UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE SISTEMAS ELETRÔNICOS

SERGIO ARRIBAS GARCÍA

MAXIMIZING CACHE HITS RATIO THROUGH PLANNED CACHE EVICTION POLICY SELECTION

DISSERTAÇÃO DE MESTRADO

Joinville
[2024]

Sergio Arribas García

MAXIMIZING CACHE HITS RATIO THROUGH PLANNED CACHE EVICTION POLICY SELECTION

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Engenharia de Sistemas Eletrônicos.
Supervisor: Dr. Giovani Gracioli

Joinville
[2024]

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Arribas, Sergio
MAXIMIZING CACHE HITS RATIO THROUGH PLANNED CACHE EVICTION POLICY SELECTION / Sergio Arribas ; orientador, Giovani Gracioli, 2024.
126 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Campus Joinville, Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos, Joinville, 2024.

Inclui referências.

1. Engenharia de Sistemas Eletrônicos. 2. Memórias caches. 3. Políticas de substituição de linhas da cache. 4. Otimização. I. Gracioli, Giovani. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos. III. Título.

Sergio Arribas Garcia

Título do trabalho: MAXIMIZING CACHE HITS RATIO THROUGH PLANNED CACHE EVICTION POLICY SELECTION

O presente trabalho em nível de Mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Dr. Rafael de Santiago
UFSC

Prof. Dr. Anderson Wedderhoff Spengler
UFSC

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em dezeseis de abril de dois mil e venite e quatro.

ICP-Edu
Documento assinado digitalmente
Lucas Weihmann
Data: 04/07/2024 15:26:05-0300
CPF: ***.676.080-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Lucas Weihmann
Coordenador do Programa

ICP-Edu
Documento assinado digitalmente
Giovani Gracioli
Data: 04/07/2024 14:44:36-0300
CPF: ***.838.060-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Dr. Giovani Gracioli
Orientador

Joinville, 3 de junho de 2024.

RESUMO

Os processadores modernos estão continuamente avançando, criando uma diferença de desempenho cada vez maior em relação às memórias disponíveis no mercado atual. As memórias cache desempenham um papel fundamental na redução dessa diferença, mitigando a latência de acesso à memória. No entanto, a eficácia dessas caches, que possuem tamanho limitado, depende fortemente da política de substituição de cache, impactando diretamente a taxa de hits na cache. As CPUs atuais empregam uma política de substituição fixa determinada no momento do projeto, que pode não ser a ideal durante toda a execução do programa, levando a um desempenho subótimo e à redução da escalabilidade do sistema. Para abordar essa limitação, este trabalho apresenta uma estrutura destinada a aumentar as taxas de hits, selecionando e alternando dinamicamente as políticas de substituição de cache em tempo de execução. Além disso, a estrutura inclui uma descrição abrangente de uma ferramenta de código aberto para profiling, juntamente com quatro abordagens baseadas no reconhecimento de padrões de acesso à memória, que são avaliadas. Os testes realizados com a estrutura demonstram possíveis melhorias na taxa de misses de cache de até 60% em comparação com a utilização da política de substituição LRU. Esses resultados impulsionaram melhorias adicionais na estrutura, incluindo a incorporação de políticas de substituição mais sofisticadas e a implementação de abordagens adicionais para alcançar resultados superiores

Palavras-chave: Memórias caches. Políticas de substituição de linhas da cache. Otimização.

ABSTRACT

Modern processors are continually advancing, creating a widening performance gap relative to current market-available memories. Cache memories play a pivotal role in bridging this gap by mitigating memory access latency. However, the effectiveness of these size-constrained caches heavily relies on the cache eviction policy, directly impacting the hit rate. Current CPUs employ a fixed replacement policy determined at design time, which may not be optimal throughout program execution, leading to suboptimal performance and reduced system schedulability. To address this limitation, this work introduces a framework aimed at enhancing hit ratios by dynamically selecting and switching cache eviction policies at runtime. Additionally, the framework includes a comprehensive description of an open-source tool for profiling, along with four approaches based on memory access pattern recognition, which are evaluated. Benchmarks conducted with the framework demonstrate potential cache miss rate improvements of up to 60% compared to using the LRU replacement policy. These findings have spurred further enhancements to the framework, including the incorporation of more sophisticated eviction policies and the implementation of additional approaches to achieve superior results.

Keywords: Cache memories. Eviction policies. Optimization.

RESUMO EXTENDIDO

Introdução: O texto discute a importância das memórias cache no desempenho dos sistemas de computação modernos, destacando a necessidade de políticas de substituição de cache eficazes devido à capacidade limitada das caches. Tradicionalmente, processadores usam uma única política de substituição de cache, mas pesquisas mostram que variar essas políticas pode melhorar o desempenho. A tese explora novas abordagens para otimizar essas políticas de forma dinâmica, adaptando-se a seções específicas de código para aumentar a eficiência do sistema. Durante a preparação deste trabalho, o autor contribuiu para várias publicações relevantes, abordando desde a minimização do uso de cache em sistemas em tempo real até o desenvolvimento de software embarcado para aquisição e calibração de dados em unidades de controle eletrônico automotivas, e a melhoria do tempo de execução de aplicações industriais através da seleção planejada de políticas de substituição de cache.

Objetivos: O principal objetivo do trabalho é identificar como o uso de diversas políticas de substituição em diferentes partes do código de um mesmo programa afeta a taxa de acertos na cache e, consequentemente, o tempo de execução. O autor divide a tarefa em objetivos específicos: propor quatro abordagens diferentes para simular código compilado e selecionar a melhor política de substituição de cache para seções específicas do código; implementar as quatro abordagens propostas como uma extensão de um framework de perfilamento de cache (Cachegrind) e disponibilizá-las como um artefato de código aberto para futuras pesquisas sobre tópicos relacionados; e avaliar e comparar as abordagens propostas em termos de falhas de cache e tempos de execução, com base na implementação do framework, usando aplicações de benchmark relevantes.

Metodologia: O documento revisa conceitos fundamentais sobre políticas de substituição de cache, investiga trabalhos anteriores e define as premissas e condições iniciais do estudo, além de descrever detalhadamente o modelo de sistema considerado. Apresenta quatro abordagens para otimizar políticas de substituição de cache, explicando sua implementação e algoritmos. Desenvolve experimentos para avaliar essas abordagens, executa simulações e coleta dados experimentais. Analisa os resultados comparando as abordagens propostas com políticas tradicionais, discutindo descobertas e tendências. Conclui com um resumo das conclusões baseadas nos resultados e propõe áreas de pesquisa futura e possíveis melhorias nas abordagens.

Resultados: Os testes realizados com a estrutura demonstram possíveis melhorias na taxa de misses de cache de até 60% em comparação com a utilização da política de substituição LRU. Esses resultados impulsionaram melhorias adicionais na estrutura, incluindo a incorporação de políticas de substituição mais sofisticadas e a implementação de abordagens adicionais para alcançar resultados superiores

Palavras-chave: Memórias caches. Políticas de substituição de linhas da cache. Otimização.

LIST OF FIGURES

Figure 1.1 – Fragment of cache hit ratio for `pca-small` benchmark considering four cache eviction policies (LRU, FIFO, RANDOM, and BIP). . . . 25

Figure 2.1 – Example of cache hierarchy with private L1 and L2 caches and shared L3 cache within a CPU architecture. 27
Figure 2.2 – Memory hierarchy. 29
Figure 2.3 – Cache associativity representation showing the blocks inspected during cache read. 31
Figure 2.4 – Cache metadata example for a 32-bit memory. 33
Figure 2.5 – Example of wrong selection of eviction policy using the LRU policy. 37
Figure 2.6 – Diagram for eviction policies types classification. 38
Figure 2.7 – Example of an application of traditional eviction policies (LRU, RANDOM and FIFO) in cache read. 39
Figure 2.8 – Example of use of BIP and LIP policies in a LRU friendly memory access sequence. 40
Figure 2.9 – Policies metadata representation using an encoding system based in arrays. 41
Figure 2.10 – Transformation of policies metadata encoded as array to a register. 42

Figure 3.1 – Set dueling mechanism for selection between LRU and BIP policies. 48

Figure 4.1 – Example of policy change in the naive approach. 53
Figure 4.2 – Overview of the naive search approach. 54
Figure 4.3 – Fixed-Window approach overview. 55
Figure 4.4 – Example of eviction policy transitions with the sliding-window approach. 57
Figure 4.5 – Overview of the online selection approach. 58
Figure 4.6 – Relation between probability of selecting best policy with Set Dueling and the dedicated number of sets. 60
Figure 4.7 – Overview of the proposed framework for cache eviction policy selection. 61
Figure 4.8 – Developer approach code implementation. 62
Figure 4.9 – Naive approach code section that selects the next eviction policy. . 63
Figure 4.10 – Fixed-window approach implementation overview. 64
Figure 4.11 – Fixed-widows approach pseudocode. 65
Figure 4.12 – Sliding-window approach implementation overview. 66
Figure 4.13 – Register usage for the aging implementation. 67
Figure 4.14 – Aging control pseudocode. 68
Figure 4.15 – Online approach implementation overview. 68

Figure 4.16 – Online approach pseudocode. 69
Figure 4.17 – Set dueling with 3 policies implementation overview. 70
Figure 4.18 – Set Dueling approach pseudocode. 71

Figure 5.1 – Miss rate evolution for several cache sizes using traditional policies with the liblinear-tsmall benchmark. 77
Figure 5.2 – Miss rate evolution for several cache sizes using traditional policies with the spc-small benchmark. 77
Figure 5.3 – Miss rate evolution for diverse cache sizes using traditional policies with sphinx-small benchmark. 78
Figure 5.4 – Miss rate evolution for diverse cache sizes using traditional policies with spc-small benchmark. 78
Figure 5.5 – Miss rate evolution for diverse cache sizes using traditional policies with lda-small benchmark. 79
Figure 5.6 – Miss rate evolution for diverse cache sizes using traditional policies with pca-small benchmark. 80
Figure 5.7 – Miss rate evolution for diverse cache sizes using naive approach with pca-small benchmark. 81
Figure 5.8 – Miss rate evolution for diverse cache sizes using 3 executions of naive approach with pca-small benchmark. 82
Figure 5.9 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with pca-small benchmark. . 83
Figure 5.10 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with rbm-small benchmark. . 83
Figure 5.11 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with spc-small benchmark (low range of windows). 84
Figure 5.12 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with spc-small benchmark. . 84
Figure 5.13 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with liblinear-tsmall benchmark. 85
Figure 5.14 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with liblinear-tlarge benchmark. 86
Figure 5.15 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with pca-large benchmark. . 86
Figure 5.16 – Miss rate evolution for diverse cache sizes and threshold configurations using Sliding Windows approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows. 98
Figure 5.17 – Miss rate evolution for several cache sizes and windows configurations using Sliding Windows approach with rbm-small benchmark. 99

Figure 5.18 – Miss rate evolution for several cache sizes and windows configurations using Sliding Windows approach with pca-small benchmark. 99
Figure 5.19 – Miss rate evolution for diverse cache sizes and windows configurations using Sliding Windows approach with srr-small benchmark. . 100
Figure 5.20 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows. 101
Figure 5.21 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows, focusing in small values of threshold. 102
Figure 5.22 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows, focusing in small values of threshold. 103
Figure 5.23 – Miss rate comparison for Set Dueling approach when using different memory line sizes (32 and 64 bytes). 104
Figure 5.24 – Miss rate evolution for diverse cache sizes configurations using Set Dueling approach with kmeans benchmarks with a diverse size of windows. 105
Figure 5.25 – Miss rate evolution for diverse cache sizes configurations using Set Dueling approach with lda benchmarks with a diverse size of windows. 106
Figure 5.26 – Miss rate evolution using Online with Cortex benchmarks (Cache size 32768 bytes). 107
Figure 5.27 – Miss rate evolution using Fix Window with Cortex benchmarks (Cache size 8192 bytes). 107

