Políticas Eficientes de Reemplazamiento Caché para Memorias Racetrack

Efficient Cache Replacement Policies for Racetrack Memories

Trabajo Fin de Grado
Curso 2024-2025

Autor
Jiahui You

DIRECTORES
Dr. Fernando Castro Rodríguez
Dr. Roberto Rodríguez-Rodríguez

Calificación otorgada por el Tribunal: 10.0 sobre 10

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Políticas Eficientes de Reemplazamiento Caché para Memorias Racetrack

Efficient Cache Replacement Policies for Racetrack Memories

Trabajo de Fin de Grado en Ingeniería Informática

Autor

Jiahui You

DIRECTORES

Dr. Fernando Castro Rodríguez

Dr. Roberto Rodríguez-Rodríguez

Convocatoria: 09 Junio 2025

Calificación otorgada por el Tribunal: 10.0 sobre 10

Grado en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

26 de mayo de 2025

DEDICATION

This final degree project is dedicated to my two academic advisors, **Prof. Fernando** and **Prof. Roberto**, who have guided and encouraged me throughout my academic journey.

I would like to express my **special gratitude to Prof. Fernando**. I had the privilege of attending his class in my first year, and I was deeply impressed by his rigorous explanations, dedicated attitude, and tireless work ethic.

What started as a simple wish—"I hope he could be my final degree project advisor one day"—eventually came true.

He has not only been my research mentor, but also a scholar I deeply trust, admire, and aspire to learn from.

ACKNOWLEDGEMENTS

It has been a great honor and privilege to complete this final degree project under the patient guidance and support of my two advisors. I would like to express my sincerest gratitude to both of them for accompanying me throughout this journey of academic and personal growth.

First and foremost, I would like to thank **Prof. Fernando** for his clear direction on the research topic, and for his patient and detailed guidance in methodology, technical implementation, and final degree project writing. His rigorous academic attitude, sharp insight, and deep professional knowledge have deeply inspired and motivated me.

I would also like to thank **Prof. Roberto** for his valuable advice and technical support throughout the research process, especially for his insightful suggestions during the key stages of system implementation, which provided a solid foundation for the smooth progress of this project.

Additionally, I am grateful to the university for providing the experimental resources and computing platform that ensured a stable and reliable environment for this study. I would also like to thank my classmates and friends for their encouragement and assistance, and my family for their unwavering support over the years.

This final degree project is the result of collective wisdom, guidance, and help from many people. I would like to express my deepest respect and heartfelt appreciation to all of them.

RESUMEN

Este Trabajo de Fin de Grado presenta un estudio detallado sobre la memoria Racetrack (RTM), una tecnología emergente no volátil, y su integración en la jerarquía de memoria de nivel L2 en arquitecturas de procesadores. El objetivo principal es analizar el impacto de distintas políticas de reemplazo —como LRU, FIFO, BRRIP y una versión modificada de BRRIP— sobre el número de desplazamientos ("shifts") requeridos en RTM. Se modificó el simulador gem5 para emular los comportamientos específicos de RTM, incluyendo la posición del puerto de acceso y la latencia asociada a cada operación. Las simulaciones se realizaron con benchmarks de SPEC CPU2017. Los resultados muestran que la política propuesta reduce significativamente la latencia total y el número de shifts, mejorando así el rendimiento y la eficiencia energética del sistema.

Palabras clave: Memoria Racetrack, BRRIP, BRRIP-mod, gem5, caché L2, política de reemplazamiento, desplazamiento, memoria no volátil, simulación, SPEC CPU2017

ABSTRACT

This Final Degree Project explores the design and evaluation of cache replacement policies tailored for Racetrack Memory (RTM), a novel non-volatile memory technology characterized by its high density and low power consumption. The study integrates RTM into the L2 cache of a system simulated using gem5, a cycle-accurate simulator. Multiple classical and custom replacement policies—including LRU, FIFO, BRRIP, MRU, and a proposed shift-aware BRRIP-mod—are implemented and evaluated based on metrics such as total shift count, cache miss rate, and latency. Benchmarks from the SPEC CPU2017 suite are used for performance analysis. The results demonstrate that the proposed policy significantly reduces both shift cost and overall access latency while maintaining competitive miss rates, thus proving its suitability for RTM-based cache systems.

Keywords: Racetrack Memory, BRRIP, BRRIP-mod, gem5, L2 cache, cache replacement policy, shift cost, non-volatile memory, simulation, SPEC CPU2017

Contents

List of Figures

Chapter 1 - Introduction

This chapter introduces the motivation for studying **Racetrack Memory (RTM)**, also known as **Domain-Wall Memory (DWM)**, a novel non-volatile memory technology. It emphasizes its advantages over traditional SRAM and DRAM—particularly in second-level (L2) caches—such as higher storage density, lower energy consumption, and data persistence. The objectives of this research are clearly defined: to evaluate how different cache replacement policies influence data shift operations in RTM-based caches. To achieve these goals, we modified the gem5 simulator to accurately represent the physical structure and behavior of RTM in the L2 cache, and conducted benchmark experiments using the SPEC CPU2017 suite on several classical strategies including LRU, FIFO, BRRIP, and MRU. These objectives form the core contributions of this project. Finally, the research workflow is summarized, including a literature review, adaptation of gem5, benchmark integration, and performance evaluation.

1.1 Motivation

Racetrack Memory (RTM) is an emerging non-volatile memory technology that is gradually becoming a promising candidate to replace traditional cache memory solutions. To better understand its potential, we can first look at the conventional memory cache commonly used in computer systems. The memory cache serves as a small, high-speed temporary storage unit, typically implemented using Static Random Access Memory (SRAM). Each bit in an SRAM cell is stored using a group of transistors, and the data remains valid as long as the system is powered, eliminating the need for periodic refreshing. While this architecture offers low access latency, it also presents

several limitations: high power consumption—especially pronounced in larger cache levels such as L2 and L3, low storage density, and volatility, meaning data is lost once power is removed.

In contrast, RTM offers several significant advantages. Fundamentally, RTM stores and accesses data by shifting magnetic domains along a nanoscale magnetic wire. Each magnetic domain can store multiple bits, providing much higher storage density compared to the one-bit-per-cell structure of SRAM. Moreover, instead of using electrical charge to represent binary values as in SRAM, RTM uses the direction of magnetization, allowing data to persist even when the system is powered off.

In summary, due to its structural efficiency and lower power consumption, Racetrack Memory (RTM) demonstrates greater flexibility and potential for replacing traditional SRAM in L2 and L3 cache applications. Although some studies have questioned the feasibility of applying RTM technology in Last-Level Caches (LLCs), no prior work has systematically evaluated the performance of classical replacement policies in RTM-based LLCs. This project aims to fill this research gap by simulating and analyzing the behavior of different replacement policies in the context of RTM, and assessing their impact on cache efficiency.

1.2 Objectives

The specific objectives of this work are as follows:

- First, to accurately simulate the shift operations during data access by modifying the baseline implementation of L2 cache storage in the gem5 simulator, aligning its logical structure with the characteristics of Racetrack Memory (RTM).
- Second, to evaluate the performance of various cache replacement policies within RTM caches using the SPEC CPU2017 benchmark suite. This involves

recording the number of shift operations generated during read and write accesses at the L2 cache under both hit and miss scenarios, and analyzing the impact of different replacement strategies on the total number of shifts and access latency.

- The ultimate goal is to identify more efficient and energy-saving replacement schemes, providing valuable insights for the future design of RTM-based cache architectures.

1.3 Contributions

Under the supervision of the advisor, the author has completed several key technical and research tasks throughout this final degree project. The main contributions are summarized as follows:

- Gained an in-depth understanding of the internal architecture of the gem5 simulator and, with the advisor's guidance, independently implemented a behavioral model for Racetrack Memory (RTM). This included developing mechanisms for shift operation simulation, access port position tracking, and latency calculation.
- Modified the underlying C++ code of gem5 to implement a custom L2 cache structure based on RTM, incorporating accurate tracking of shift overhead during both hit and miss events in read/write operations.
- Designed and implemented a novel replacement strategy, BRRIP-mod, based on the BRRIP baseline, which introduces a shift-aware victim selection mechanism aimed at reducing total shift cost in RTM caches.

- Successfully set up and configured the SPEC CPU2017 benchmark platform, and conducted extensive simulation campaigns under various cache configurations. The author independently carried out the evaluation of multiple replacement strategies (LRU, FIFO, BRRIP, MRU, and BRRIP-mod) under the RTM model, and analyzed the results in terms of shift count, miss rate, and access latency.
- Drafted the majority of the sections related to technical implementation, experimental setup, and results analysis, and performed the final revisions based on the advisor's feedback.

All source code modifications and experimental configurations were fully developed and tested by the author.

1.4 Work plan

To achieve the proposed research objectives and ensure the validity of the results, the project is organized into several key phases. These include understanding the architecture and storage mechanisms of Racetrack Memory, analyzing and adapting the gem5 simulator, and configuring the experimental setup using the SPEC CPU2017 benchmark suite. This section outlines the overall work plan, following a progression from theoretical foundations to practical implementation and performance evaluation.

1.4.1 Theoretical Study and Literature Review

- Conducted an extensive review of existing literature to systematically understand the fundamentals of Racetrack Memory (RTM), including its advantages and

disadvantages, storage mechanism (magnetic domain wall motion), access characteristics, and primary sources of energy consumption.

- Gained familiarity with the underlying codebase of the gem5 simulator, with a particular focus on cache storage modules, and studied the structural and behavioral differences and similarities between traditional cache memory and RTM to facilitate subsequent modifications.
- Investigated the characteristics of the SPEC CPU2017 benchmark suite, analyzing the purpose and configuration data of each application, in order to prepare for future performance evaluations within the gem5 simulation environment.

1.4.2 System Design and Development

During memory initialization, differentiated the L2 cache from other storage units and modified the storage logic specifically for the L2 cache. In the low-level code, implemented mechanisms to calculate the number of shifts based on whether an L2 cache access results in a hit or a miss.

Modified parts of the cache access handling logic in the gem5 simulator by introducing a pointer for each L2 cache block to emulate the *access port* behavior of Racetrack Memory (RTM). This design enables accurate calculation of the number of shifts and distinguishes the shift costs associated with different types of access operations (hit/miss and read/write).

1.4.3 Testing and Evaluation

- Configured and executed the SPEC CPU2017 speed subset, collecting simulation data to analyze the impact of different cache replacement policies on key metrics such as shift count, access latency within Racetrack Memory (RTM) caches.
- During the execution of each benchmark, recorded the total number of shifts, average access latency, total shifts under L2 cache hits and misses, as well as other critical performance indicators.
- Generated relevant comparison charts (e.g. trends of shift counts under different replacement policies) and conducted systematic analysis and summarization of the results, providing insights for future optimization efforts.

1.4.4 Results Summary and Future Work

- Analyzed the simulation results to identify the classical cache replacement policy that best suits Racetrack Memory (RTM)-based cache systems.
- Based on the characteristics of the selected policy, proposed potential optimization directions, such as further refining the replacement strategy to better adapt it to RTM architecture.
- Provided insights and future research directions for the continued development of RTM cache management techniques.

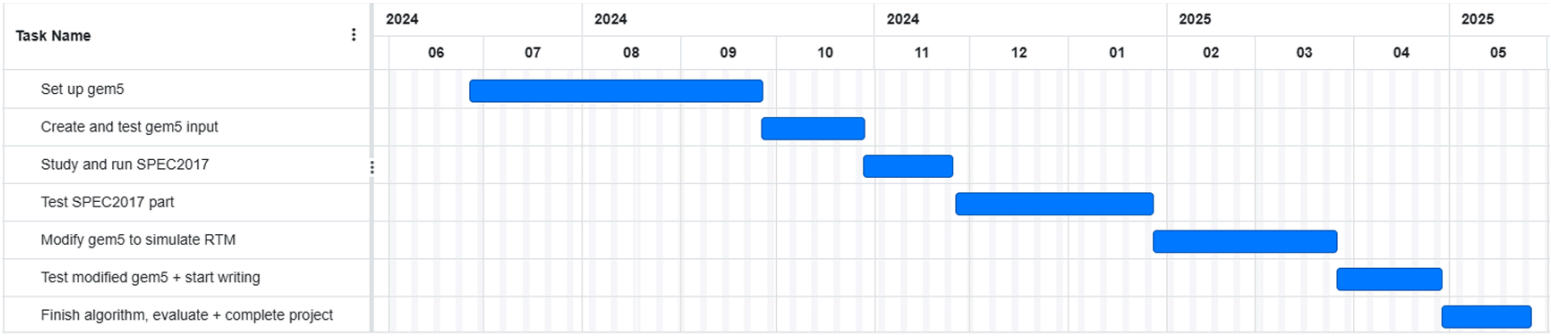

Figure 1.1. Gantt chart of the Final Degree Project.

