M. Lodde et al. Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols. In *Euro-Par 2012 Parallel Processing*, pages 206–218, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 87

[96] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009. 95

[97] A. Agrawal et al. Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies. In *HPCA-2013*, pages 400–411, Feb 2013. 100

[98] Sparsh Mittal, Rujia Wang, and Jeffrey Vetter. DESTINY: A Comprehensive Tool with 3D and Multi-level Cell Memory Modeling Capability. *Journal of Low Power Electronics and Applications*, 7(3):23, 2017. 103

[99] R Takemura, T Kawahara, K Ono, K Miura, H Matsuoka, and H Ohno. Highly-scalable disruptive reading scheme for Gb-scale SPRAM and beyond. In *IEEE IMW*, pages 1–2, 2010. 103, 104, 106, 118

[100] R. Bishnoi, M. Ebrahimi, F. Oboril, and M. B. Tahoori. Read disturb fault detection in stt-mram. In *2014 International Test Conference*, pages 1–7, 2014. 104, 106

[101] Rujia Wang, Lei Jiang, Youtao Zhang, Linzhang Wang, and Jun Yang. Selective restore: an energy efficient read disturbance mitigation scheme for future STT-MRAM. In *DAC*, 2015. 104, 118, 128

[102] Elham Cheshmikhani, Amir Mahdi Hosseini Monazzah, Hamed Farbeh, and Seyed Ghassem Miremadi. Investigating the effects of process variations and system workloads on reliability of STT-RAM caches. In *12th European Dependable Computing Conference (EDCC)*, pages 120–129, 2016. 104

[103] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches. *IEEE TPDS*, 2014. 104

[104] Sukarn Agarwal and Hemangee K Kapoor. Improving the lifetime of non-volatile cache by write restriction. *IEEE Transactions on Computers*, 68(9):1297–1312, 2019. 104

[105] Lei Jiang, Wujie Wen, Danghui Wang, and Lide Duan. Improving Read Performance of STT-MRAM based Main Memories through Smash Read and Flexible Read. *ASP-DAC*, 2016. 104, 118

[106] Wang Kang, Yuanqing Cheng, Youguang Zhang, Dafine Ravelosona, and Weisheng Zhao. Readability challenges in deeply scaled STT-MRAM. In *NVMTS*, pages 1–4, 2014. 104

[107] Sparsh Mittal. A survey of soft-error mitigation techniques for non-volatile memories. *Computers*, 6(8), 2017. 104

[108] Elham Cheshmikhani, Hamed Farbeh, and Hossein Asadi. A system-level framework for analytical and empirical reliability exploration of STT-MRAM caches. *IEEE Transactions on Reliability*, 69(2):594–610, 2019. 106

[109] Xuanyao Fong, Yusung Kim, Sri Harsha Choday, and Kaushik Roy. Failure mitigation techniques for 1T-1MTJ spin-transfer torque MRAM bit-cells. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(2):384–395, 2013. 106

[110] Sparsh Mittal, Jeffrey Vetter, and Lei Jiang. Addressing read-disturbance issue in stt-ram by data compression and selective duplication. *IEEE Computer Architecture Letters*, 16(2):94–98, Jul-Dec 2017. 109

[111] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, and Yuan Xie. Power and performance of read-write aware hybrid caches with non-volatile memories. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 737–742. IEEE, 2009. 113, 118, 120, 126

[112] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Write intensity prediction for energy-efficient non-volatile caches. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 223–228. IEEE, 2013. 113, 118, 120, 126

[113] Jun Yang and Rajiv Gupta. Frequent value locality and its applications. *ACM Trans. Embed. Comput. Syst.*, 1(1):79–105, November 2002. 114

[114] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. 117

[115] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. 117, 152

[116] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Base-delta-immediate compression: practical data compression for on-chip caches. In *PACT*, 2012. 118, 127

[117] Sukarn Agarwal and Hemangee K Kapoor. Improving the performance of hybrid caches using partitioned victim caching. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(1):1–27, 2020. 125

[118] Kyle Kuan and Tosiron Adegbija. Lars: Logically adaptable retention time stt-ram cache for embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 461–466, 2018. 134, 159

[119] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, and Yuan Xie. Power and performance of read-write aware hybrid caches with non-volatile memories. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 737–742, 2009. 134

[120] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61, 2011. 134, 135, 137, 159

[121] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps. In *DAC Design Automation Conference 2012*, pages 243–252, 2012. 138, 152, 158

[122] Kyle Kuan and Tosiron Adegbija. Mirrorcache: An energy-efficient relaxed retention l1 sttram cache. GLSVLSI '19, page 299–302, New York, NY, USA, 2019. Association for Computing Machinery. 138, 159

[123] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, Yiran Chen, and Yanxiang He. Compiler-assisted refresh minimization for volatile stt-ram cache. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 273–278, 2013. 138

[124] Keni Qiu, Junpeng Luo, Zhiyao Gong, Weigong Zhang, Jing Wang, Yuanchao Xu, Tao Li, and Chun Jason Xue. Refresh-aware loop scheduling for high performance low power volatile stt-ram. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 209–216, 2016. 138

[125] Farzane Rabiee, Mostafa Kajouyan, Newsha Estiri, Jordan Fluech, Mahdi Fazeli, and Ahmad Patooghy. Enduring non-volatile l1 cache using low-retention-time sttram cells. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 322–327, 2020. 138

[126] M. M. Islam et al. Zero-value caches: Cancelling loads that return zero. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 237–245, Sept 2009. 138

[127] Wangyuan Zhang and Tao Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *MICRO*, pages 2–13, 2009. 138

[128] Sparsh Mittal, Jeffrey S. Vetter, and Lei Jiang. Addressing read-disturbance issue in stt-ram by data compression and selective duplication. *IEEE Computer Architecture Letters*, 16(2):94–98, 2017. 139

[129] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012. 152, 160

[130] Sparsh Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 4(1):33–43, 2014. 158, 159

[131] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Dasca: Dead write prediction assisted stt-ram cache architecture. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, 2014. 159

[132] Zhe Wang, Daniel A. Jiménez, Cong Xu, Guangyu Sun, and Yuan Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2014. 159

List of publications

Journals

1. Sheel Sindhu Manohar, Sparsh Mittal, Hemangee K. Kapoor: CORIDOR: using COherence and tempoRal localIty to mitigate read Disurbance errOR in STT-RAM caches, September 2021ACM Transactions on Embedded Computing Systems DOI: 10.1145/3484493

2. Sheel Sindhu Manohar, Hemangee K. Kapoor: Dynamic reconfiguration of embedded-DRAM caches employing zero data detection based refresh optimisation. J. Syst. Archit. 100 (2019)

3. Sheel Sindhu Manohar, Hemangee K. Kapoor, "CAPMIG: Coherence Aware block Placement and MIGration in Multi-retention STT-RAM Caches", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems , 2022.

Conference proceedings

1. Sheel Sindhu Manohar, Sukarn Agarwal, Hemangee K. Kapoor: Towards Optimizing Refresh Energy in embedded-DRAM Caches using Private Blocks. ACM Great Lakes Symposium on VLSI 2019: 225-230

2. Sheel Sindhu Manohar, Hemangee K. Kapoor: Refresh optimised embedded-dram caches based on zero data detection. ACM SAC 2019: 635-642

* * *