LIST OF TABLES

Table 3.1 – Related work overview. 50

Table 4.1 – Overview of the approaches features. 52

Table 5.1 – Parameters of the considered processors. 76
Table 5.2 – Number of sets depending on cache configuration. 90
Table 5.3 – Suboptimal percentage of memory, considering 32 sets per policy. . 90
Table 5.4 – Parameters used in every method for comparison with traditional policies 93

LIST OF ACRONYMS

BIP	Bimodal Insertion Policy	24
CPU	Central Processing Unit	27
SRAM	Static Random-Access Memory	28
DRAM	Dynamic Random-Access Memory	28
LLC	Last-Level Cache	30
NINE	Non-Inclusive Non-Exclusive	30
PRNG	Pseudo-Random-Number-Generator	38
LSFR	Left Shift Forward Register	38
LRU	Last Recently Used	38
MRU	Most Recently Used	39
BIP	Bimodal Insertion Policy	39
LIP	LRU Insertion Policy	39
mlp	Minimal life span	45
PVISAM	Predictable Variable Isolation in Shared Antipodal Memory	46
ARC	Adaptive Replacement Cache	46
DIP	Dynamic Insertion Policy	46
SDMs	Set Dueling Monitors	47
SRRIP	Static Re-Reference Interval Prediction	47
BRRIP	Bimodal RRIP	47
DRRIP	Dynamic Re-Reference Interval Policy	47
BTB	Branch target buffers	48
HITL	Human-In-The-Loop	52
DIP-SD	DIP Set Dueling	58

CONTENTS

1 INTRODUCTION 23
1.1 MOTIVATION 24
1.2 OBJECTIVES 24
1.2.1 Methodology 25
1.2.2 Outline 26
2 BACKGROUND 27
2.1 MEMORY HIERARCHY 28
2.2 CACHE MEMORIES 29
2.2.1 Locality Principles 29
2.2.2 Cache Hierarchy 30
2.2.3 Cache Management 30
2.2.3.1 Cache tags 31
2.2.4 Cache Performance 33
2.3 CACHE OPTIMIZATIONS 34
2.4 ACCESS PATTERNS 35
2.5 EVICTION POLICIES 36
2.5.1 Traditional policies 38
2.5.2 BIP 39
2.6 POLICIES METADATA 41
3 RELATED WORK 45
3.1 OPTIMAL APPROACH 45
3.2 LRU BASED APPROACHES 45
3.3 HYBRID POLICIES 46
3.4 AI APPROACHES 47
3.5 OFFLINE APPROACHES 48
3.6 POLICIES COMBINATION 49
3.7 OVERVIEW 49
4 SELECTION OF EVICTION POLICIES 51
4.1 ASSUMPTIONS AND SYSTEM MODEL 51
4.2 PROPOSED APPROACHES 52
4.2.1 Developer Approach 52
4.2.2 Naive search approach 53
4.2.3 Fixed-Window Approach 54
4.2.4 Sliding-Window Approach 56
4.2.5 Online Selection Method 57
4.2.6 Set Dueling 58
4.3 FRAMEWORK IMPLEMENTATION 60

4.3.1 Developer Approach **62**
4.3.2 Naive Approach **62**
4.3.3 Fixed-Window Approach **64**
4.3.4 Sliding-Window Approach **65**
4.3.5 Online Selection Approach **67**
4.3.6 Set Dueling Approach **69**
4.4 CONCLUSION 71
5 EVALUATION 73
5.1 EXPERIMENTAL SETUP 73
5.2 TRADITIONAL POLICIES 75
5.3 DEVELOPER AND NAIVE APPROACHES 79
5.4 FIXED WINDOW EVALUATION 81
5.5 SLIDING WINDOW EVALUATION 85
5.6 ONLINE SELECTION EVALUATION 87
5.7 SET DUELING EVALUATION 89
5.8 COMPARATIVE ANALYSIS 92
5.9 DISCUSSION 95
6 CONCLUSION 109
6.1 IMPLICATIONS AND RECOMMENDATIONS 109
6.2 LIMITATIONS AND FUTURE WORK 110
BIBLIOGRAPHY 111
APPENDIX A – COMPARISON OF MISS RATE FOR DIVERSE BENCHMARKS115

1 INTRODUCTION

Cache memories play a crucial role in enhancing the performance of modern computing systems by exploiting the temporal and spatial locality of memory accesses. As caches have limited capacity, the decision of which data to retain in the cache during a given time window becomes pivotal for system performance. This decision is governed by a cache eviction policy, responsible for selecting a cache line to replace upon encountering a cache miss (REINEKE et al., 2007). Commonly employed eviction policies in commercial processors include the Least Recently Used (LRU) and its variants, such as First in First Out (FIFO), and Random, each offering distinct performance profiles based on the memory access patterns of the target application and the cache size (REINEKE et al., 2007; TOUZEAU et al., 2019; SEGARRA; Gran Tejero; VIñALS, 2021).

Traditionally, processors have been designed with a single cache eviction policy per cache level, uniformly applied across all applications. However, several studies have explored the performance implications and potential benefits of varying the eviction policy (QURESHI et al., 2007; MANCUSO; YUN; PUAUT, 2019; ARAUJO et al., 2021). Yet, none of these studies have delved into the optimization of eviction policies tailored to specific code sections. This task is inherently challenging, requiring careful consideration of the sequence of memory accesses and the status of cache lines.

Furthermore, the ability to dynamically adjust the eviction policy at runtime, adapting the cache configuration as the process executes, holds the potential to enhance cache-hit rates and overall system performance. Such adaptability could enable the cache policy to evolve in tandem with the execution phases of a process.

The aim of this thesis is to explore novel approaches for optimizing cache eviction policies, with a focus on adapting policies to specific code sections dynamically. By leveraging runtime insights into memory access patterns and cache behavior, we seek to improve cache performance and overall system efficiency.

During the preparation of this document, the author has contributed to several publications in the field of real-time systems, embedded software development, and industrial electronics. These include "Minimizing Cache Usage for Real-time Systems" presented at the 31st International Conference on Real-Time Networks and Systems (RTNS ’23)(SUN et al., 2023), "Development of Embedded Software for Data Acquisition and Calibration of Automotive ECUs" presented at the 29th International Symposium of Automotive Engineering (SIMEA 2022)(ARRIBAS et al., 2022), "Improving the Execution Time of Industrial Applications through Planned Cache Eviction Policy Selection" presented at the 32nd International Symposium on Industrial Electronics (ISIE 2023)(GARCíA et al., 2023), and "Low-Cost Data Acquisition System for Automotive Electronic Control Units" published in Sensors journal in 2023(BEDRETCHUK et al.,

2023).

1.1 MOTIVATION

The cache-hit rate is an important metric to evaluate the efficiency of cache eviction policies. In fact, the ability to keep relevant data blocks close to the processor is key to preventing the need to retrieve data blocks from off-chip memory (i.e., *cache miss penalty*), as accessing the latter is orders of magnitude slower than accessing on-chip memory.

In the case of a cache miss, a cache line must be evicted to make space for the insertion of the newly requested memory data block. Due to the significant cache-miss penalty, selecting the line to be evicted is neither trivial nor without consequences for the eventual execution time. Evicting a cache line that will be reused shortly afterward will incur a performance penalty, as an extra cache miss penalty will have to be paid. On the other hand, evicting a cache line that will not be reused afterward is more acceptable.

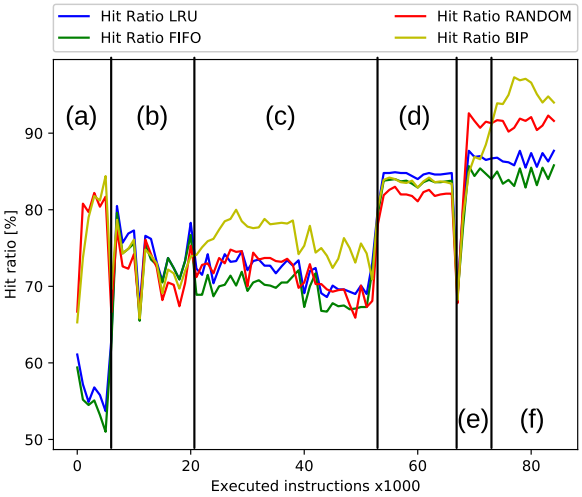
Using a fixed eviction policy during the execution of code is not a guarantee of the best performance. As illustrated by Figure 1.1, an application presents different performance in terms of cache hit ratio according to certain code sections and the used eviction policy. This figure presents the cache hit rate (y-axis) for four different eviction policies (LRU, FIFO, RANDOM, and BIP) and the number of executed instructions (x-axis) for the `pca-small` benchmark issued from the *Cortex Benchmark Suite* (THOMAS et al., 2014) running on top of Cachegrind (a cache profiler tool) (SEWARD; NETHERCOTE; WEIDENDORFER, 2008). Each point represents the percentage of cache hits in the last 1000 executed instructions. The benchmark's execution is divided into six segments (noted from a to f). Each segment is associated with the best-performing policy with respect to the hit rate. For instance, in segment (c), BIP is selected as it outperforms the other policies. This example highlights the potential achievable gains that can be obtained by enabling and exploiting eviction policy changes during process execution.

Moreover, choosing the best policy for specific code sections requires analyzing the cache status and the behavior of the hit ratio after the policy change. Thus, the four approaches to selecting cache eviction policies for code sections proposed in this work can help with that process.

1.2 OBJECTIVES

The main objective of this work is to identify how the use of diverse eviction policies in different parts of the code of the same program will affect the hit rate and, consequently, the execution time. We divided the task into specific objectives:

Figure 1.1 – Fragment of cache hit ratio for `pca-small` benchmark considering four cache eviction policies (LRU, FIFO, RANDOM, and BIP).



Source: Author (2023).

- To propose four different approaches to simulate compiled code and to select the best cache eviction policy for specific code sections. Three of the approaches are offline (executed through simulations), and one of them is online (based on cache-related information obtained at runtime);
- To implement the four proposed approaches as an extension of a cache profiler framework (Cachegrind (SEWARD; NETHERCOTE; WEIDENDORFER, 2008)) and release it as an open-source artifact for future research on related topics;
- To evaluate and compare the proposed approaches in terms of cache misses and execution times, based on the framework implementation, using relevant benchmark applications.

1.2.1 Methodology

- **Background Review:**
  - Exhaustive review of fundamental concepts related to cache eviction policies.
  - Investigation of previous works and analysis of existing literature in the field of study.
- **Framework Definition:**
  - Establishment of assumptions and initial conditions of the study.
  - Detailed description of the system model considered for the research.
- **Development of Proposed Approaches:**

  - Presentation of the four proposed approaches to optimize cache eviction policies.
  - Explanation of the implementation of each approach and the underlying algorithms used.

- **Experimental Evaluation:**
  - Design of experiments to evaluate the performance of the proposed approaches.
  - Execution of simulations and collection of experimental data.

- **Results Analysis:**
  - Interpretation of the experimental results obtained.
  - Comparison of the proposed approaches with traditional cache eviction policies.
  - Discussion of findings and trends observed in the results.