Chapter 2 - Background and State of the Art

This chapter provides an overview of the technological background and prior work relevant to this study. First, we present an in-depth explanation of Racetrack Memory (RTM), including its definition, working principle, and cache-level organization, expanding on the introductory discussion from Chapter 1. Next, we review conventional cache replacement policies such as LRU, FIFO, MRU and BRRIP, which serve as the foundation for later comparisons. Finally, we focus on RTM-specific replacement strategies, particularly the 2023 Shift-Aware Replacement (SAR) policy, and analyze its approach to minimizing shift cost. Together, these sections establish the theoretical framework necessary to understand the design choices and evaluation presented in later chapters.

2.1 Racetrack memory

To understand the cache system architecture used in this research, it is essential to begin with a comprehensive explanation of **Racetrack Memory (RTM)**, a promising non-volatile memory technology. This section provides an in-depth overview of RTM, covering its definition, data access principles, and system-level organization. By examining both the physical operation and architectural integration of RTM, we highlight its key characteristics—particularly its serial access behavior—and the unique challenges it introduces compared to traditional memory technologies such as SRAM and DRAM.

2.1.1 Brief Definition (What is RTM)

Racetrack Memory (RTM) is an emerging non-volatile memory technology first proposed by Stuart Parkin and his colleagues [1]. It operates by moving magnetic

domain walls within a nanowire to perform read and write operations(a *nanowire* is a nanoscale magnetic wire composed of multiple magnetic domains, each capable of storing one bit of information via its magnetization direction. Unlike traditional storage technologies that rely on mechanical contact or volatile circuits, RTM utilizes current-driven domain wall motion, enabling contactless data access. This structure eliminates physical friction, significantly enhancing the durability and reliability of the device, and extends its operational lifespan.

2.1.2 Operating Principle (How Data is Accessed)

Racetrack Memory (RTM) is based on the motion of magnetic domain walls. Its core unit consists of a nanowire composed of multiple magnetic domains, where the magnetization direction of each domain represents a binary bit — magnetization pointing upward denotes bit '1', and downward denotes bit '0'. An example of a nanowire with eight magnetic domains (K = 8) is shown in Figure 2.1[2].

These magnetic domains are separated by domain walls and are sequentially aligned along the magnetic track. Instead of directly addressing a specific location, data read and write operations are performed by applying current pulses that shift the domain walls along the track. This motion continues until the target bit aligns precisely with a fixed access port (as shown in Figure 2.1). This process is referred to as a **shift** operation.

In RTM, each shift — the movement of domain walls along the racetrack — is typically triggered by ultra-short current pulses, often on the nanosecond scale. These pulses rely on a physical phenomenon known as the spin-transfer torque effect. The basic principle is that a spin-polarized current (i.e., a stream of electrons with aligned

spin direction) transfers angular momentum to the magnetic structure in the material, such as domain walls. This momentum transfer exerts a force on the domain wall, causing it to move along the track. Every time a current pulse is applied, the spin-polarized current pushes the domain walls forward, enabling a read or write operation. Since this happens in a very short timespan, RTM achieves high-speed data access [1].

- **Best case**: the target bit is already aligned with the access port (shift cost = 0).
- **Worst case**: the target bit is at the far end of the track, requiring K−1 shifts.

This structure introduces a distinctly serial access nature. Unlike traditional SRAM or DRAM, which support random access to any address [3], every RTM access inherently involves a shift operation. This becomes a potential bottleneck in high-access-frequency scenarios.

As a result, many research efforts aim to reduce the average number of shifts through techniques such as pre-shifting [8,9], data placement optimization [10,11,12], or shift-aware replacement policies [2], all of which aim to improve RTM's viability in high-performance cache systems [2].

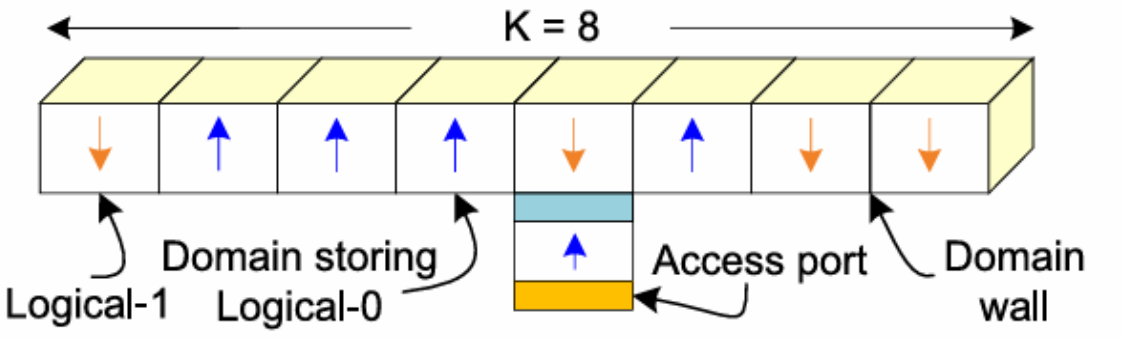

Figure 2.1. Structure of a basic Racetrack Memory cell with K = 8. Adapted from [2].

2.1.3 Macro Architecture: RTM Memory Cell and Cache Structure

As previously discussed, Racetrack Memory (RTM) is not only a novel storage medium but also differs significantly in its internal organization compared to traditional SRAM or DRAM-based caches.

To understand the overall structure, RTM's fundamental storage unit is a nanoscale magnetic track, composed of multiple magnetic domains that can each independently store a bit of information. These domains encode data via their magnetization direction, and each track can hold multiple bits — for example, $k = 32$ in Figure 2.2, indicating 32 bits per track[2]. Data is read or written by aligning the target magnetic domain to a fixed access port through current pulses.

In cache systems, multiple tracks are typically grouped into a logical block called a Domain Block Cluster (DBC). Each DBC contains T magnetic tracks, with each track storing K bits. As shown in Figure 2.2 [2], DBC 0 consists of 512 tracks (cache lines), with each track containing 32 bits.

All tracks within a DBC share a common access port position pointer, which records the current position of the read/write head (as indicated by the arrows in Figure 2.2). Due to this shared access head mechanism, accessing one position affects the access state of the entire track group. Therefore, optimizing the sequence of accesses is crucial to minimizing shift costs.

At the system level, RTM cache structures typically follow a hierarchical organization: multiple tracks form a DBC; multiple DBCs form a Subarray; multiple subarrays compose a Bank; and multiple banks together make up the complete RTM Cache architecture. This layered structure facilitates management and enables

large-scale scalability. In practical designs, a common configuration consists of cache sets formed by 32 tracks, each supporting a 512-bit cache line. The system tracks the current position of each set to evaluate the shift cost and to apply appropriate replacement strategies [2] (as illustrated in Figure 2.2).

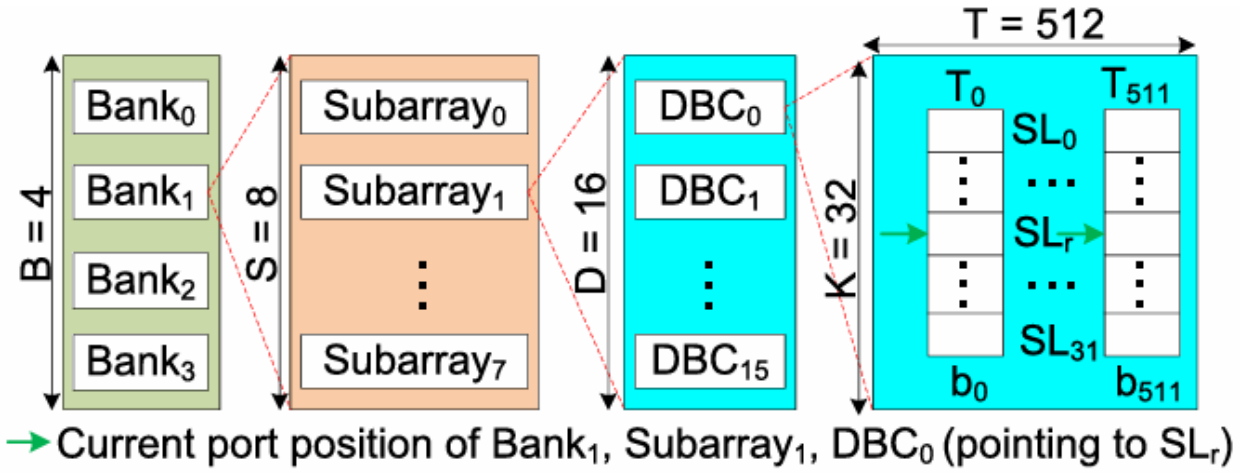

Figure 2.2. Hierarchical organization of Racetrack Memory with B = 4, S = 8, D = 16, and T = 512 tracks. Adapted from [2].

2.2 Cache Replacement Policies

Cache replacement policies play a critical role in hierarchical memory systems by determining which data should be retained or evicted when cache space is limited. This section reviews a selection of classical replacement strategies, including **LRU** [4], **FIFO** [4], **MRU** [4], and **BRRIP** [5], focusing on their design principles, performance trade-offs, and applicable usage scenarios. Although originally designed for SRAM and DRAM caches, these policies provide an essential foundation for evaluating and adapting cache management strategies in emerging memory technologies such as RTM.

2.2.1 LRU

The Least Recently Used (LRU) policy is a widely used cache replacement strategy that operates on the assumption that data not accessed recently are unlikely to be used again soon. As such, it prioritizes evicting the cache block that has not been accessed for the longest period. This policy performs well in systems where applications exhibit strong temporal locality, i.e., frequent re-access to recently used data [4].

However, in Racetrack Memory (RTM), access cost is not solely determined by cache hit or miss, but also by the distance (or number of shifts) required to align the target data with the access port. In this context, even if LRU maintains a high hit rate, it may still incur substantial access cost due to large shift distances when accessing data located far from the current port position.

Therefore, we include LRU in our evaluation as a baseline to examine whether a traditional hit-rate-oriented replacement policy remains effective under RTM's shift-cost-dominated performance model, or whether it behaves differently compared to SRAM-based caches.

2.2.2 FIFO

The First-In First-Out (FIFO) policy is a cache replacement strategy that evicts the oldest loaded cache block regardless of its access history. Its principle is simple: the block that entered the cache earliest is replaced first, making FIFO easy to implement and potentially effective in scenarios with predictable access patterns.[4]

However, similar to LRU, FIFO does not consider the physical position of data within Racetrack Memory (RTM) nor the shift cost associated with accessing it. In RTM-based architectures, access latency and energy consumption are highly

influenced by the distance (i.e., number of shifts) between the target data and the access port. As a result, although FIFO may achieve reasonable hit rates in some workloads, it can lead to suboptimal performance in shift-dominated environments like RTM caches.

2.2.3 BRRIP

Before introducing BRRIP, it is important to briefly revisit its foundational strategy—Re-reference Interval Prediction (RRIP)—including its motivation and operating principle.

Traditional cache replacement strategies such as LRU (Least Recently Used) rely on precisely tracking the access history of each cache block to determine the eviction target. While effective in many cases, LRU is costly to implement in high-associativity caches, where maintaining access order introduces significant hardware overhead. To address this, Jaleel et al. proposed a more efficient predictive replacement framework known as Re-reference Interval Prediction (RRIP) [5].