- **Conclusions and Future Directions:**
  - Summary of conclusions reached based on the experimental results.
  - Proposal of future research areas and potential enhancements to the proposed approaches.

1.2.2 Outline

The remainder of this document is structured as follows.

Chapter 2 provides a comprehensive review of background concepts and discusses related work in the field of cache eviction policies.

Chapter 4 outlines the assumptions made in this study, presents the system model considered, and introduces the four proposed approaches along with their implementation details.

Chapter 5 presents the evaluation of the proposed approaches, including experimental results and analysis.

Finally, Chapter 6 serves as the conclusion, where the findings of the thesis are summarized, and future research directions are proposed. This chapter discusses the implications of the results obtained and suggests potential avenues for further exploration in the field.

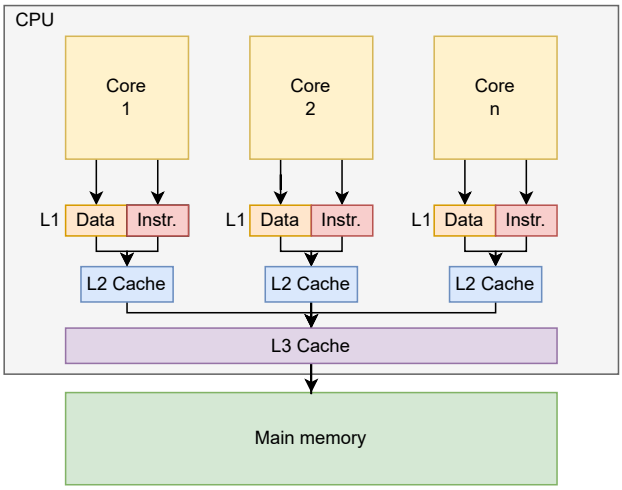
2 BACKGROUND

In computer architecture, the Central Processing Unit (CPU) employs cache memories to mitigate the latency associated with accessing slower memories. Cache memories serve as a buffer between the CPU and main memory (RAM), providing quick access to frequently accessed data and instructions. This design enhances CPU performance by reducing the number of cycles required to fetch information from main memory or secondary storage devices such as hard disks or solid-state drives (SSDs).

Figure 2.1 illustrates a typical cache hierarchy within a CPU architecture. At the top level, the CPU contains a small but fast cache known as the Level 1 (L1) cache, which is directly integrated into the CPU core. Below the L1 cache, there may be additional levels of cache, such as Level 2 (L2) and Level 3 (L3) caches, which are larger but slower than the L1 cache. These caches collectively form the cache hierarchy, with each level providing progressively larger storage capacity but with increasing access latency.

The purpose of this cache hierarchy is to exploit the principle of locality, which refers to the tendency of programs to access data and instructions that are spatially or temporally close to previously accessed items. By storing frequently accessed data and instructions in the faster cache levels closer to the CPU, the cache hierarchy aims to minimize the time spent waiting for data to be fetched from slower memory tiers, such as main memory or secondary storage.

Figure 2.1 – Example of cache hierarchy with private L1 and L2 caches and shared L3 cache within a CPU architecture.



Source: Author (2023).

Understanding the organization and operation of cache memories is crucial for optimizing CPU performance. Effective cache management techniques, including cache eviction policies, play a critical role in determining the hit rate of the cache, which measures the proportion of memory accesses that are satisfied by the cache

without requiring access to slower memory tiers. Maximizing the hit rate is essential for achieving high CPU performance and minimizing the overall execution time of programs.

This chapter begins by introducing the main concepts about memories and then proceeds to discuss cache memories, including their design principles and operational concepts. The chapter concludes by explaining how caches are accessed and reviewing cache replacement algorithms.

2.1 MEMORY HIERARCHY

Cache memories are an integral part of the typical memory hierarchy, which is designed to manage memory resources efficiently in computer systems. The memory hierarchy consists of multiple levels of storage, each optimized for specific tasks and selected during system design to balance factors such as performance, cost per bit, and power consumption (JACOB; WANG; NG, 2010).

At the top of the hierarchy are registers, which are small, extremely fast, and expensive memory units located directly within the processor core. Registers are used for storing data operands and intermediate results during computational tasks. They are controlled directly by the compiler and provide the fastest access times but have limited capacity due to their high cost per bit.

The next level in the hierarchy is the cache memory, typically implemented using Static Random-Access Memory (SRAM) technology. Cache memory is used to store frequently accessed instructions and data, providing low latency, high bandwidth, and relatively low energy consumption. It is integrated onto the processor chip and is organized into multiple levels (L1, L2, etc.) with varying capacities and access speeds.

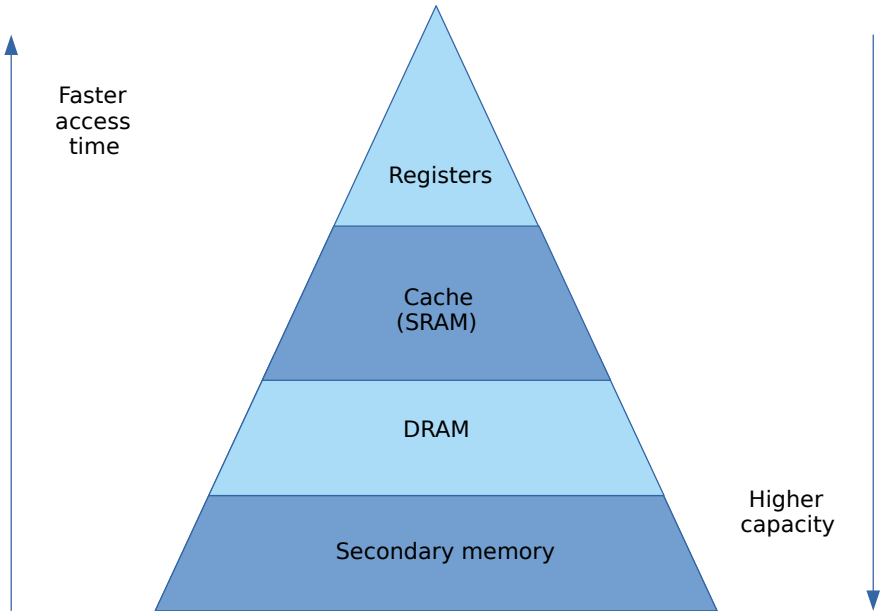
Below the cache memory is Dynamic Random-Access Memory (DRAM), which is larger and less expensive than cache memory but slower in terms of access times. DRAM serves as the main system memory, holding program instructions and data that are not currently stored in the cache. Unlike cache memory, DRAM requires periodic refreshing to maintain data integrity due to its internal structure.

Finally, secondary memory, such as hard disk drives (HDDs) or solid-state drives (SSDs), represents the bottom tier of the memory hierarchy. Secondary memory provides permanent storage for programs, data, and operating system files at a low cost per bit but with significantly slower access times compared to cache and DRAM.

The design of the memory hierarchy aims to exploit the principles of locality and temporal and spatial locality to optimize memory access times. By storing frequently accessed data and instructions in the faster and more expensive levels of the hierarchy (registers and cache), the system can minimize latency and improve overall performance. Less frequently accessed data is stored in slower and less expensive memory tiers (DRAM and secondary memory), allowing the system to make efficient use of available resources.

Figure 2.2 provides a graphical representation of the memory hierarchy, illustrating the different types of memory and their positions within the hierarchy. This visualization highlights the tiered structure of the hierarchy, with faster and more expensive memories positioned at the top and slower and less expensive memories positioned at the bottom.

Figure 2.2 – Memory hierarchy.



Source: Author (2023).

2.2 CACHE MEMORIES

In modern computer systems, the speed disparity between CPU processing speed and memory access latency poses a significant challenge to achieving optimal performance. Cache memories play a crucial role in bridging this gap by providing fast access to frequently used data and instructions, thereby reducing the overall latency of memory accesses (HENNESSY; PATTERSON, 2011). However, the design of cache memories involves trade-offs between latency, size, and cost. As cache memories are more expensive per unit of storage compared to main memory, there is a trade-off between the size of the cache and its latency-reducing effectiveness (AKANKSHA; CALVIN, 2019).

2.2.1 Locality Principles

Cache memories exploit the principle of locality of reference to improve memory access efficiency. This principle is based on the observation that memory references tend to exhibit locality, meaning that accesses to memory locations are often clustered together in time and space (JACOB; WANG; NG, 2010).

Temporal Locality

Temporal locality refers to the tendency of programs to reuse recently accessed memory locations in the near future. Cache memories exploit temporal locality by storing recently accessed data and instructions in the cache, allowing the CPU to access them quickly without having to fetch them from slower main memory repeatedly (PATTERSON; HENNESSY, 2017).

Spatial Locality

Spatial locality arises from the tendency of programs to access memory locations that are close to each other in memory space. Cache memories take advantage of spatial locality by fetching entire cache lines, which contain multiple adjacent memory locations, into the cache when a single memory location is accessed. This prefetching strategy reduces the latency of subsequent memory accesses to neighboring locations (PATTERSON; HENNESSY, 2017).

Algorithmic Locality

Algorithmic locality refers to patterns in memory access that are not strictly temporal or spatial but are characteristic of certain algorithms or program behaviors. For example, some programs may exhibit access patterns where data scattered across different memory regions are repeatedly accessed in a predictable manner. Prefetching techniques, either implemented manually or by the compiler, can exploit algorithmic locality to improve cache performance (HENNESSY; PATTERSON, 2011).

2.2.2 Cache Hierarchy

Cache memories in modern computer systems are organized into a hierarchy of multiple levels, typically including L1, L2, and L3 caches. Each core in a multicore processor usually has its own dedicated L1 cache, while the L2 and L3 caches are shared among multiple cores or a cluster of cores (PATTERSON; HENNESSY, 2017). The cache closest to the main memory, often referred to as the Last-Level Cache (LLC), serves as a shared cache for all processor cores and is crucial for maintaining coherence and reducing memory access latency (GRACIOLI et al., 2015).

The design of the cache hierarchy may incorporate different inclusion policies, such as inclusive, exclusive, or NINE policies. Inclusive caches contain all memory blocks present in lower-level caches, while exclusive caches only contain memory blocks that are not found in lower-level caches. The NINE policy allows different blocks in different cache levels after evictions, but new memory blocks fetched from DRAM or the LLC cache are placed in both levels (PATTERSON; HENNESSY, 2017).