RRIP assigns each cache block a **Re-reference Prediction Value (RRPV)**, typically using 2 bits, resulting in values from **0 to 3**. Lower RRPV values (e.g., 0) indicate that a block is likely to be reused soon, while higher values (e.g., 3) suggest it is less likely to be reused and thus more likely to be replaced.

Upon cache insertion, BRRIP employs a probabilistic insertion policy: most blocks are inserted with RRPV = 3 (i.e., low priority), and with a small probability (e.g., 1 out of 64 insertions), a block is inserted with RRPV = 2 (i.e., slightly higher priority). This bimodal insertion helps reduce cache pollution from transient data.

During cache hits, the RRPV of the accessed block is reset to 0, marking it as recently reused and giving it maximum protection from replacement.

For replacement, BRRIP scans for blocks with RRPV = 3. If no such block is found, all RRPVs in the set are incremented (aging), and the search is retried. This ensures eventual victim selection while maintaining adaptability.

However, BRRIP was primarily designed to improve hit rate and performs well in traditional SRAM or DRAM-based caches. In Racetrack Memory (RTM), where access cost is heavily influenced by the shift distance between the target block and the access port, BRRIP's replacement decision does not account for physical positioning. As a result, it may frequently evict blocks located far from the access port, increasing latency and energy consumption.

Despite this limitation, BRRIP remains a structurally simple and behaviorally robust strategy with considerable research value [4]. In this study, we implement BRRIP in our RTM-based cache system and evaluate its performance in terms of shift cost.

2.2.4 MRU

The Most Recently Used (MRU) policy adopts a strategy opposite to that of LRU. It assumes that the data most recently accessed are less likely to be reused in the near future, and therefore prioritizes replacing the most recently used cache block. This policy is particularly effective in workloads where data are accessed only once and not reused—such as preprocessing steps or sequential scans [4].

However, similar to LRU and FIFO, MRU does not consider the shift distance between the target data and the access port in Racetrack Memory (RTM). In

RTM-based systems, where access cost is dominated by the number of shifts, MRU may lead to inefficient energy usage and increased latency.

Nevertheless, we include MRU in our evaluation because it serves as a conceptual counterpoint to LRU, enabling a more comprehensive understanding of how opposing replacement behaviors affect shift cost in RTM caches. By analyzing the contrast between these two extremes, we aim to gain insights into which access patterns align better with shift-sensitive architectures, thereby informing the development of more intelligent replacement strategies in the future.

2.3 Replacement Policies for RTM

In traditional SRAM or DRAM cache systems, replacement policies are primarily designed to maximize the cache hit rate, based on the assumption of fast and uniformly random memory access. Common strategies such as LRU, FIFO, and Random perform well under these conditions.

However, in Racetrack Memory (RTM), due to its serial access nature—where data must be physically shifted to the access port through domain wall motion—each replacement operation typically incurs additional latency and energy consumption. As a result, replacement policies in RTM must consider not only hit rate, but also **shift cost**, which becomes a critical design constraint absent in conventional memory architectures.

To address this challenge, Fazal Hameed et al. proposed an energy-efficient replacement strategy in 2023, known as **Shift-Aware Replacement (SAR)** [2]. SAR is explicitly tailored to the physical characteristics of RTM and aims to minimize the number of shift operations while maintaining a reasonably high hit rate. The core

innovation of SAR is to incorporate **physical shift distance awareness** into the victim selection process to improve energy efficiency.

Specifically, SAR replaces the conventional LRU logic—which always evicts the least recently used block—with a more nuanced policy. SAR first divides each cache set into two groups:

- Non-victim group: blocks that have been frequently reused and are protected from eviction;
- Victim group: blocks with low reuse likelihood, from which eviction candidates are chosen.

Within the victim group, SAR evaluates each candidate's Cache Port Proximity index (CPP_i), which quantifies the physical distance from the candidate block to the current access port position. CPP_i is calculated as:

$$\mathbf{CPP_i = |Block\ Position - Current\ Port\ Position|}$$

The candidate block with the lowest CPP_i value—i.e., the one requiring the fewest shifts—is selected for replacement. This approach significantly reduces shift overhead while maintaining spatial locality.

Figure 2.3 illustrates this process using three examples:
(a) a standard LRU policy (Victim Group size VG = 1),
(b) SAR with VG = 2, and
(c) SAR with VG = 3.

In each case, the right-hand table lists the CPP_i and associated shift cost for all eviction candidates. As the VG size increases, the policy gains greater flexibility to avoid

evicting blocks that are far from the access port—resulting in lower total shift cost (from 13 in (a) to 6 in (c)).

SAR was evaluated using the sim-zesto simulation platform on the SPEC2006/2017 benchmark suites, with TapeCache [1,3] (an LRU-based design) used as the baseline. One of SAR's main tunable parameters is the Victim Group size (VG), typically ranging from 2 to 7. This parameter enables a trade-off between shift efficiency and hit rate:

- A larger VG allows for more shift-aware victim selection and thus reduces shift count;
- However, it may also increase the risk of evicting useful (but recently inactive) blocks, slightly increasing the miss rate.

The evaluation results demonstrate that increasing VG from 4 to 7 yields:

- 48.8% reduction in total shift cost;
- 48.1% reduction in average access latency;
- 23.2% reduction in total energy consumption;
- With only a 1.1% increase in miss rate compared to the baseline.

Ultimately, the authors identify VG = 4 as the most balanced configuration (termed *best-SAR-single*), achieving significant performance and energy gains with minimal degradation in cache hit rate. These results confirm that shift-aware victim selection, combined with tunable grouping, is highly effective for optimizing RTM cache architectures.

s.

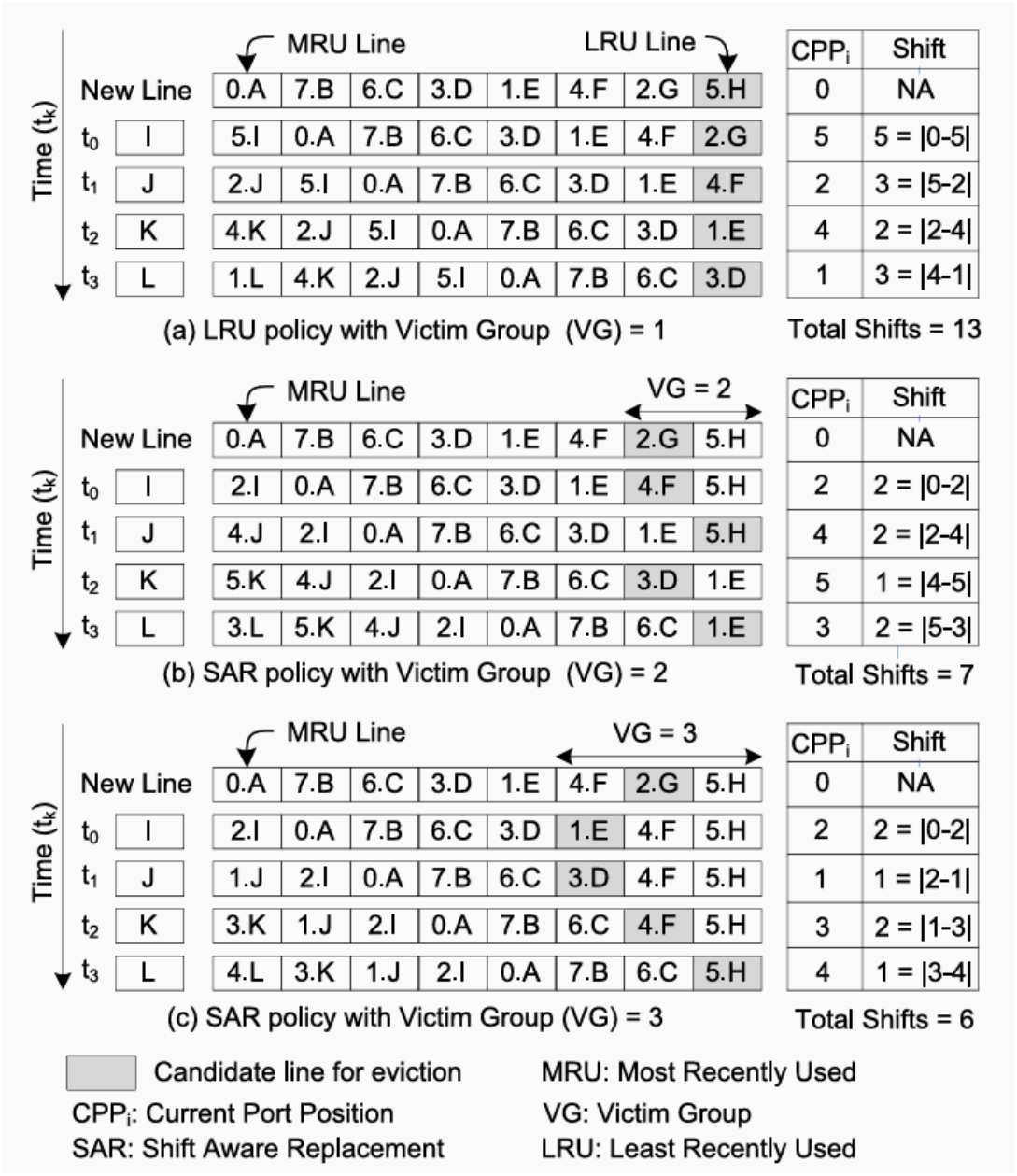

Figure 2.3. Comparison between LRU and SAR policies using different victim group sizes (VG). Adapted from [2].

Chapter 3 - Experimental Environment

This chapter describes the experimental environment and setup used to evaluate Racetrack Memory (RTM) cache systems in this study. First, we introduce the gem5 simulation platform, highlighting the key modifications made to support RTM-specific features—particularly in L2 cache replacement logic and shift cost modeling.

Next, we present the configuration of the processor model, cache hierarchy, and the selected SPEC CPU2017 benchmarks, including the rationale behind their selection and how they are used to assess the performance of various replacement policies.

Finally, we provide an overview of the hardware platform used to run the simulations, including CPU specifications, memory capacity, and computational capabilities. This ensures that the study is conducted on a reliable and scalable infrastructure, supporting reproducibility and fairness in performance comparisons.

The primary goal of this chapter is to establish a clear, repeatable, and scientifically valid simulation foundation that underpins the performance analysis and conclusions presented in the following chapters.

3.1 Introduction to the Gem5 Simulator

Gem5 [6] is an open-source, system-level computer architecture simulator widely adopted in both academia and industry. It supports multiple instruction set architectures (ISAs), including x86, ARM, RISC-V, and SPARC, as well as various processor models such as O3 (out-of-order execution), Minor (simplified pipeline), Timing (cycle-accurate), and Atomic (functional simulation). This flexibility enables Gem5 to

simulate a wide range of systems, from simple embedded processors to complex multicore architectures with high precision [6].

The Gem5 project is the result of a merger between two earlier simulators: M5 (from the University of Michigan) and GEMS (from the University of Wisconsin). As such, Gem5 offers strong modularity and extensibility, allowing researchers to customize components such as CPUs, cache hierarchies, interconnects, and memory controllers. It provides two main simulation modes:

- Syscall Emulation (SE): a fast, user-mode simulation suitable for algorithm verification and instruction-level analysis;
- Full System (FS): supports booting full operating system images, making it suitable for OS-level behavior analysis and system-wide architectural studies.

In this study, we utilize Gem5 in Syscall Emulation mode, combined with SPEC CPU2017 benchmarks, focusing on the simulation and evaluation of a Racetrack Memory (RTM) based L2 cache architecture.

Gem5's internal structure consists of a Python-based configuration layer and a C++-based simulation core. The Python scripts define the system architecture, initialize parameters, and connect components such as CPUs and caches. The C++ modules implement the low-level behavior of memory access paths, control logic, and performance/statistical tracking. Thanks to this hybrid design, Gem5 provides a powerful and flexible platform for conducting cycle-accurate architectural modeling, custom replacement policy implementation, and memory hierarchy evaluation under real instruction streams.

3.1.1 RTM Simulation Architecture: Python Configuration and Module Integration

In this study, we build a multi-level cache hierarchy that supports Racetrack Memory (RTM) using the gem5 simulator. The system architecture is constructed through modular Python scripts, with the primary configuration and component interconnection defined in the `2017.py` file. The following content does not describe source code, but rather provides a conceptual overview of the simulation system structure, focusing on the integration of RTM-based cache at the architectural level.