2.2.3 Cache Management

Cache memories operate by organizing memory blocks into sets of fixed size, obtained from the logical partition of main memory. These memory blocks are stored in

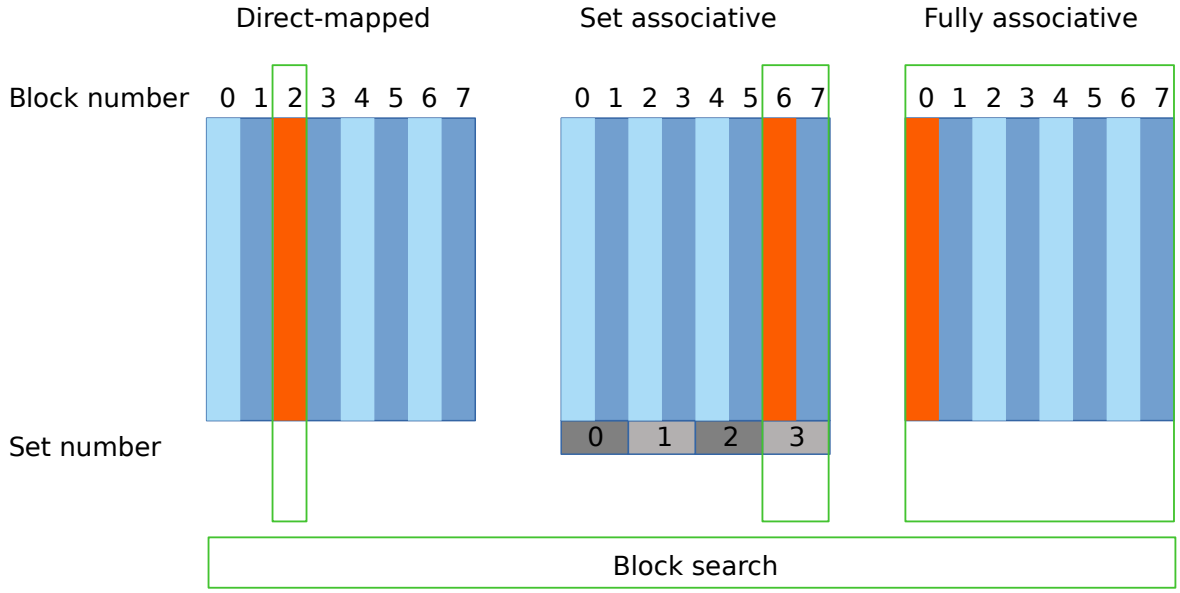
specific cache locations based on memory mapping strategies. Cache memory can be categorized into different types of associativity:

- **Direct-mapped**: Each memory block is assigned to a specific cache location, allowing for quick access but limiting flexibility.
- **Fully associative**: Any memory block can be stored in any cache location, providing maximum flexibility but requiring more complex search operations.
- **Set associative**: Cache memory is divided into equally-sized cache sets, with each set containing a certain number of lines. Within each set, memory blocks are mapped to specific lines, providing a balance between speed and flexibility.

The terms "block" and "line" are often used interchangeably when referring to cache memories.

Figure 2.3 illustrates the three types of cache associativity. In a direct-mapped cache, each memory block corresponds to a specific cache location. In a fully associative cache, any memory block can be stored in any cache location. In a set associative cache (shown as a 2-way set-associative example in the figure), memory blocks are mapped to specific lines within each set.

Figure 2.3 – Cache associativity representation showing the blocks inspected during cache read.



Source: Author (2023).

2.2.3.1 Cache tags

Cache memories are smaller than other types of memory in systems, which means that frequently accessed memory data needed by the CPU may not always

be present in the cache. Cache tags serve as a mechanism to identify the lines, and therefore the memory segments, stored in the cache. These tags, also known as cache metadata, are stored alongside the cache data and are generated using the memory address of the data.

Because accessing data quickly is crucial, devices need to know which locations in the cache memory they should check to confirm whether data is stored there or not. The position of the data in the cache memories depends on the cache associativity. In fully associative memories, all metadata of the blocks in the cache must be verified and compared with the searched memory address. However, this differs for other cache associativities. In direct-mapped caches, the block to be checked is determined by the operation shown in Equation 2.1, while in set associative caches, all blocks within the set must be checked to identify if the searched data is stored there. The set where a memory block could be stored is defined by Equation 2.2. When the processor needs to search multiple locations in the cache memory for memory data, the verification of the blocks is done in parallel.

$$\text{(Block address in memory) modulo (Number of blocks in cache memory)} \quad (2.1)$$

Equation 2.1 – Calculation of cache block number

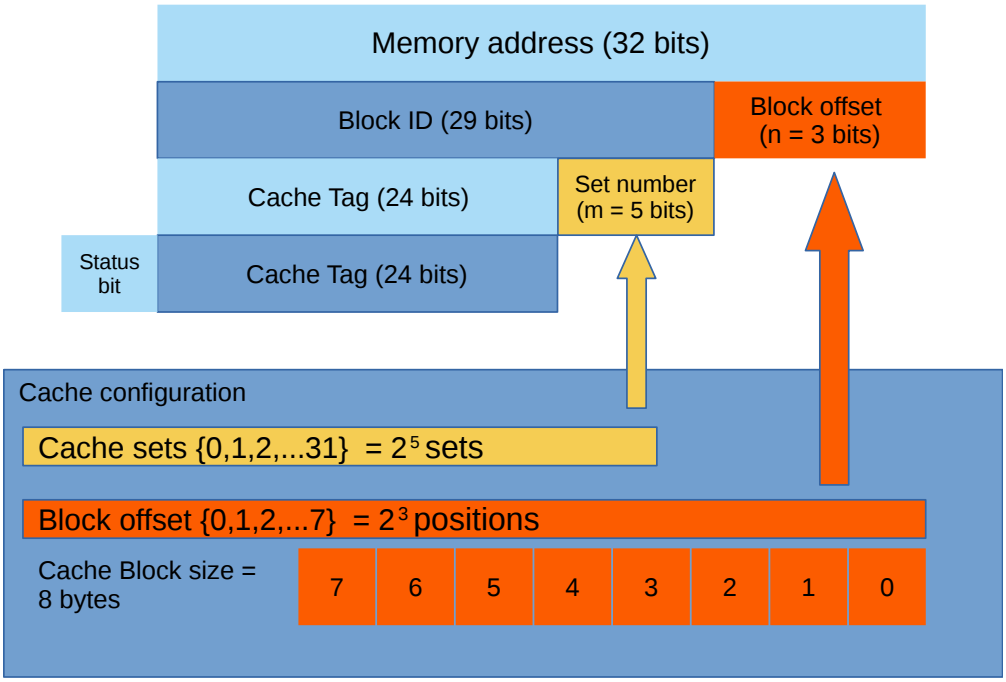
$$\text{(Block address in memory) modulo (Number of sets in cache memory)} \quad (2.2)$$

Equation 2.2 – Calculation of cache set number

Because of the limitation in cache memories space and aiming for optimization of hardware and power saving, metadata is stored together with the data as block identification instead of the full memory address. The metadata is formed from the tag, which is generated starting from the address of the memory block stored in the cache. For example, considering a 32-bit physical memory address and a block size of 8 bytes, the first 3 bits of the address represent the position of the byte inside the memory block (referred to as the block offset) (HENNESSY; PATTERSON, 2011). The size of the block defines the **n** bits used in the block offset. The next **m** bits after the **n** bits of the block offset are determined by the number of sets in the cache memory (considering 0 for a fully associative cache and the number of blocks of the cache for a direct-mapped cache), known as the *block ID* (JACOB; WANG; NG, 2010). For instance, if the cache memory has 32 sets, then the value of **m** is 5. The tag is obtained by removing the **n + m** lower bits of the address from the 32-bit physical address, resulting in 24 bits in the example, this is depicted in 2.4.

The tag is not the only part of the metadata in the cache memory. Normally, one bit is added to the tag to indicate the status of the cache position (valid or invalid).

Figure 2.4 – Cache metadata example for a 32-bit memory.



Source: Author (2023).

The fixed size of the cache blocks allows the use of cache tags in the metadata, reducing the data needed to determine if some data is stored in the cache memories, and it is also responsible for the number of bits needed for the block size. Similarly, increasing the associativity of the cache reduces the number of sets in the memory and the number of bits needed to indicate the set number, affecting the cache tag size.

When the processor must perform a write operation, it should indicate the byte of the block that must be modified, and then depending on the design of the cache:

- **Write through**: Both cache and RAM memories are written.
- **Write back**: Only the cache is written, and the cache block is marked to be written in the RAM memory when replaced.

Because the low-level memories do not provide the same bandwidth as the cache, some write buffers, faster than the memories, are used to avoid delays in *Write through* caches. The *Write back* design is more interesting for embedded systems, where power saving is a priority. In this type of design, an extra bit is used in the cache metadata, known as the *dirty bit*, which indicates whether the block must be written to the memory when replaced from the cache (HENNESSY; PATTERSON, 2011).

2.2.4 Cache Performance

When the CPU accesses a memory block, it checks if the block is in the expected cache position using the metadata, resulting in a *cache hit* when found or a *cache miss*

otherwise. During this comparison, the cache data is already loaded, so if the metadata matches the searched block, the data is readily available for use without the need for a new read operation.

Even though the processor can execute other instructions while waiting for the data not found in the cache memory, the performance of the memory system is evaluated based on the miss rate, miss penalty, and average access time. The *miss rate* is defined as the percentage of memory accesses resulting in cache misses, while the *hit rate* represents the opposite. Following a cache miss, the CPU experiences a stall for several cycles, and the *miss penalty* accounts for the total number of cycles required to store the memory block in the L1 cache and deliver it to the CPU.

The Average Access Time and CPU execution time are calculated as follows (KUMAR; SINGH, 2016):

$$AverageAccessTime = HT \times HR + MP \times MR \tag{2.3}$$

$$CPUExecutionTime = (CCC + MSC) \times CCT \tag{2.4}$$

$$MSC = Misses \times MP \tag{2.5}$$

$$= IC \times \frac{Misses}{Instructions} \times MP \tag{2.6}$$

$$= IC \times \frac{MemoryAccess}{Instructions} \times MR \times MP \tag{2.7}$$

Here, HT denotes the Hit Time, which is the time consumed when data is found in the Cache. HR represents the Hit Rate, indicating the percentage of cache hits. IC represents the number of instructions executed, MP stands for the Miss Penalty, representing the time consumed when a cache miss occurs. MR denotes the Miss Rate, indicating the percentage of cache misses. CCC refers to the CPU Clock Cycles, MSC denotes the Memory Stall Cycles, and CCT is the Clock Cycle Time.

On state-of-the-art architectures, a cache miss may incur several hundred CPU cycles, and it is anticipated that new architectures may exhibit even larger cache miss penalties (REINEKE; GRUND, 2013). Given that the system's performance is contingent on the cache performance, minimizing the number of cache misses is essential.

2.3 CACHE OPTIMIZATIONS

Several optimization mechanisms have been proposed to enhance cache performance:

a) **Increase cache size**: Expanding the cache's storage capacity allows for more memory blocks to be stored, thus reducing the miss rate. However, this option is often costly and not always feasible due to hardware constraints.

b) **Multilevel caches**: Introducing a second-level cache memory, typically larger and slower than the first level cache, enables the system to leverage the first level cache to minimize hit time and the second level cache to reduce miss penalty time. Although a miss in both levels results in a larger penalty compared to a single-level cache, overall performance improves.

c) **Prefetching**: Employing software or compiler techniques to predict which memory addresses will be needed in the cache before the processor accesses them, thereby reducing the miss ratio.

d) **Software blocking**: For codes utilizing data structures larger than the cache, such as large arrays, developers can optimize their code with algorithms that prevent loading the same data into the cache multiple times. These algorithms load part of the array, perform all necessary operations on that data, and then repeat the process with other parts of the array, rather than applying a single operation to the entire array before proceeding to the next operation.

e) **Increase cache associativity**: Enhancing cache associativity reduces conflicts after loading new data, as the new data can be stored in a set where the evicted data may no longer be needed. However, a fully associative cache model is suboptimal as it requires complex hardware and slows down the process due to the need to check all cache blocks after a cache miss.

f) **Increase cache block size**: Larger cache blocks result in fewer blocks being replaced in case of a cache miss. However, this may lead to cache misses when the program lacks spatial locality.

g) **Read priority**: Allowing the processor to execute read operations without waiting for a preceding write operation to complete can expedite memory accesses, especially when a write buffer is employed. However, this approach may generate conflicts when updated values of a memory address are stored in the buffer and a read operation of those values is requested.

2.4 ACCESS PATTERNS

As discussed in Section 2.2, the performance of a system is closely linked to the efficiency of the cache memory, which, in turn, depends on the number of cache misses. Ideally, a cache memory would have no misses, which could be achieved with either a large cache memory or the ability to predict future memory accesses requested by the processor before program execution. However, in real systems, predicting future memory accesses is not feasible. Nonetheless, studies (JALEEL et al., 2010) have identified common access patterns exhibited by memory accesses:

- Recency-friendly Access Patterns: In this pattern, a sequence of memory accesses is repeated *N* times. Within this sequence, accesses to blocks that were recently used are repeated in a short period of time.
- Thrashing Access Patterns: This pattern involves a sequence of memory accesses of length *k* being repeated *N* times, where accesses are not repeated in the same sequence.
- Streaming Access Patterns: Occurring when the length of the sequence of memory accesses is infinite.
- Mixed Access Patterns: This pattern combines sequences of memory accesses with block reuses in both the near and distant future.

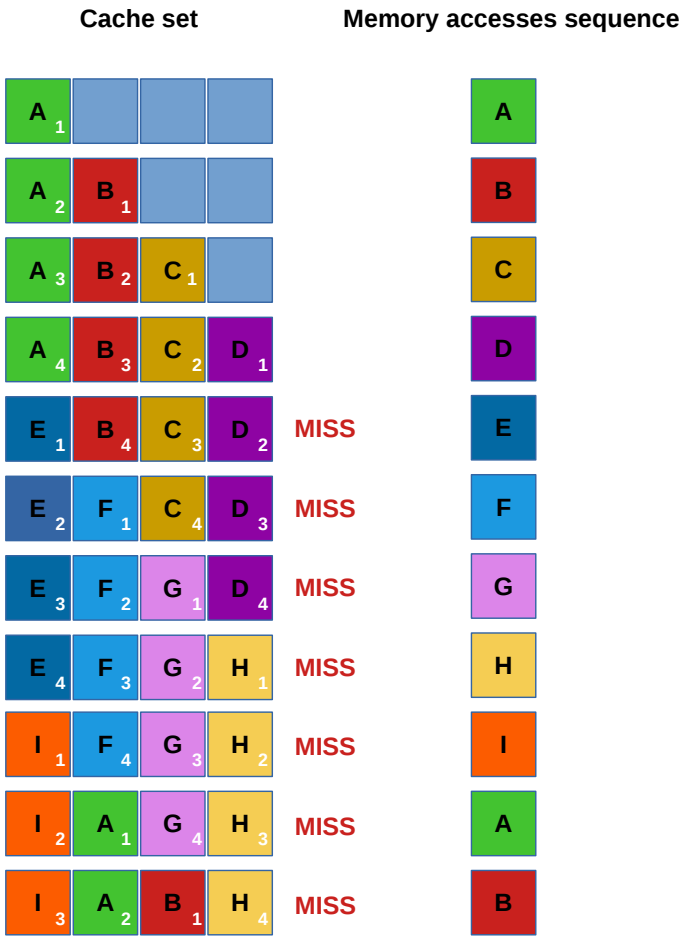
Identifying the access pattern allows for predicting which memory blocks will be used, enabling the cache to retain them and evicting other blocks less likely to be accessed. By selecting the cache line to be evicted based on the access pattern of the code, it is possible to reduce the miss ratio by maintaining memory blocks that are likely to be reused in the cache.

2.5 EVICTION POLICIES

When the processor requests data that is not present in the cache, this data is loaded into the cache memory. However, when the cache is full, it must replace old data previously stored. If the new blocks should be stored in an already full cache set, the hardware has to choose which block should be evicted and sent to a lower-level cache or removed from the cache. This process is not trivial because evicting blocks that will be reused in the near future will result in cache miss penalties, affecting performance.

An example of suboptimal selection of an eviction policy is illustrated in Figure 2.5. The figure shows a cache set composed of 4 memory blocks and the memory access sequence for that set. Each memory block in the set is assigned a number to indicate its position in the eviction queue, with 1 being the block that was most recently used and 4 indicating the next block to be evicted, as defined by the eviction policy applied. In this example, a traditional eviction policy is used, the **Least Recently Used** (LRU) policy. LRU stores memory blocks in the set in anticipation of their reuse in the near future, based on temporal locality. However, due to the size of the working set, the blocks are evicted before being reused, resulting in a low hit ratio, with all memory accesses after the set is full resulting in cache misses. The use of another policy for this access sequence would return a better hit ratio, reducing the number of misses. For example the use of the bimodal insertion policy (BIP) or LRU insertion policy (LIP), both explained in 2.5.2. We show a similar access sequence for the same cache configuration in the figure 2.8, where better results are obtained.

Figure 2.5 – Example of wrong selection of eviction policy using the LRU policy.



Source: Author (2023).

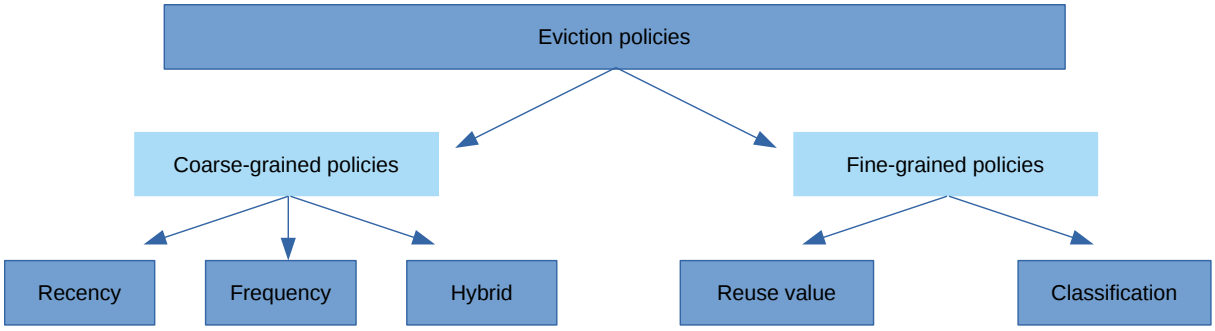
Various eviction policies could be employed in cache memories (AKANKSHA; CALVIN, 2019), and they can be classified based on the granularity of the insertion decision into:

- Coarse-Grained policies: All cache blocks are inserted in the cache identically, with the same priority or the same position in the eviction queue.

- Fine-Grained policies: New cache blocks can be inserted with different priorities.

Figure 2.6 illustrates the classification of eviction policies. Within coarse-grained policies, there are three groups: recency-based policies (such as LRU and its variants (REINEKE et al., 2007)), frequency-based policies (like LRFU (LEE et al., 1997)), and hybrid policies. The fine-grained policies are divided into two groups: classification-based policies and reuse distance-based policies.

Figure 2.6 – Diagram for eviction policies types classification.



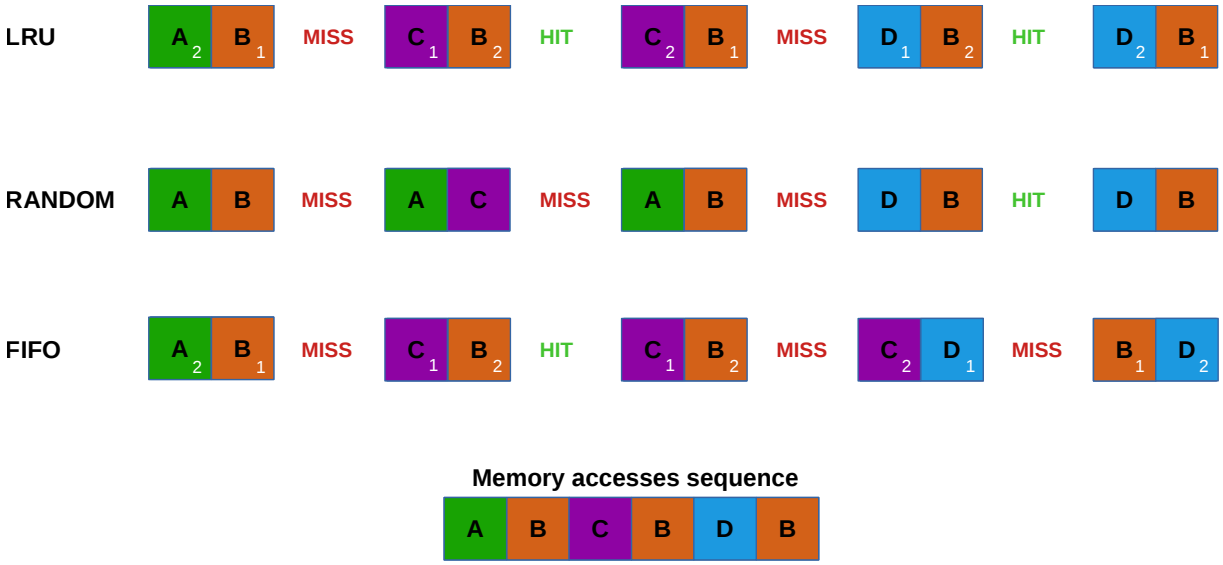
Source: Author (2023).

2.5.1 Traditional policies

In this section we review three commonly used eviction policies (REINEKE et al., 2007): *Least Recently Used* (*LRU*), *First In First Out* (*FIFO*), and *Random*. Both LRU and FIFO rely on intuitive heuristics based on the temporal locality of the data. The LRU policy and its pseudo-LRU counter-part keep track of which way has been accessed most recently. In the case of an eviction, the LRU policy will designate the least recently accessed cache line of the set as the victim. In contrast, the FIFO policy simply keeps track of how recently the cache lines in the set have been inserted and keeps that value unchanged when the cache line is accessed. Upon eviction, the policy will simply designate the cache line featuring the smallest value (i.e., the oldest line), replaces it with the highest possible value (i.e., $Ways - 1$), and decrement the other cache lines values in the set. Unlike the aforementioned policies, the Random eviction policy does not rely on any temporal locality assumption. Instead, the designation of the victim is dictated by a Pseudo-Random-Number-Generator (PRNG), often either a Fibonacci or Galois Left Shift Forward Register (LSFR).

In the example of the figure 2.7 the three traditional policies, *LRU, FIFO, and RANDOM* are applied to a memory accesses sequence. The figure shows a cache memory set composed by two blocks. For the *LRU and FIFO* policies a white number appears together with the blocks, this number represents the order of the block in the eviction queue of the set. The biggest number indicates the block that must be evicted when a cache miss occurs in the set. For instance, in the *LRU* policy when the processor requests the block with the letter **C** a miss is detected and the block **A** is evicted because it has the biggest number. The figure shows the transition between states of the cache set, between every transition we show the result of the request done by the processor, **miss** or **hit**. The first state of the set is the same for all the three policies, because we consider the cache was empty before the processor start requesting the data. For the *RANDOM* policy, the block evicted after every miss is selected randomly, it results in only one hit for this example but this may change in

Figure 2.7 – Example of an application of traditional eviction policies (LRU, RANDOM and FIFO) in cache read.



Source: Author (2023).