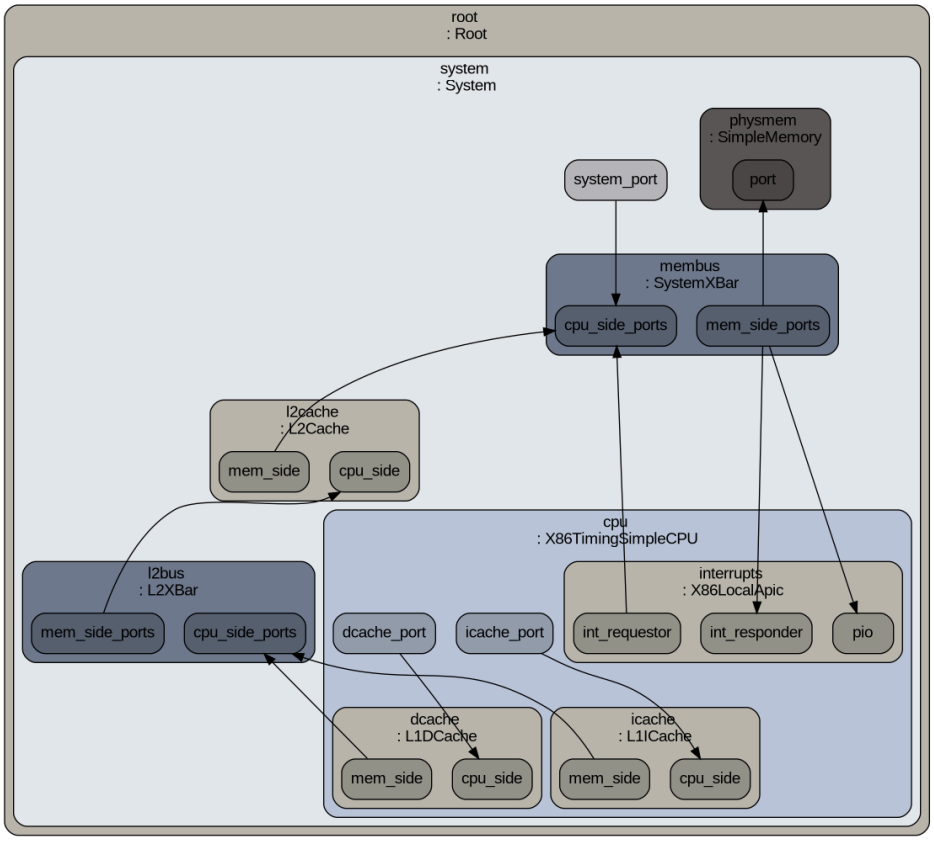

Figure 3.1. System architecture implemented in gem5 for this project. Source: Own elaboration.

As shown in the figure 3.1, the simulation platform consists of the following key components:

- **Processor model**: We use the `X86TimingSimpleCPU` core, which supports cycle-accurate modeling of instruction execution and memory access latency.
- **L1 cache hierarchy**: The system includes separated instruction (`L1ICache`) and data (`L1DCache`) caches. Each is connected to the CPU via the `connectCPU()` method, which binds the caches to the `icache_port` and `dcache_port`, respectively.
- **L2 cache**: The `L2Cache` module is a custom implementation representing the RTM cache layer. It connects to the upper-level L1 caches via the `L2XBar` (L2 crossbar) using the `connectCPUSideBus()` method, and to the memory bus (`SystemXBar`) using `connectMemSideBus()`.
- **Main memory**: A simple 8GB DRAM model is instantiated using `SimpleMemory`, which interfaces with the L2 cache through the system bus (`SystemXBar`).
- **Interrupt controller**: The `X86LocalApic` module is configured to handle external interrupt events. It is connected to the memory bus via `pio`, `int_requestor`, and `int_responder` ports.
- **Program workload**: The simulated program is defined using a `Process()` object, with its executable path and working directory configured. The workload is initialized via the `SEWorkload.init_compatible()` method to ensure compatibility with the gem5 version.

The initialization of the Python-level simulation modules follows the sequence below:

1. Initialize the `System` object and the main memory bus (`SystemXBar`);
2. Configure the CPU and L1 I/D caches, then connect them to the L2 cache through `L2XBar`;
3. Instantiate the custom `L2Cache()` module, which models the RTM cache behavior and connects it to the memory system;

4. Set up the interrupt controller and workload, then start the simulation using `m5.instantiate()`.

This structure enables the integration of replacement policies and shift-cost modeling at the L2 cache, while maintaining high flexibility for evaluating various cache strategies through the Python configuration interface.

3.1.2 Cache System Configuration and Parameter Settings

In this study, the simulated platform adopts a two-level cache hierarchy composed of split L1 instruction/data caches and a unified L2 cache. The overall configuration is inspired by the settings used by Hameed et al. in their 2023 work on cache replacement policies for Racetrack Memory (RTM) [2].

L1 Cache: L1 Cache: In our experiments, the L1 instruction cache was configured as 16KB and the data cache as 64KB, representing a slight deviation from the sizes used in [1,3] in order to explore the impact of asymmetrical L1 cache designs. Both caches were configured as 8-way set-associative structures, employing SetAssociative indexing and gem5's default BaseTags for tag storage.Access latencies were set to 2 cycles for tag lookup and 3 cycles for data access, reflecting a typical performance characteristics of SRAM-based L1 caches.

L2 Cache: As the central focus of this work, the L2 cache is modeled as an RTM-based structure to evaluate the impact of different replacement policies under shift-aware constraints. The configuration uses a 1MB capacity, 32-way set associativity, and `SetAssociative` indexing with `BaseTags`. Both data access latency and response latency are uniformly configured as 10 cycles.

It is important to clarify that these latency values do not reflect the actual physical read/write latency of RTM, but rather serve as standard Gem5 parameters to ensure simulation stability. The real RTM access overhead—particularly the latency induced by domain wall shifts—has been modeled separately in the C++ backend. This includes calculating shift costs based on port positions and incorporating additional latency for each shift operation. Further details about the read/write latency modeling in RTM are discussed in later sections.

Additionally, the L2 cache is configured with the following parameters:

- 20 Miss Status Holding Registers (MSHRs);
- 12 targets per MSHR (tgts_per_mshr = 12);
- Initial replacement policy set to LRU (Least Recently Used), using Gem5's built-in `LRURP()` module.

These settings aim to replicate the high associativity and concurrency characteristics of practical RTM cache systems, creating a more challenging and realistic testbed for evaluating replacement strategies.

The entire cache hierarchy is implemented through the Python configuration layer. In the `2017.py` script, L1 and L2 caches are properly connected to the processor and system buses. The L1 caches use the `connectCPU()` and `connectBus()` methods to interface with the CPU and the L2 crossbar (`L2XBar`). The L2 cache is connected to the memory system via `connectCPUSideBus()` and `connectMemSideBus()`, bridging the L1 caches and the system crossbar (`SystemXBar`).

This configuration provides a representative and scalable simulation environment for evaluating RTM-aware cache replacement policies, while maintaining functional stability throughout the simulation process.

3.1.3 Custom Modifications in gem5 for Shift-Aware RTM Cache Simulation

To more accurately simulate domain wall shifts caused by read and write operations in Racetrack Memory (RTM), we introduced several custom modifications to the low-level cache logic in gem5, particularly within `cache/base.cc` and `cache/tag/base_tags.cc`. These changes enable shift-aware modeling of the L2 cache and include the following key functionalities:

1. Access Port Tracking and Shift Cost Modeling

   In the L2 cache, we introduced a new array `rtmSetPointer`, which tracks the current logical access port position for each cache set. During every cache access, the simulator compares the location of the accessed block within the set against the current access port, allowing it to compute the number of required shifts.

   Specifically, in the `BaseCache::access()` function:

   - We use `BaseSetAssoc::getPossibleEntries()` to retrieve all candidate blocks within the current set and identify the index (`new_position`) of the target block.
   - The shift count is calculated as the absolute difference between `new_position` and the current access port position (`rtmSetPointer[set_index]`).

- And read latency is computed as 5 + shift × 2 cycles, consistent with [2], and added to the cumulative counter `stats.readLatencyTotal`.
- The variable `stats.hitTotal` is used to accumulate the total number of shifts incurred on read hits, rather than just counting the number of hits.
- Finally, the `rtmSetPointer` is updated to reflect the new access position, simulating the movement of the access port in RTM hardware.

2. Modeling Shift and Latency During Writes

   A similar mechanism is implemented in the `allocateBlock()` function, which handles block replacement upon cache miss:

   - The system determines the index of the victim block and computes its shift distance relative to the current access port.
   - A write latency is computed as 8 + shift × 2 cycles, consistent with [2], and added to the cumulative counter `stats.writeLatencyTotal`.
   - The variable `stats.missTotal` is used to accumulate the total number of shifts incurred on write misses, rather than just counting the number of misses.
   - The `rtmSetPointer` is again updated to reflect the port's new position after the write.

3. Latency Model Adjustment

   To simulate the latency behavior of RTM more realistically, the `calculateAccessLatency()` function was customized for the `system.l2cache` component:

   - For read operations, an additional fixed latency of 5 cycles is added.
   - For write operations, the latency is set as `max(tag_latency, 8)` cycles, ensuring a minimum write cost of 8 cycles.

This latency model approximates the time required for domain wall alignment in RTM. The values used are based on the latency configuration reported in Hameed et al.'s 2023 SAR strategy paper, particularly as shown in their Figure 3.2 [2], which characterizes shift-induced delays.

Parameter	1 MB	4 MB
Leakage power (Tag array) [mW]	6.01	19.35
Leakage power (Data array) [mW]	20.57	64.72
Latency (Tag array) [ns/cycles]	0.42/2	1.03/4
Read latency (Data array) [ns/cycles]	1.27/5	1.79/6
Write latency (Data array) [ns/cycles]	2.29/8	2.97/10
Latency per single shift (Data array) [cycles]	2	2
Read energy (Data array) [pJ]	44.62	133.25
Write energy (Data array) [pJ]	64.57	217.53
Energy per single shift (Data array) [pJ]	43.8	121.61
Dynamic energy (Tag array) [pJ]	5.34	14.64

Figure 3.2. Comparison of latency and energy parameters for 1MB and 4MB RTM cache configurations. Taken from [2].

4. Port Position Updates on Block Insertion

   Finally, in the `BaseTags::insertBlock()` function, we added logic to synchronize the access port position with the location of any newly inserted block. By updating `rtmSetPointer[set]` to the `way` index of the inserted block, we ensure that subsequent shift computations remain correct, especially under frequent replacement scenarios.

These combined modifications allow gem5 to simulate RTM-specific shift behavior with high fidelity at the cache level, laying the groundwork for evaluating shift-aware replacement policies under realistic workload conditions.

3.2 Benchmarks and Processor Configuration

To evaluate the performance of various cache replacement policies under the Racetrack Memory (RTM) architecture, this study selects a subset of benchmarks from the **SPEC CPU2017** benchmark suite [7]. As one of the most widely adopted tools for microarchitectural evaluation in both academia and industry, SPEC CPU2017 provides realistic, representative, and computationally intensive workloads that are particularly suitable for analyzing memory hierarchy behavior [7].

Compared to its predecessor CPU2006, the CPU2017 suite reflects more modern computational demands, covering a wide range of applications such as compilers, graph search, AI inference, XML transformation, and scientific simulation. This diversity makes it ideal for examining how cache replacement policies perform under varying access patterns [7].

However, due to several practical constraints, we do not use all 20 benchmarks in our experiments. Instead, we select **6 integer programs** and **2 floating-point programs** to cover a range of memory access patterns and access intensities, based on the following considerations:

- Some programs failed to run under our gem5 syscall emulation setup due to configuration-related issues;
- Other programs have extremely high resource demands, making them difficult to simulate even with the test input set on a standard PC;
- To ensure that the experiments remain reproducible under constrained environments, we opted for benchmarks with moderate runtime and diverse behavior, and used the test input set for all programs in our evaluation.