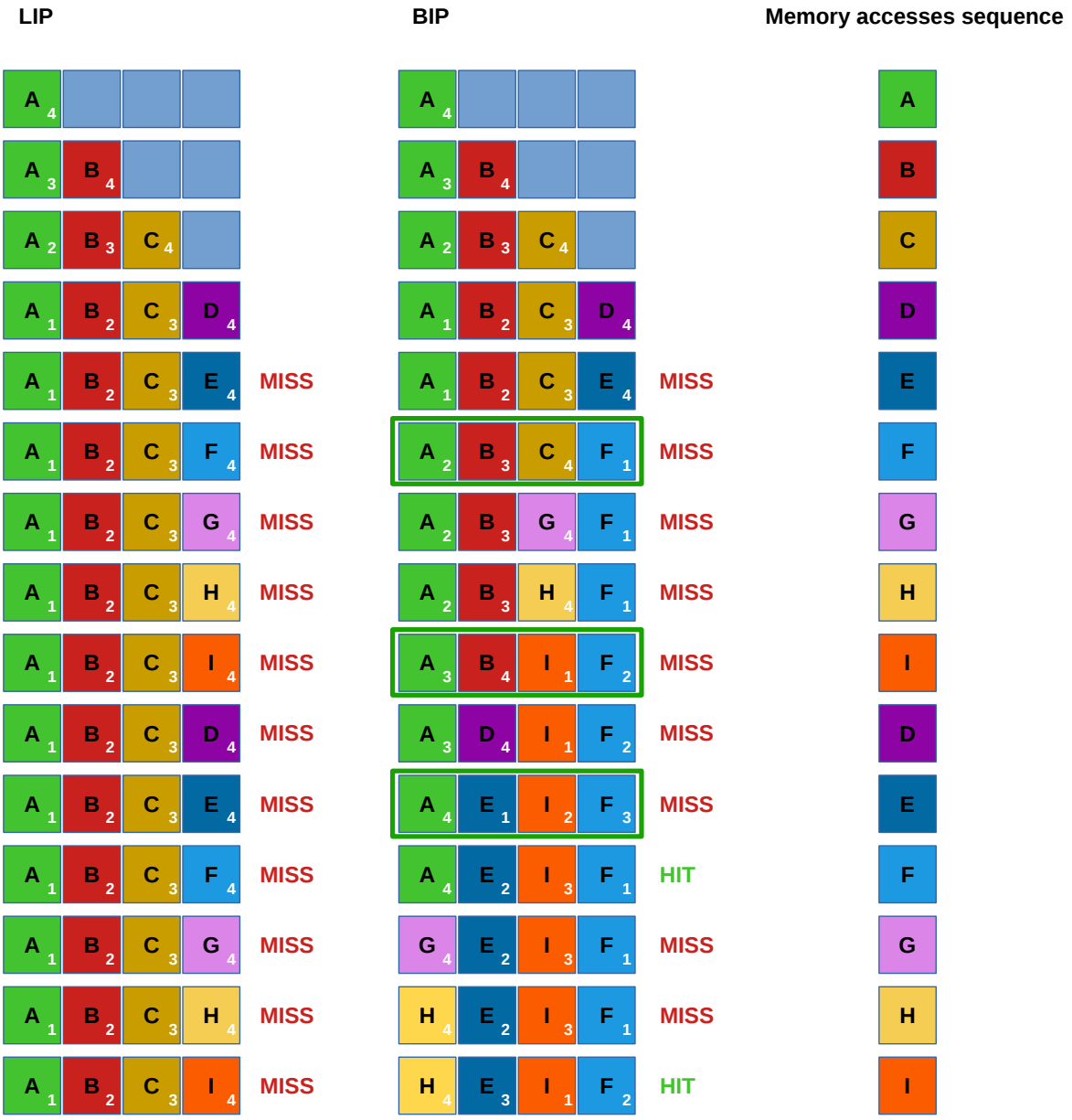
different executions of the same accesses sequence. For the *LRU and FIFO* policies, the difference between them is emphasized in the forth transition, where in *LRU* is a cache hit and in *FIFO* is a cache miss. This is a consequence of the second transition, since, even when both policies are cache hits, the *LRU* changes the order of the blocks in the eviction queue setting the block **B** as the in the MRU position and block **C** as the next block to be evicted. For that reason in the third transition the *FIFO* evicts the block **B**, which was inserted before block **C** but *LRU* evicts block **C** because **B** was the more recently used (MRU) block.

2.5.2 BIP

The bimodal insertion policy (BIP) is a coarse-grained eviction policy able to adapt itself to changes in the working set (QURESHI et al., 2007). The policy is an evolution of the LRU insertion policy (LIP), that inserts new memory blocks into the LRU position of the cache set, instead of into the MRU as the traditional LRU policy does. The LIP can address access patterns that are not LRU-friendly, as streaming or thrashing patterns. During working set changes between thrashing pattern and recency-friendly patterns the LIP has a bad behavior increasing the miss rate because is not able to store memory blocks in the cache that will be used in the short future. The BIP evolves from LIP inserting new blocks in the MRU position with a low probability. This low probability, which is a parameter call BIP throttle ϵ makes possible retain new values in the cache. Frequent values of ϵ are 1/32 or 1/64, when this parameter is 0 the BIP works as a LIP and with 1 as a LRU policy.

In the figure 2.8 we compared the behavior of **LIP** and **BIP** with a working set

Figure 2.8 – Example of use of BIP and LIP policies in a LRU friendly memory access sequence.



Source: Author (2023).

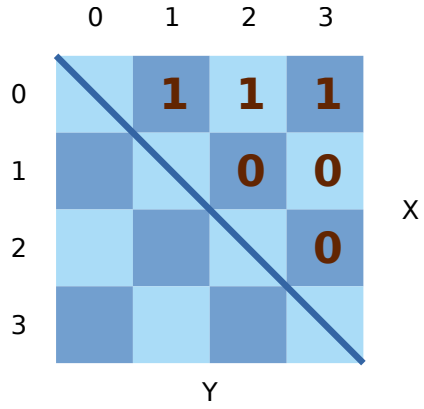
LRU-friendly bigger than the cache size. The cache memories are represented like a set of 4 blocks in both policies. The column named **"Memory accesses sequence"** enumerates the blocks requested by the processor. And the white numbers in the memory blocks of the sets represent the position of the memory blocks in the eviction queue, the bigger the number the closer to the LRU block that would be evicted in both cases after a cache miss. In the **BIP** policy the cache sets are surrounded by a green square when the block inserted on them is stored in the *MRU* position. In the example

the three first memory accesses are the end of a non-recency-friendly pattern and starting from the block **D** comes a recency-friendly sequence that repeats only twice, but it is possible to appreciate that if the sequence would continue repeating several times **LIP** would always result in cache misses whereas **BIP** gives a better hit ratio because store some of the blocks that are reused by the sequence.

2.6 POLICIES METADATA

The implementation of eviction policies for cache memories requires storing some data (metadata) related to the aging and eviction order of the blocks inside the cache sets, this information is used together with the logic of the policy to select which blocks must be replaced in case of a cache miss in the set. The way of encoding this metadata is crucial, it should be easy to use and reduced in size to avoid hardware overhead in cache. As example, we present an LRU metadata encoding representation, where the recency of every cache block of a set is stored. It is based in a **NxN** matrix where N is the amount of cache blocks in a cache set, where **x** is used to define the number of the rows and **y** the number of the columns. For instance, figure 2.9 shows the encoding representation for a set with 4 memory blocks.

Figure 2.9 – Policies metadata representation using an encoding system based in arrays.



Source: Author (2023).

That matrix uses a 1 in [x][y] when the block x of the set is more recent than the block number y. We consider that the matrix could be simplified, because [x][x] positions in the matrix compare the recency of the same memory block, and [x][y] and [y][x] represents the comparison of the same blocks. Therefore only the upper or lower triangle matrix must be used, resulting in a size defined by Equation 2.8.

$$Elements = ((NxN) - N)/2 \tag{2.8}$$

Upon a cache hit, wherein a block within the set is accessed by the processor, it mandates promotion. Under the purview of the LRU policy, this block is transitioned to the MRU (Most Recently Used) position. To illustrate, let's consider the block **m**. Its promotion is realized by updating all elements in row **m** to 1, while concurrently setting all elements in column **m** to 0.

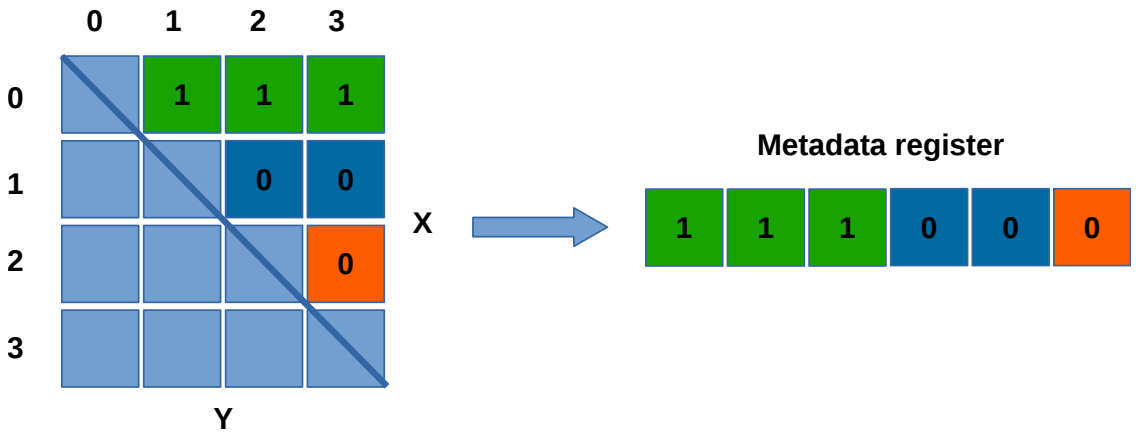
A scenario wherein every element in column **x** is 1 and every element in row **x** is 0 is indicative of the block x of the set being in the LRU position. Thus, upon a cache miss in that specific set, this block becomes the candidate for replacement.

Interestingly, this encoding paradigm extends its utility to MRU or FIFO policies as well. In the FIFO policy's context, a block's promotion is reserved exclusively for its insertion moment within the set, and it retains its position until eventual replacement, never advancing to the MRU spot. For ascertaining the MRU position, applicable to the BIP policy as well, it is imperative that all elements in the corresponding row bear the value 1, while all elements in the associated column are set to 0. Figure 2.9 show this, depicting block 0 of the set as the MRU block.

Transitioning to Figure 2.10, it's discernible that the block recency within sets can be encapsulated using a register. Every set within the cache necessitates a dedicated register, the bit-count of which is derived from equation 2.8.

However, the adoption of multiple eviction policies within a singular cache is not without its intricacies, particularly in relation to metadata. Transitioning between policies can lead to metadata that may not intuitively align with the underlying logic of all adopted policies. For example, when LRU is in action, metadata is reflective of the block accessed furthest back in time, given that blocks are elevated to MRU status upon utilization. This phenomenon is absent in the FIFO policy. Consequently, a switch from LRU to FIFO might result in a scenario where the eviction policy begins targeting more recently accessed blocks. This is due to the older blocks having been promoted during the LRU's use.

Figure 2.10 – Transformation of policies metadata encoded as array to a register.



Source: Author (2023).

In this chapter we went through the main concepts of memory hierarchy and cache memories. We discussed the bases of the cache memories and how they are controlled and operated. The space limitation of the cache memories forces the hardware designers and developers to search for improvements in this kind of memory. The frequent cache optimizations presented in this work are related to cache configuration or software methods to load the cache memories with data that would be used in a short future. We lately presented some of the more frequent memory access patterns that could be found in code and some of the traditional eviction policies used in cache memories, some of them designed for specific access patterns. Finally, we explained how this policies are managed by the processor to decide the data that must be evicted from the memory when new data must be inserted in the cache and the memory is already full.