The selected 8 integer benchmarks are:

- **600.perlbench_s** – Perl interpreter and syntax analysis
- **602.gcc_s** – C language compilation
- **605.mcf_s** – Network shortest path algorithm

- **621.wrf_s** – Weather forecasting model
- **623.xalancbmk_s** – XML transformation engine
- **631.deepsjeng_s** – Game tree search for chess
- **641.leela_s** – Go-playing AI
- **649.fotonik3d_s** – 3D photonic wave propagation

These benchmarks cover a variety of memory behaviors, including strong locality, frequent read/write operations, random accesses, and long compute phases. As such, they provide a solid foundation for analyzing the impact of different replacement policies on shift cost, latency, and energy implications in RTM-based cache systems [7].

Through this benchmark set, we are able to quantitatively assess shift counts, L2 miss rates, and access latency, offering a practical perspective on policy effectiveness and guiding the development of optimized RTM-aware strategies.

3.3 Computing Infrastructure for Simulation

All simulations and benchmark evaluations in this study were conducted on the high-performance computing node **bujaruelo.dacya.ucm.es**, which belongs to the **Department of Computer Architecture and Automation** (*Departamento de Arquitectura de Computadores y Automática*) at the **School of Computer Science, Universidad Complutense de Madrid**. The system configuration is as follows:

- **CPU**: 2 × Intel Xeon E5-2695 v3 (28 cores, 56 threads, up to 3.3 GHz)

  These server-grade processors provide excellent support for parallel workloads and are well-suited for compiling and running gem5 simulations.
- **RAM**: 64 GB

  The large memory capacity ensures stable execution of long-running simulations and allows for concurrent benchmarking without performance degradation.
- **GPU**: Although the system is equipped with multiple NVIDIA GTX 980 and GTX 1080 graphics cards, these were not used, as gem5 is a CPU-based architectural simulator.

Although the bujaruelo.dacya.ucm.es node was technically well-suited for running the simulations, it was a shared resource among many students. As a result, the machine's availability was limited—both in terms of access time and runtime allocation. This constraint ultimately prevented the execution of some benchmarks, for which no results could be obtained.

In summary, the hardware platform provided sufficient computational capacity and stability to support the majority of the experimental tasks conducted in this research.

To support reproducibility and comply with institutional guidelines, the source code developed and used in this project is available in the following public repositories:

- gem5 simulation and configuration base:
  https://github.com/angelesjiahuiyou/gem5-custom

- BRRIP-mod implementation (custom replacement strategy):
  https://github.com/angelesjiahuiyou/gem5_BrripNew

These repositories include simulation scripts, system configuration files, and implementation of cache replacement policies such as BRRIP, BRRIP-mod, and SAR. Instructions for building and executing the experiments are provided in the README of each repository.

Chapter 4 - Methodology and Results

This chapter presents a comprehensive evaluation of cache replacement policies under the Racetrack Memory (RTM) architecture, along with the design and analysis of a proposed improvement. Our analysis is divided into two parts: first, we evaluate several classical cache replacement strategies—including LRU, FIFO, MRU, and BRRIP—within an RTM model to understand their behavioral differences in a shift-cost-dominant environment. Second, we propose a new replacement strategy tailored to the unique physical characteristics of RTM, aiming to achieve a better trade-off between system performance and energy efficiency.

In Section 4.1, we perform a comparative analysis of the four classical policies using a unified gem5 simulation platform across several SPEC CPU2017 benchmarks. The evaluation focuses on three core performance metrics: total shift count, L2 miss rate, and access latency. Based on this analysis, we explain the motivation for selecting BRRIP as the foundation for further improvement.

In Section 4.2, we extend the BRRIP framework by introducing a shift-aware victim selection mechanism, and propose a new RTM-oriented replacement strategy named **BRRIP-mod**. We detail its design rationale, implementation logic, and expected benefits. This section also integrates the experimental results directly, comparing BRRIP-mod with the previously evaluated policies across all metrics to validate its performance advantages and suitability for RTM-based cache systems.

4.1 Analysis of Classical Replacement Policies

To systematically evaluate the adaptability and performance of several classical cache replacement policies under the Racetrack Memory (RTM) architecture, we selected eight representative SPEC CPU2017 benchmarks. These benchmarks cover a variety of workloads such as compilation, graph search, and image processing, and were chosen for their meaningful memory access behavior. Certain benchmarks were excluded due to extremely sparse memory activity or uncharacteristic access patterns, which made them less suitable for comparative analysis.

Using a unified simulation platform, we evaluated four classical replacement policies—LRU, FIFO, BRRIP, and MRU—and compared them across five key performance metrics. To ensure consistency and enable normalized analysis, all results are presented relative to LRU, which serves as the baseline.Additionally, both the arithmetic mean and geometric mean are presented, considering all evaluated benchmarks.

The following five dimensions were analyzed:

- **Shift Total**: The total number of domain wall shifts caused by data accesses in RTM; this is the core performance indicator in this study.
- **L2 Miss Rate**: Reflects the effectiveness of each policy in managing data locality and reuse.
- **L2 Read Latency**: Captures the impact of the replacement policy on the efficiency of read operations, especially in terms of shift cost.
- **L2 Write Latency**: Assesses the cost of write operations, which can be substantial in RTM due to domain movement.
- **L2 Total Latency**: A combined metric of read and write latencies, offering a holistic view of overall delay under each policy.

The results will be presented through a series of comparative bar charts, allowing for direct observation of trends across benchmarks. These visualizations are accompanied by detailed discussions that highlight the strengths, weaknesses, and behavioral patterns of each strategy in the RTM context.

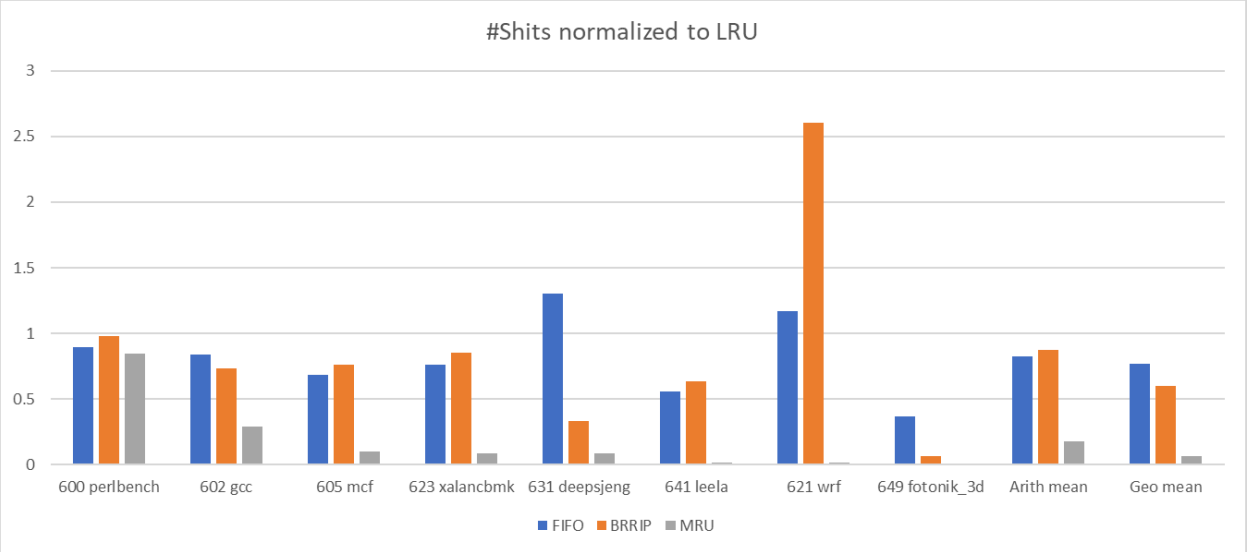

Figure 4.1: Total Shift Count of Replacement Policies, Normalized to LRU

Figure 4.1 compares the total number of shift operations produced by four classical replacement policies (FIFO, BRRIP, MRU, LRU baseline) across eight SPEC CPU2017 benchmarks. Values are normalized to LRU. BRRIP and MRU show shift cost reductions in a subset of benchmarks, particularly in memory-intensive workloads.

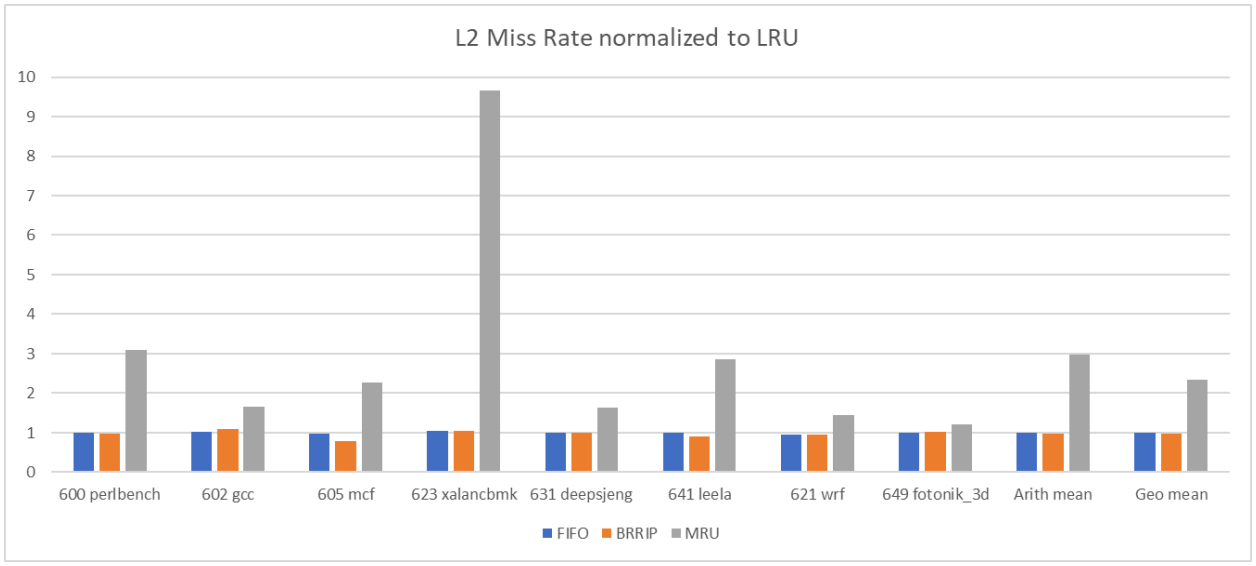

Figure 4.2: L2 Cache Miss Rate of Replacement Policies, Normalized to LRU

Figure 4.2 shows the L2 cache miss rate for FIFO, BRRIP, and MRU policies across eight SPEC CPU2017 benchmarks, normalized to LRU. While MRU achieves the lowest shift cost (see Figure 4.1), it suffers from a significantly higher miss rate, exceeding 9X of LRU in some cases (e.g., xalancbmk). In contrast, BRRIP maintains a low and stable miss rate across most benchmarks, making it a more balanced choice.

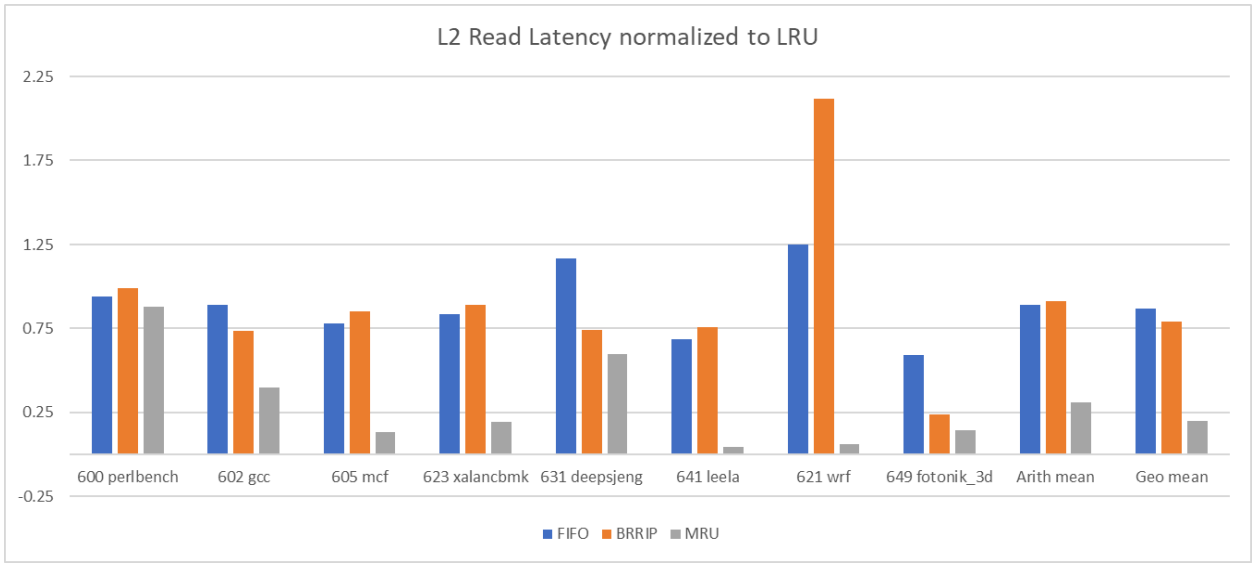

Figure 4.3: L2 Read Latency of Replacement Policies, Normalized to LRU

Figure 4.3 compares the normalized L2 read latency across different replacement policies. While MRU consistently achieves the lowest read latency due to its tendency to evict recently accessed blocks (close to the access port), it suffers from poor miss rate as seen in Figure 4.2. BRRIP performs comparably to FIFO in most benchmarks, but exhibits a spike in latency for `wrf`, highlighting the potential for optimization in shift-aware designs.

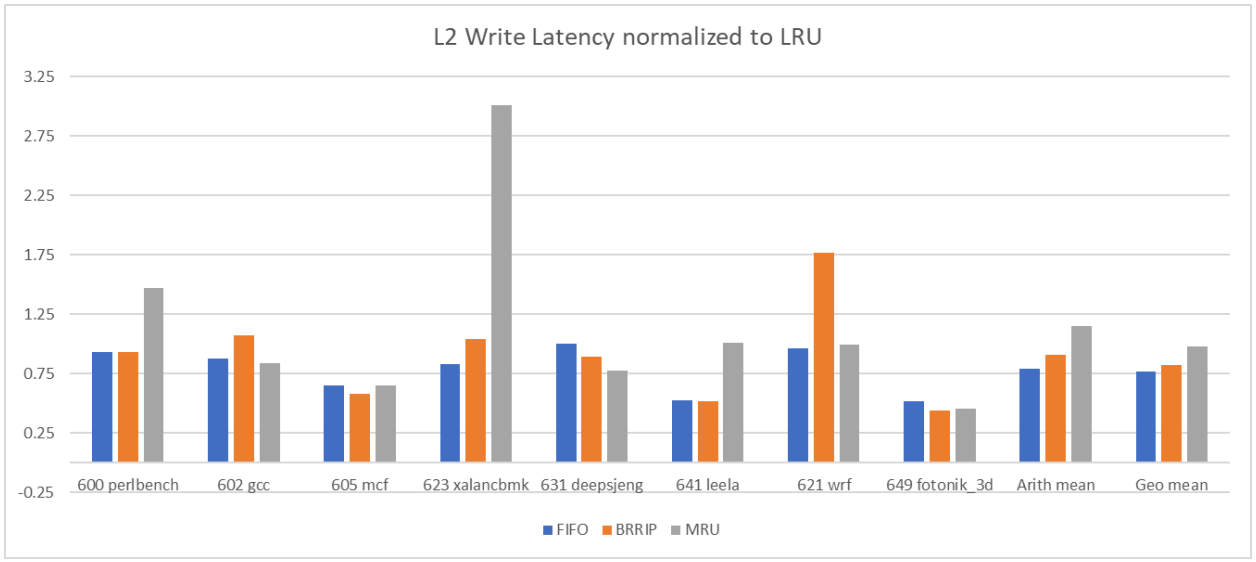

Figure 4.4: L2 Write Latency of Replacement Policies, Normalized to LRU

Figure 4.4 illustrates the normalized L2 write latency for FIFO, BRRIP, and MRU policies across several SPEC CPU2017 benchmarks. MRU shows the worst-case latency in benchmarks like xalancbmk, with values exceeding 3X that of LRU, due to frequent evictions of recently reused blocks. BRRIP maintains more stable latency, although in some cases (e.g., wrf), write costs increase due to shift-intensive replacements. These results highlight the importance of incorporating shift cost into replacement decisions for RTM.

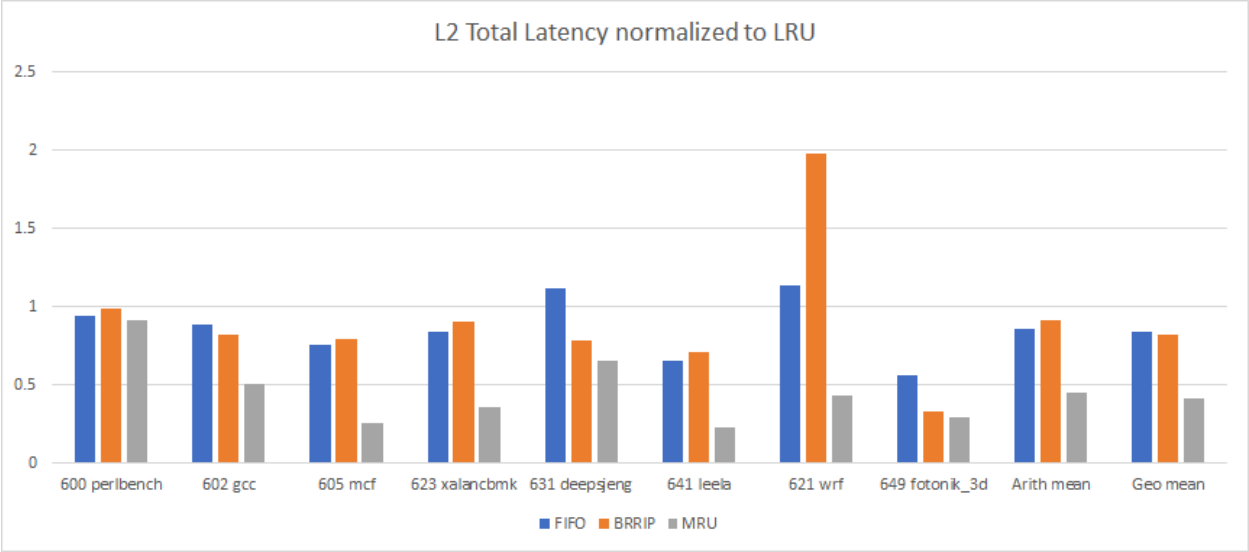

Figure 4.5: Total L2 Cache Latency of Replacement Policies, Normalized to LRU

Figure 4.5 presents the normalized total L2 cache latency (including both reads and writes) across all evaluated benchmarks. MRU consistently achieves the lowest total latency, although its performance is heavily skewed by high miss rates (see Figure 4.2). BRRIP shows stable latency and competitive performance, particularly in scenarios where it balances shift minimization and effective block retention. These results confirm that total latency alone cannot determine the best policy, and must be interpreted alongside miss rate and shift cost metrics.

IPC Analysis: Minimal Differences, Limited Usefulness

The IPC results across all four replacement policies showed very small differences, typically under 1% for most benchmarks. This limited variation can be attributed to the use of a simplified out-of-order CPU model and a fixed-latency memory system (gem5's SimpleMemory). These settings reduce the sensitivity of IPC to variations in cache behavior.

Additionally, in the evaluated memory hierarchy, the L2 cache directly interfaces with main memory (no L3), making memory latency and cache misses more relevant than IPC for performance evaluation.

As a result, this study focuses on memory-related metrics such as L2 miss rate, read/write latency, and shift cost, which are better suited for analyzing the efficiency and shift-awareness of replacement strategies in RTM-based cache designs.

Miss Rate and Shift Cost: More Informative Metrics

In contrast, L2 miss rate and total shift count provide significantly more insight into the behavior of replacement strategies under RTM in our setting:

- The miss rate directly reflects the strategy's ability to retain useful data;
- The shift count determines the actual energy and latency overhead unique to RTM architectures.

From our experimental results, the following patterns emerged:

LRU (Least Recently Used)

LRU typically yields moderate miss rates, but it tends to retain recently accessed blocks, which often reside far from the current access port. As a result, shift totals are relatively high, especially in benchmarks like `mcf` and `fotonik3d` with frequent cache accesses. While LRU is a reliable policy in traditional systems, it offers no energy-saving advantage in RTM environments.

FIFO (First-In First-Out)

FIFO performs slightly better than LRU in terms of shift count for certain benchmarks (e.g., *gcc*). However, due to its lack of locality awareness, it often evicts hot data too early, leading to increased miss rates and inefficiencies. Its performance

varies significantly across benchmarks, showing good results in some cases but poor adaptability in others, particularly when memory access patterns are not uniform or predictable.

MRU (Most Recently Used)

MRU yields the lowest replacement-related shift totals across almost all benchmarks. By design, it favors replacing the most recently accessed blocks, which are located at the access port, resulting in replacement operations with zero shift cost.

However, shift operations may still occur during regular read/write accesses (both hits and misses that do not trigger replacement), where the target block is not initially aligned with the access port.

While this strategy significantly reduces the shift cost associated with block evictions, it comes at the cost of extremely high miss rates—exceeding 90% in several benchmarks—and thus severely degrades overall system performance.

BRRIP (Bimodal RRIP)

BRRIP offers the most balanced and robust performance across all evaluated dimensions. By probabilistically inserting new blocks with RRPV = 3, it effectively balances data retention and eviction frequency, minimizing cache pollution.

In several benchmarks—such as *deepsjeng* and *xalancbmk*—BRRIP achieves lower shift totals and competitive miss rates compared to LRU and FIFO. Although in some cases (e.g., *gcc*) its miss rate is slightly higher, the overall trade-off with reduced shift cost makes it well-suited for shift-sensitive memory architectures like RTM.

Based on the above findings, BRRIP stands out as the most RTM-friendly replacement strategy, for the following reasons:

- Compared to LRU, it significantly reduces shift count and access cost;
- Compared to FIFO, it intelligently retains frequently reused data, lowering the miss rate;
- Compared to MRU, it avoids the dramatic performance penalties caused by excessive misses.

Therefore, BRRIP is selected as the foundation for our custom policy design, due to its ability to balance shift reduction, miss rate control, and system performance. In the next section, we propose an enhanced version of BRRIP, incorporating shift-awareness to better suit the physical characteristics of RTM.

Moreover, as shown in the geometric mean results across benchmarks, BRRIP consistently outperforms other classical policies in both total shift count and L2 miss rate, indicating strong generalizability and stability.

Its performance in latency-related metrics (Figures 4.3 to 4.5) is also notable: BRRIP achieves lower read, write, and total latency compared to LRU and FIFO, suggesting that reduced shift cost translates effectively into lower access delay.

In contrast, although MRU achieves the lowest shift cost, its miss rate is excessively high, especially in workloads such as `623.xalancbmk`, where it exceeds 9× that of LRU—rendering the cache nearly ineffective. This further reinforces the conclusion that BRRIP is the most well-rounded and RTM-compatible strategy among those evaluated.

Based on these findings, we proceed to design a new strategy in the following section: BRRIP-mod（Shift-Aware BRRIP），a shift-aware enhancement of BRRIP tailored to the RTM architecture.

4.2 Proposal of New RTM-Oriented Replacement Policies

As discussed earlier, the Bimodal Re-reference Interval Prediction (BRRIP) policy [5] is widely used due to its ability to reduce cache pollution and deliver stable performance in conventional memory hierarchies. In BRRIP, each cache block is assigned a *Re-reference Prediction Value (RRPV)*, typically ranging from 0 to 3. New blocks are inserted with RRPV = 3 (i.e., low likelihood of reuse), and only a small fraction are inserted with RRPV = 2. Upon a cache hit, the RRPV is decremented by one, and replacement occurs by selecting blocks with RRPV = 3. This conservative insertion scheme improves overall cache effectiveness by avoiding premature promotion of transient data.