3 RELATED WORK

The capability of the system caches to keep hold of the adequate data and provide a high cache hit rate depends on many architectural aspects such as the number of sets, the number of ways, the write policy, and the hit latency. But since the beginning of modern processor the eviction policies were in the focus of researchers who look for optimization of cache memories.

3.1 OPTIMAL APPROACH

The eviction policies aim to minimize the number of misses, this could be done retaining in the memory the blocks with highest probability of be reused in near future and doing the opposite with those with lowest probability (BELADY, 1966). The analysis of the sequence of address references of the program shows if a eviction policy has a good behavior. When a block must be evicted, the selection of dead blocks, those that never will be used, would be the result of good policy. Oppositely, the eviction of a block that will be used in the short future indicates that the policy is improvable. Because in runtime the next memory references are unknown, the optimal selection of the evicted blocks is not possible. The only way to reach the optimal eviction is running the software once, getting the memory reference sequence and using that information in a second execution to evict the blocks that never will be used or will be used later than the others when a cache miss occurs. Belady (1966) called this a two-pass job, and named the algorithm as **MIN**. This algorithm, even when impracticable for most of the applications, but according to the author could be used as base of comparison for other policies.

3.2 LRU BASED APPROACHES

As highlighted by Heckmann et al. (2003) and Reineke et al. (2007), amongst the state of the art eviction policies such as FIFO, and LRU, all of them provide the soundest resistance to "chaos", storing the aging of the data in a way that in case of lost metadata the policies may recover it after some new updates in cache content (HECKMANN et al., 2003). Unfortunately, in addition to requiring more local memory resources, the LRU policy suffers from one notable aspect: the minimal life span (abbreviated *mlp* in (REINEKE et al., 2007)). In other words, the minimal amount of time a cache line stays in the cache without being accessed. In the case of LRU, the *mlp* is a function of the number of ways. Qureshi et al. proposed three new cache line eviction policies derived from LRU (QURESHI et al., 2007). Referred to as *Adaptive Insertion Policies*, they differ by the way new cache lines are inserted in the cache set. The proposed LIP borrows all the precepts of LRU except that new caches line are inserted at the LRU position instead of the MRU position, addressing LRU's *mlp* shortcoming. Newly

inserted lines are promoted to the MRU position provided that they are accessed again after their insertion.Alternatively, Qureshi et al. (2007) proposed the BIP, an extension of LIP where the insertion of each cache line at the LRU position instead of the MRU is decided with a probability $\epsilon \in [0, 1]$.

In the work of Mancuso, Yun e Puaut (2019), the authors combine the Deterministic Memory (DM) concept with the LRU, creating an eviction policy named DM-LRU. The authors showed, through simulations, that the approach reduces tasks WCETs with less overhead than LRU and decache block-based approaches. For multicore systems with unified share caches, Haque et al. (HAQUE; EASWARAN, 2018) presented the Predictable Variable Isolation in Shared Antipodal Memory (PVISAM) eviction policy, which reserves fixed parts of the cache for each processing core and assigns the rest of the cache on demand among all remaining cores.

3.3 HYBRID POLICIES

Thy hybrid policies are based on the fact that some parts of the workload of a program would work better with a policy while other parts would do it with a different policy (JALEEL et al., 2010; QURESHI et al., 2007). Depending in the working set size a recency friendly or an anti-thrashing policy could perform better. The hybrid policies try to identify the changes in the working set and switch between different policies adapting to the changes. The challenge of hybrid policies is the identification of the changes in the working set without resulting in a high hardware cost. Diverse policies were researched in this sense, like the Adaptive Replacement Cache (ARC) , where the author create a policy that selects between recency-friendly and frequency-friendly policies(MEGIDDO; MODHA, 2003). The hybrid policy works with 2 tag directories, one for recency and another for the frequency, whose size is increasing or decreasing according to the detected cache hits. The final size of both tag directories is twice as big as the normal cache directory, but it only stores data for the same number of positions as the normal cache does, the rest of the directory tags save only metadata.

Qureshi et al. (2007) proposed a dynamically adapting policy called Dynamic Insertion Policy (DIP) that switches between the LRU and BIP policies, recency-friendly and anti-thrashing policies. The decision to switch is taken by monitoring cache hits trends, selecting the policy incurring the lower amount of misses. The selection is made dynamically online and the authors propose some alternatives to get this. **DIP-Global** implements two separated auxiliary tag directories for LRU and BIP and a counter that informs which policy has less misses selecting the best policy. The **DIP-DSS** (DIP Dynamic Set Sampling) reduces the hardware overhead of DIP-Global using two ATDs with 32 sets instead. Although the authors claim that all DIP variants are practicable (in terms of hardware implementation), only the **DIP-Set dueling** option does not imply in a big hardware overhead or extra ATDs. **DIP-Set dueling** method still combines LRU and

BIP policies as the other DIP solutions, but instead of using extra hardware or software it applies the set-dueling technique for the detection of the best policy for the current working set. DIP-Set dueling dedicates few sets to LRU and another few sets to BIP (same number for both), these are the set dueling monitors (SDMs) . The sets selection could be done randomly in runtime or fixed offline. Figure 3.1 depicts the set-dueling policy, showing 4 sets for each policy (LRU and BIP) and 6 following sets. The sets that are not part of the SDMs are called *follower sets*. The method uses a policy selector counter (PSEL) that is incremented when the sets dedicated to LRU have a cache hit and decreased when a hit is detected in the BIP SDMs. The counter is started to its middle value, and the most significative bit of the PSEL is used to define the policy used by the follower sets, when this bit is 1 LRU is used and when the MSB is 0 the BIP is used. The only extra storage needed by DIP-Set dueling is the PSEL. The adaptive insertion policies commented above were designed to work with recency-friendly and thrashing-friendly access patterns, but they do not work properly with other patterns, like the streaming access pattern, where data that will never be reused is accessed by the program.

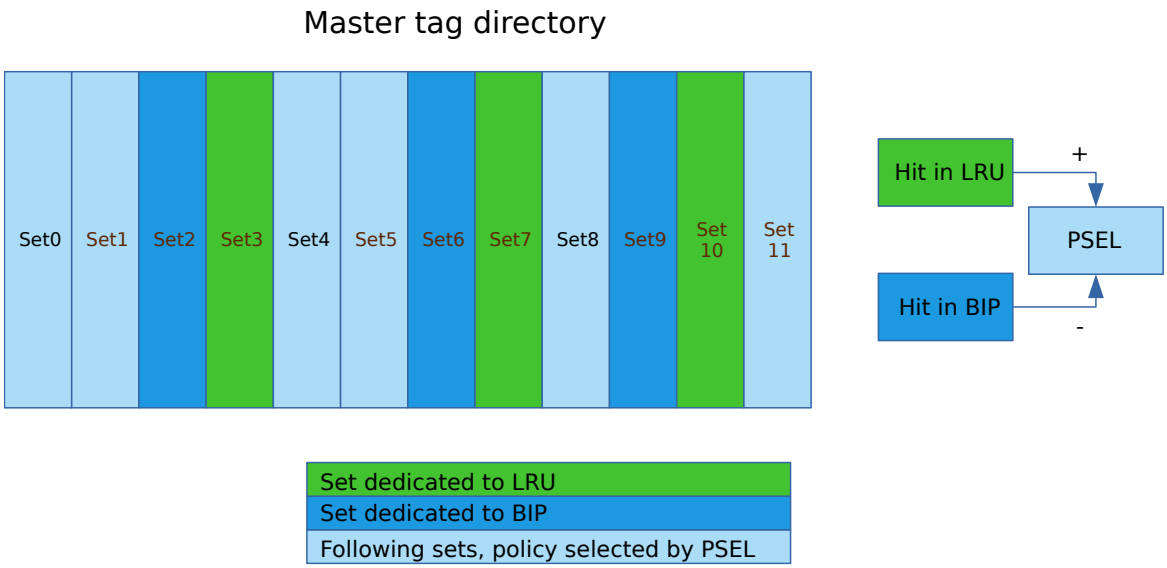
Jaleel et al. (2010) presented the dynamic re-reference interval policy (DRRIP), an approach based on set-dueling where the LRU and BIP policies are replaced with SRRIP and BRRIP , which are the streaming-resistant versions of LRU and BIP respectively. The streaming-resistant policies organize the block in a set regarding the prediction when a block is re-referenced. In the case of SRRIP the new blocks are in the middle of the prediction queue and promoted when referenced to the beginning of the queue. The queue is updated moving all the blocks to the end of the queue when a cache miss is detected and the latest block in the queue is evicted. The BRRIP works similarly to SRRIP, but as BIP has a throttle $\epsilon \in [0, 1]$ that defines the probability that a new block would be inserted in the middle of the queue, normally the blocks are inserted at the end of it and the aging and eviction follows the SRRIP logic.

Research focusing on the Real-time implications of *Adaptive Insertion Policies* (ARAUJO et al., 2021) have shown that DIP can improve the scheduling ratio by up to 50% in comparison to LRU using a fixed-priority non-preemptive scheduler.

3.4 AI APPROACHES

The difficulty to assess the usefulness of a cache line at a given instant has prompted researchers to rely on artificial intelligence (AI) techniques. These approaches train their classifiers using a sequence of eviction decisions generated by the optimal eviction policy (i.e., Belady's/Oracle's policy (BELADY, 1966)). Tools such as Hawkeye (JAIN; LIN, 2016) and Glider (SHI et al., 2019) train classifiers to guess whether a line is "cache-friendly" or "cache-averse" and base any eviction decision on this. These AI-based approaches, while successful at providing increased cache hit

Figure 3.1 – Set dueling mechanism for selection between LRU and BIP policies.



Source: Author (2023).

rates, are impractical to implement in real hardware as the network's depth drives the frequency down, and implementing extra memory, adders, and multipliers is expensive. Similar approaches were used for instruction caches as describes in (AJORPAZ et al., 2018), where prediction is shared for replacement policies used for caches and branch target buffers (BTB).

3.5 OFFLINE APPROACHES

The static analysis of the memory accesses could be used to reduce the miss ratio in the caches. The AI approaches are composed by a training phase where a model is trained to get a predictor that will be use in run time to classify memory block like cache-friendly or cache-averse, the eviction of the blocks perform by the policy is based on this classification. There are some other researches focused on the improvement of traditional eviction policies where a static compilation analysis is done before to run the software. The analysis returns some information about when a block is reused and allows the cache to improve eviction policies results (WANG et al., 2002). Wang et al. (2002) developed a theoretical model that uses the compiler to separate the memory blocks used in arrays of the software regarding their reuse level. They determine static locality patterns in the code after a full trace of code. The classification is then added to the cache LRU metadata of the blocks and use during a cache miss to evict the blocks with biggest reuse level together with a LRU policy that is apply in case of blocks having the same level. The hardware overhead generated by the 16 auxiliary bits the author propose to classify the reuse levels makes the proposal impractical. The author simplifies the solution reducing the 16 bits to a single bit called evict-me bit. This

bit is set following an algorithm that checks the data processed between re-reference of a block, number of references into the same cache set, and loop bounds of a nest in code. The cache then uses LRU together with the bit to select the block to be evicted. When the cache has a miss in a set first the blocks with the evict-me bit are candidates to be evicted following the LRU metadata, and in case of none of the blocks have the bit a standard LRU policy is applied to the set.