However, in Racetrack Memory (RTM) architectures, BRRIP does not consider the physical shift distance between the access port and the target block—a key performance factor unique to RTM that directly impacts latency and energy. As a result, BRRIP may select replacement candidates that incur large shift costs, even if their reuse probability is correctly predicted.

To address this issue, we propose BRRIP-mod, an RTM-aware enhancement of BRRIP that incorporates shift cost awareness into the victim selection process. Our design preserves the original BRRIP insertion logic but modifies the replacement stage by prioritizing victims with lower shift distances when multiple blocks have the same RRPV value. This modification enables more energy-efficient and latency-aware replacements under RTM constraints.

The key components of the proposed method are as follows:

1. **Filtering for blocks with maximum RRPV values**

   In the `getVictim()` function, the policy first scans the replacement candidates

and selects those whose RRPV (Re-reference Prediction Value) equals the maximum value (typically $2^n - 1$; in our implementation, 3).

Additionally, cache lines marked as invalid (i.e., unused or with valid bit = 0) are also added to the candidate pool. These entries are prioritized when available, as using them avoids evicting valid blocks and helps utilize free space more efficiently.

2. **If no max-RRPV block is found, increment all RRPVs and retry**

   When no suitable victim is found with the maximum RRPV, the policy increments the RRPV of all candidates and recursively re-executes the selection. This simulates the aging mechanism used in RRIP-based policies.

3. **Among the max-RRPV candidates, select the one with the smallest shift cost**

   This is the core enhancement introduced in our strategy. Using a custom function `tags->calcRTMShift(candidate)`, we compute the shift distance between each candidate and the current access port position. The victim is then selected as the block with the lowest shift cost, minimizing latency and energy overhead.

   This RTM-aware BRRIP enhancement offers several advantages:

- Preserves BRRIP's low-pollution and stable behavior;
- Improves adaptability to position-sensitive access patterns in RTM;
- Reduces average shift count and access cost in high-miss and high-traffic workloads.

By introducing this lightweight yet effective modification, we provide a shift-aware intelligent replacement mechanism that aligns with the physical realities of RTM-based cache structures. In this section, we will evaluate the performance of this strategy using benchmarks from the SPEC CPU2017 suite.

To evaluate the practical effectiveness of **BRRIP-mod**, the proposed replacement strategy for RTM caches, we conducted a series of controlled experiments using the unified gem5 simulation platform. We selected seven representative benchmarks from the previously discussed SPEC CPU2017 subset, and compared BRRIP-mod against four classical replacement policies: FIFO, LRU, BRRIP, and MRU.

It is worth noting that while the original benchmark suite includes more test programs, one of them—*gcc*—was excluded due to its high computational demand. Under current hardware constraints, this program could not be completed within a reasonable timeframe, and the partial results obtained were insufficient for meaningful comparison. Therefore, we limited our evaluation to seven benchmarks with complete and consistent data.

The evaluation in this section uses the **same five performance metrics** introduced in Section 4.1, allowing a consistent comparison between classical and RTM-aware replacement policies.

The following figures present the normalized results for each metric (with LRU as the baseline), along with both arithmetic and geometric mean values to highlight overall performance trends. Through this analysis, we aim to determine whether BRRIP-mod effectively reduces shift costs while maintaining overall cache performance under RTM constraints.

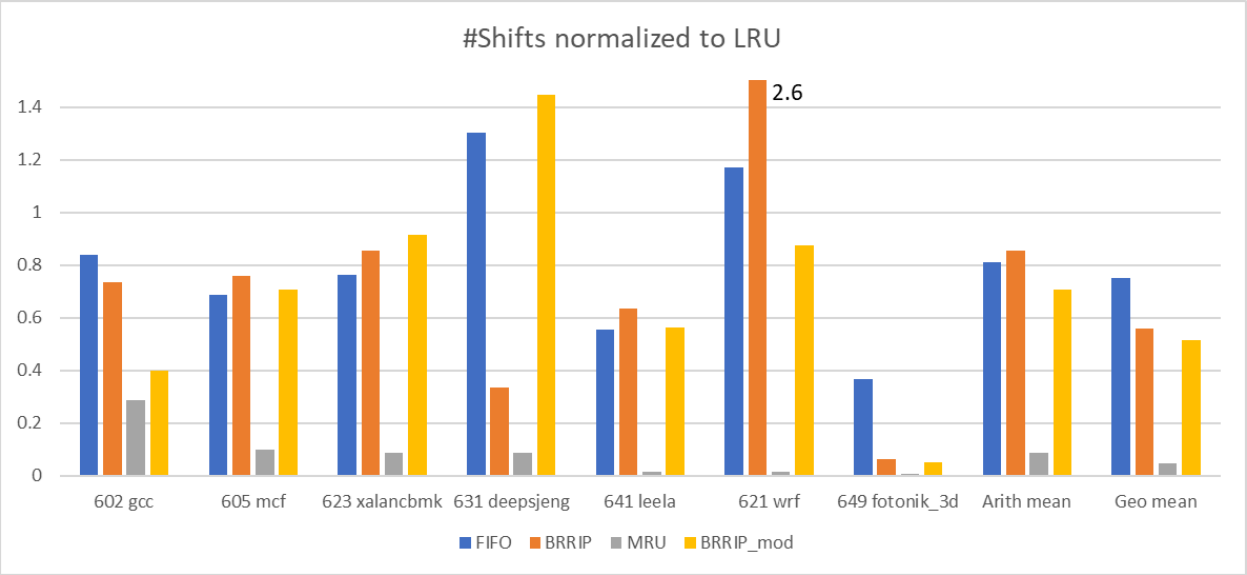

Figure 4.6: Total Shift Count of Replacement Policies (Normalized to LRU)

The figure 4.6 compares the total number of shift operations across seven SPEC CPU2017 benchmarks for four replacement policies—FIFO, BRRIP, MRU, and BRRIP-mod—normalized to the LRU baseline.

As shown in Figure 4.6, BRRIP-mod reduces shift counts compared to BRRIP and FIFO in most benchmarks. However, in some shift-sensitive programs—such as *deepsjeng* and *xalancbmk*—its shift count is higher. Despite these exceptions, BRRIP-mod achieves the lowest overall shift cost on average, as reflected in both arithmetic and geometric means.

In terms of geometric mean, BRRIP-mod reduces shift counts by 48.6% relative to LRU, compared to 44.1% with the original BRRIP. This represents an 8.1% improvement over BRRIP. While this reduction may appear modest, it is affected by specific cases such as *deepsjeng* and *xalancbmk*, where BRRIP-mod incurs more shifts than BRRIP. However, in other benchmarks like *gcc*, BRRIP-mod reduces shifts by over 30% more than BRRIP.

Moreover, compared to MRU—which achieves low shift counts at the cost of extreme instability and high miss rates—BRRIP-mod offers a more practical balance between shift efficiency and system performance, making it better suited for deployment in real RTM-based cache systems.

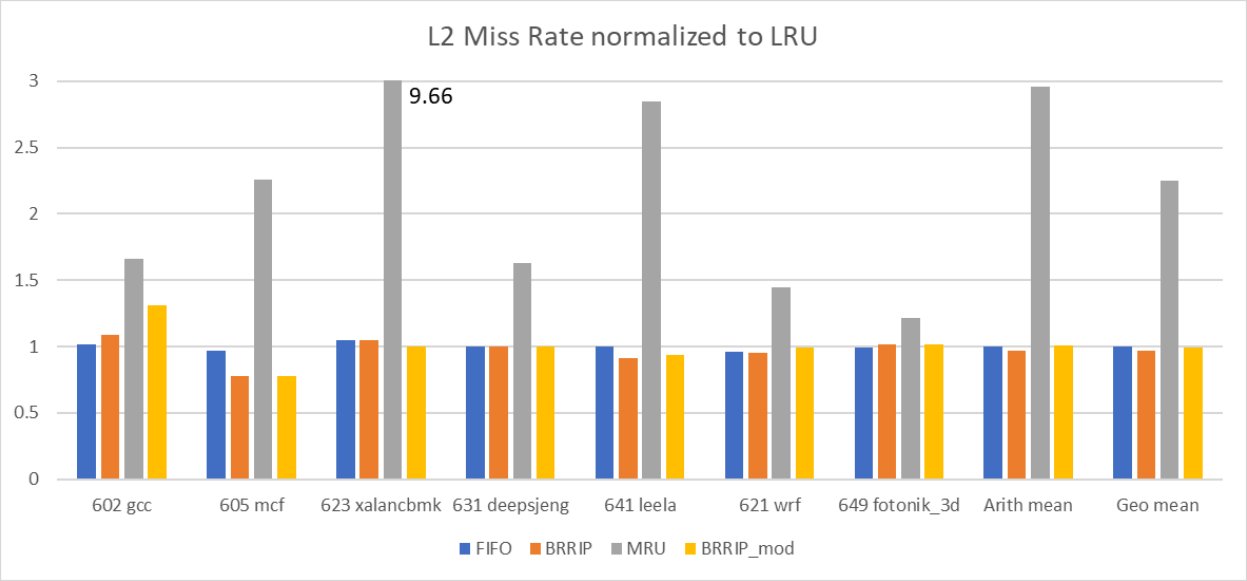

Figure 4.7: L2 Cache Miss Rate of Replacement Policies (Normalized to LRU)

The figure 4.7 presents the normalized L2 cache miss rates of FIFO, BRRIP, MRU, and BRRIP-mod across 7 SPEC CPU2017 benchmarks.

As shown, BRRIP-mod maintains a miss rate very close to BRRIP in most benchmarks, and even slightly lower in workloads like `mcf`, indicating that the shift-aware design does not compromise data locality.

According to the geometric mean, BRRIP-mod shows a slightly higher average miss rate than BRRIP (0.995 vs. 0.966). This result is consistent with the design of BRRIP-mod, which prioritizes minimizing shift costs over cache hit optimization.

In contrast, MRU suffers from extremely poor miss rates, with values reaching up to 9.66× that of LRU in `xalancbmk`, confirming that it is unsuitable for general-purpose systems despite its low shift cost.

Additionally, BRRIP slightly outperforms FIFO in terms of miss rate, further validating its effectiveness in retaining reusable data.

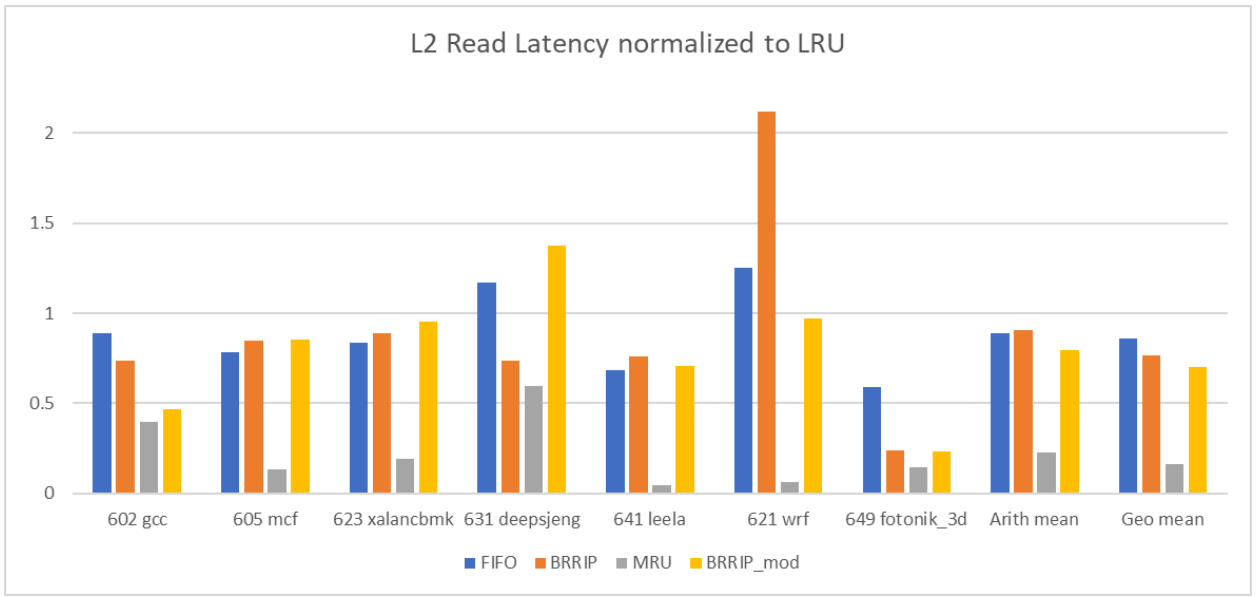

Figure 4.8: L2 Read Latency of Replacement Policies (Normalized to LRU)

The figure 4.8 shows the normalized L2 read latency of FIFO, BRRIP, MRU, and BRRIP-mod across 7 SPEC CPU2017 benchmarks.

BRRIP-mod achieves notably lower read latency than BRRIP and FIFO in several benchmarks. For instance, in *wrf*, BRRIP-mod reduces read latency by over 54% (from 2.12 to 0.97), and in *deepsjeng*, it cuts latency nearly in half. However, this improvement is not universal, and the overall gain depends on the workload's memory access patterns and shift dynamics.

Across all benchmarks, BRRIP-mod achieves an 8.66% reduction in geometric

mean read latency compared to BRRIP, demonstrating the effectiveness of its shift-aware logic in reducing access time along the read path.

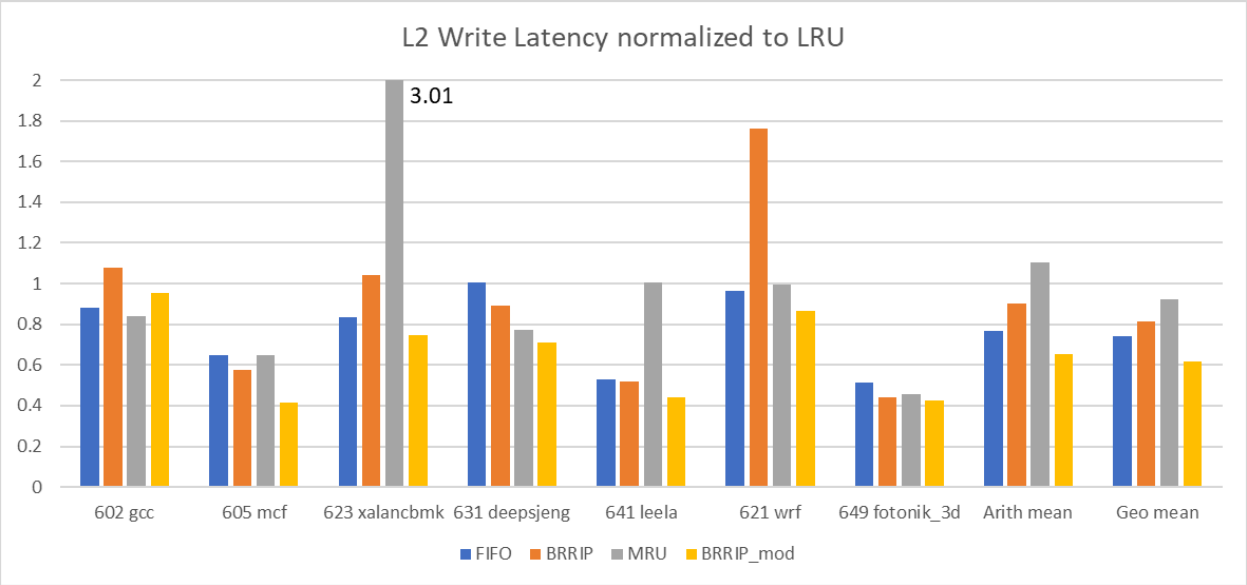

Figure 4.9: L2 Write Latency of Replacement Policies (Normalized to LRU)

The figure 4.9 presents the normalized L2 write latency of FIFO, BRRIP, MRU, and BRRIP-mod across 7 SPEC CPU2017 benchmarks.

BRRIP-mod achieves lower write latency than BRRIP in several benchmarks, particularly those with frequent shifts during write operations, such as *wrf*. However, this improvement is less noticeable in benchmarks with low write intensity, like *xalancbmk*.

In *wrf*, BRRIP's write latency reaches 1.76× that of LRU, while BRRIP-mod brings it down to 0.86×, representing a 51.1% reduction relative to BRRIP. Similarly, in *xalancbmk*, BRRIP-mod avoids the extreme latency observed in MRU (3.01×), maintaining a value of 0.75× relative to LRU.

On average, BRRIP-mod achieves a 24% reduction in geometric mean write latency compared to BRRIP, reinforcing its effectiveness not only for read access optimization but also for improving write behavior in RTM-based caches.

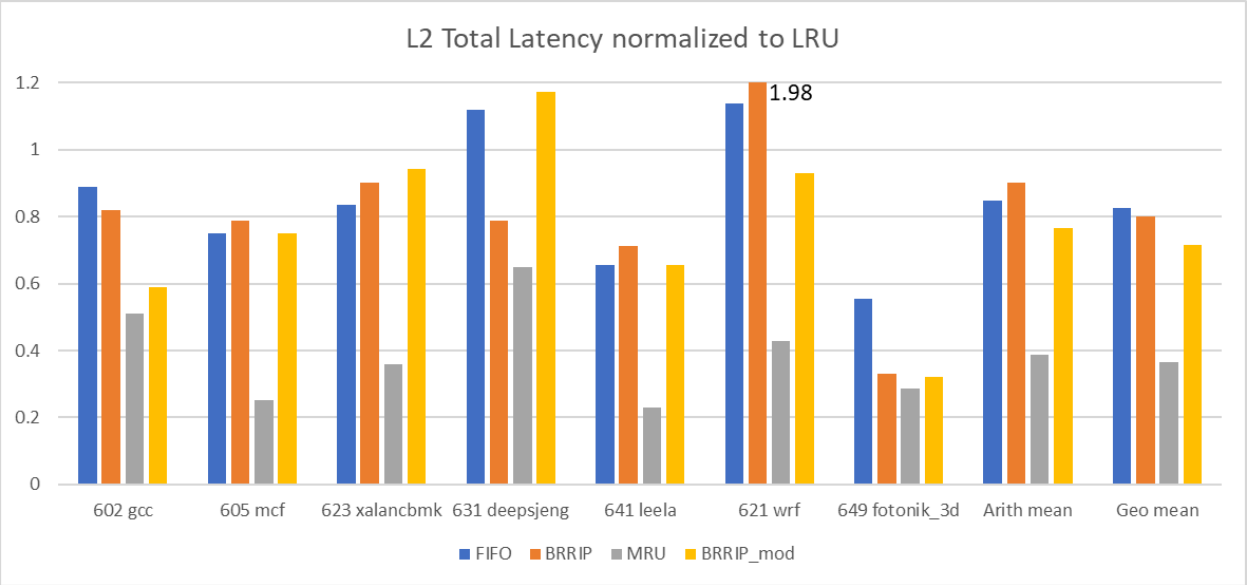

Figure 4.10: Total L2 Latency of Replacement Policies (Normalized to LRU)

The figure 4.10 compares the total L2 latency—including both read and write operations—for four replacement policies across 7 SPEC CPU2017 benchmarks, normalized to the LRU baseline.

Although MRU consistently yields the lowest total latency, BRRIP-mod achieves a better trade-off between latency and cache efficiency. In *wrf*, it reduces BRRIP's total latency from 1.98× to 0.97×, achieving a 53% reduction.

In terms of geometric mean, BRRIP-mod lowers total latency by 10.76% compared to BRRIP, confirming its ability to optimize both read and write paths simultaneously and enhance overall cache performance under RTM constraints.

Chapter 5 - Conclusions and Future Work

This study focused on the design and evaluation of cache replacement strategies in Racetrack Memory (RTM) architectures. We conducted a systematic evaluation of several classical policies, including LRU, FIFO, MRU, and BRRIP, and proposed an improved shift-aware strategy, BRRIP-mod, aiming to reduce shift costs, optimize access latency, and maintain high cache efficiency.

According to the experimental results, BRRIP-mod demonstrates clear advantages across multiple key metrics:

- Compared to BRRIP, it reduces the average shift count by approximately 8.1%;
- It slightly degrades the average L2 miss rate by less than 3%.
- In terms of latency, read latency is reduced by 8.66%, write latency by 24%, and total latency by 10.76%;
- It also maintains high stability across various benchmarks, outperforming all classical strategies overall.

In summary, BRRIP-mod effectively balances shift cost reduction, cache efficiency, and system performance, proving its practical potential for deployment in future RTM-based cache architectures.

Future Work

Despite the promising performance of BRRIP-mod, there remain several directions worth exploring:

- **Expanding strategy types**:

  This study focused solely on the BRRIP-based framework. In the future, other strategies supported by gem5 (e.g., DIP, DRRIP, SRRIP) could be evaluated to identify policy models better aligned with RTM's unique access characteristics.

- **Using a more advanced CPU model to analyze IPC**:

  Due to the use of a simplified single-core SimpleCPU model, the IPC metric showed little sensitivity to policy changes. Future work may adopt more realistic models, such as out-of-order (O3) or multicore architectures, to more accurately assess the impact of replacement strategies on throughput.

- **Incorporating energy modeling and evaluation**:

  Although this study does not directly measure energy, we can anticipate that reductions in L2 latency likely correspond to lower energy consumption, since shift operations in RTM affect both latency and power. In future work, energy models can be integrated into gem5 to enable dynamic, quantitative evaluation of energy-aware strategies.

With these extensions, future versions of shift-aware replacement policies may achieve even better performance, energy efficiency, and robustness, providing a more complete solution for RTM-based cache design.

BIBLIOGRAFÍA

[1] S. Parkin, "Racetrack memory: A storage class memory based on current controlled magnetic domain wall motion," 2009 Device Research Conference, University Park, PA, USA, 2009, pp. 3-6, doi: 10.1109/DRC.2009.5354890.

[2] Fazal Hameed, Moazam Maqsood, and Syed Ali Irtaza. 2023. An energy-efficient cache replacement policy for ultra-dense racetrack memory. J. Syst. Archit. 137, C (Apr 2023). https://doi.org/10.1016/j.sysarc.2023.102837

[3] Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.

[4] The gem5 Simulator, "Replacement Policies," *gem5 Documentation*, [Online]. Available: https://www.gem5.org/documentation/general_docs/memory_system/replacement_policies/. [Accessed: May 4, 2025].

[5] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, Saint-Malo, France, Jun. 2010, pp. 60–71. doi: 10.1145/1815961.1815971.

[6] The gem5 Simulator: Version 20.0+. Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria,

Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, Éder F. Zulian. ArXiv Preprint ArXiv:2007.03152, 2021.

[7] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU® 2017 Benchmark Suite – Overview," [Online]. Available: https://www.spec.org/cpu2017/Docs/overview.html. [Accessed: May 7, 2025].

[8] Adrián Colaso, Pablo Prieto, Pablo Abad Fidalgo, José-Ángel Gregorio, and Valentín Puente. "Architecting Racetrack Memory Preshift through Pattern-Based Prediction Mechanisms." In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 273–282. IEEE, 2019

[9] Fazal Hameed, Asif Ali Khan, Robin Bläsing, Stuart Parkin, and Jeronimo Castrillon. "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0." *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, Article 53, December 2019.

[10] Qingfeng Zhuge, Peng Hui, and Yao Chen. "Optimizing Data Layout for Racetrack Memory in Embedded Systems." In *Proceedings of the 2023 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 123–128. IEEE, 2023.

[11] Asif Ali Khan, Andrés Goens, Fazal Hameed, and Jeronimo Castrillon. "Generalized Data Placement Strategies for Racetrack Memories." In *Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1422–1427. IEEE, 2020.

[12] Haiyu Mao, Chao Zhang, Guangyu Sun, and Jiwu Shu. "Exploring Data Placement in Racetrack Memory Based Scratchpad Memory." In *Proceedings of the 2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pp. 1–6. IEEE, 2015.