3.6 POLICIES COMBINATION

Researches have been done regarding the combined use of diverse eviction policies in the same cache memory. The use of multiple policies for the same memory was successfully implemented using set-dueling technique (QURESHI et al., 2007), the limitation of this approach was that only 2 policies could be used at the same time. In Srivatsa et al. (2020), the authors present a hybrid voting-based eviction policy for caches (HyVE). The policy is composed with multiple standalone eviction policies that work in parallel during the execution of the code, each policy uses it own metadata and tag directory. The cache chooses the memory block to be evicted based on a ranking system and using voting theory methodologies. The method has no learning subsystem as the AI methods and resolve to a consensus to select the eviction block. Even improving LRU results and almost reach AI approaches miss ratio, the proposed policy present a big hardware overhead needing between 6 and 8% more resources than LRU. This method was also applied to multicore architectures showing improvements and reduction of miss rates when compared with LRU or DRRIP.

3.7 OVERVIEW

Table 3.1 summarizes the different approaches designed to improve the results of eviction policies as LRU. The Belady approach (BELADY, 1966) is considered as the optimal solution in the literature as it results in the minimal number of misses but is not practicable, because it needs to use the future cache accesses to evict the lines in the cache memory. The Hawkeye (JAIN; LIN, 2016) and the Glider (SHI et al., 2019) approaches are both based on AI and have a considerable hardware overhead. They provide better results when compared with the DIP-based approaches.

Our mechanism, listed in the last row, instruments the programs obtaining the best combination of eviction policies for specific code sections. The offline versions return the lines of code where a policy change is triggered in the CPU maximizing the hit ratio of the code. This could be done using a special code instruction that triggers the eviction policy switch in the MMU. We propose also two online mechanisms that use three standard policies (LRU, FIFO and BIP). One of the online approaches uses

a selection strategy based in a set-dueling strategy, and the second one is based in counters and thresholds.

The difference of our offline approaches, when compared with the AI or Belady methods is the simplicity of the implementation in hardware. Our proposals are based in traditional policies that are frequently used in commercial CPUs and we switch between policy according to the section of code executed by the CPU while the other methods are complete policies, like LRU or BIP, rather than a policy selection method. The DIP selects also between policies and could be compared with our methods. The DIP-Global runs 2 policies in parallel and selects the policy with better hit ratio, we implemented one offline approach able to work with more than 2 policies that implements also aging strategy that improves the DIP-Global without hardware overhead and we also designed an online approach based in counters that improves DIP-Global allowing to have more than 2 policies. The DIP-Set Dueling is limited to the selection between two policies and does not consider the aging of the misses used to compare the performance of the policies. We implemented a set dueling based method that allows the use of more than 2 policies for the selection of the best policy and implements an aging technique to adapt itself faster to the changes of behavior of the code sections.

Table 3.1 – Related work overview.

Approach	**Practicable in HW**	**Overhead**	**Online/Offline**	**Technique/Mechanism**
DIP-Global	Yes	2 ATDs	Online	Parallel execution + Counter
DIP-DSS	Yes	2kB	Online	Parallel execution (some sets) + Counter
DIP-Set dueling	Yes	15 bits	Online	Some sets for each policy + Counter
Hawkeye	Yes	28kB	Online	AI, predictor based in previous accesses
Glider	Yes	62kB	Both	AI, predictor based in sequence of PC
Belady	No	-	-	Optimal, based in future accesses
HyVE	No	8% more than LRU	Online	Use of several policies and voting system
Evict-me	Yes	1 bit	Offline + Online	Check code defining memory to be evicted after miss
Proposed Work	Yes	15 bits +Multiplexer	Offline or Online	Detection of best policy for section of code

4 SELECTION OF EVICTION POLICIES

In this chapter, we start presenting the assumptions and system model we considered to allow the usability of the proposed approaches, then we present and discuss the implementation of the approaches, creating a framework for cache eviction policy selection in Sections 4.2 and 4.3, respectively. We close the chapter by discussing relevant aspects of the proposed framework in Section 5.9.

4.1 ASSUMPTIONS AND SYSTEM MODEL

We consider a System-on-Chip (SoC) design that can feature any arbitrary amount of cores. Each of these cores must be associated with private Level-1 (L1) data and instruction caches. The cache hierarchy can be extended to an arbitrary number of levels before leading to the DRAM and can either be private, shared, or unified.In addition, we do not consider non-blocking caches, the memory level parallelism they offer and the predictability issues they introduce (VALSAN; YUN; FARSHCHI, 2016). The main memory model does not take into account typical DRAM architecture and memory transaction arbitration under saturation. This assumption is not constraining as methods such as Memguard (YUN et al., 2013) are successful in preventing DRAM saturation on MPSoCs. These assumptions ease the modelization of the platform and, most importantly, help emphasize the benefits of the proposed approaches by removing any source of noise.

These approaches are powered by a custom cache architecture that poses the characteristics of being able to enforce different eviction policies following the software-layer/end-user directive. In addition, the proposed cache architecture is capable of providing a strict partition for every task of a given core, eliminating cache-related costs during context switches. While nothing in the presented model prevents the proposed cache architecture to be implemented at any level of the cache hierarchy we only assume the proposed cache to be implemented at the L1 cache. This decision helps greatly as each cache level takes independent decisions that influence subsequent levels.

Realistically, the considered cache architecture can be deployed at any level (L1, L2, LLC). It has the same configuration regardless of its level in the memory sub-system hierarchy: (1) setting up the eviction policy; (2) setting the cache partition size; and (3) indicating the currently running task on the core. The exact method may differ from one implementation to another, but in general, it can be implemented using memory-mapped registers. Alternatively, for more tightly integrated caches such as L1, the configuration can be done via dedicated assembly instructions.

Regarding the application model, it is up to the system engineering to define the best policy for each task or application, taking into consideration the criticality

levels of the system. For instance, if a time-sensitive task is assigned to a cache partition, then predictable and analyzable eviction policies, such as LRU, must be considered (REINEKE et al., 2007). If a task or application demands only performance, then any eviction policy could be applied. We focus on the optimization of the average execution time by selecting and changing the cache eviction policy during code execution.

4.2 PROPOSED APPROACHES

Identifying the right place to choose a new eviction policy in the code is not trivial, even using automated mechanisms. The optimal selection should consider the current status of the cache memories and the next memory addresses accessed by the software. We consider offline and online methods to select the eviction policy. Table 4.1 presents the main features of the proposed approaches. In the developer approach, the developer is involved in the process (Human-In-The-Loop - HITL) and relies on a trial-and-error process, requiring experience and effort from the developer to obtain a combination of policy transitions with good results. In the other approaches, which are automated, the defined parameters determine the outcomes. For instance, the size of the time window used to check the cache status after and before each cache access has a notable impact. If the window is wider than the section of the code where a policy returns a better hit ratio, the approaches may not be able to find the right place to switch to that policy. On the other hand, if the window is too short, the approaches may request constant policy transitions.

Table 4.1 – Overview of the approaches features.

Approach	**Execution**	**Policy selection**	**Extra space**	**Parameters**
Developer	Offline	HITL	No	-
Naive	Offline	Automated	No	-
Fixed-Window	Offline	Automated	Yes	Window size
Sliding-Window	Offline	Automated	Yes	Window size
Online	Online	Automated	Yes	Counter and threshold
Set Dueling	Online	Automated	Yes	Counters

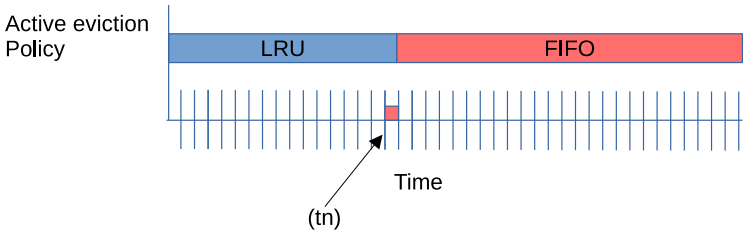
4.2.1 Developer Approach

In the developer approach, the developer is responsible for deciding when a cache eviction policy change must be performed by inserting a special instruction call into the code. This special call informs the memory controller that a new eviction policy should be used from that moment on. The chosen policy remains active until a new call selects another policy. For instance, in Figure 4.1, (tn) indicates the moment the instruction call changes the current policy (LRU to FIFO). The approach shares some

similarities with cache prefetching, because the developer could indicate when should be used, and it could be used in combination with it to improve the results. For instance, in a recency-friendly part of the code the developer may load the cache with the data that will be used and change the policy to LRU, this will retain the data in the cache and will remove data that was previously used but is not needed any longer.

The selection of a policy using an instruction that will be ran in the processor, is the same in the following approaches. The output of the offline approaches is a instruction in the program that will be inserted in the code according to the results obtained.

Figure 4.1 – Example of policy change in the naive approach.



Source: Author (2023).

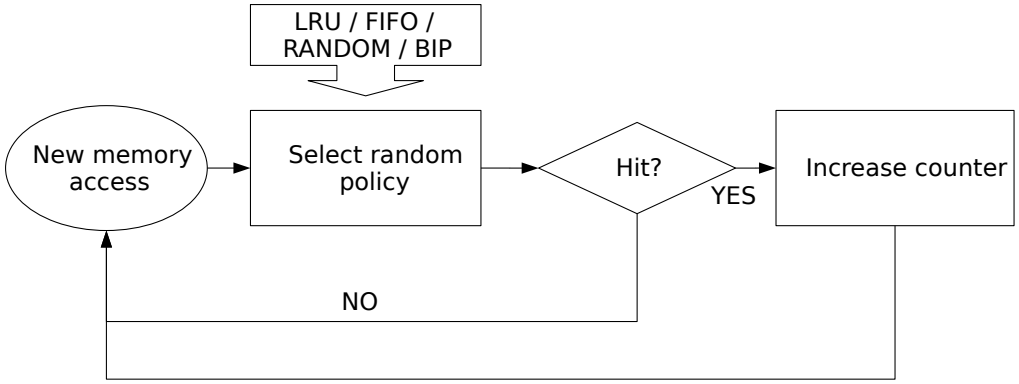
Simplicity is one advantage of this approach since it leverages the developer the decision to perform a policy change. However, it relies on the developer's knowledge and experience with the program memory access pattern and cache architecture, which complicates its usage in practice. An inexperienced developer may need several attempts to select the place in the code for the policy-changing call (mostly a trial-and-error process). Thus, a downside of the approach is its coarse granularity, the fact of being sub-optimal, and the required time to adjust the calls. It is a HITL (Human-In-The-Loop) process, without any algorithm or automation behind it. The framework exclusively changes the policy used in the next instruction after the inserted call in the code.

4.2.2 Naive search approach

In the naive search approach, depicted in Figure 4.2, the framework chooses the policy used for every memory access. It uses a variable seed to run a random function that chooses between all the eviction policies and applies it to the cache memory. With this approach, the policy selected could change constantly along the execution of the code without any logic. The results obtained with the approach are different in every execution and it could be used as a baseline to compare the improvements of other approaches. The current implementation allows the *Naive search approach* to select between LRU, FIFO, RANDOM, and BIP, and as input parameter, it allows the developer to restrict the minimum number of memory accesses before to change the policy. The

results expected with this method are not close to the optimal ones. Because it has no verification of the hit ratio during the execution nor the historic use of policies it might trigger a policy change even when the current one is giving acceptable hit ratios.

Figure 4.2 – Overview of the naive search approach.



Source: Author (2023).

4.2.3 Fixed-Window Approach

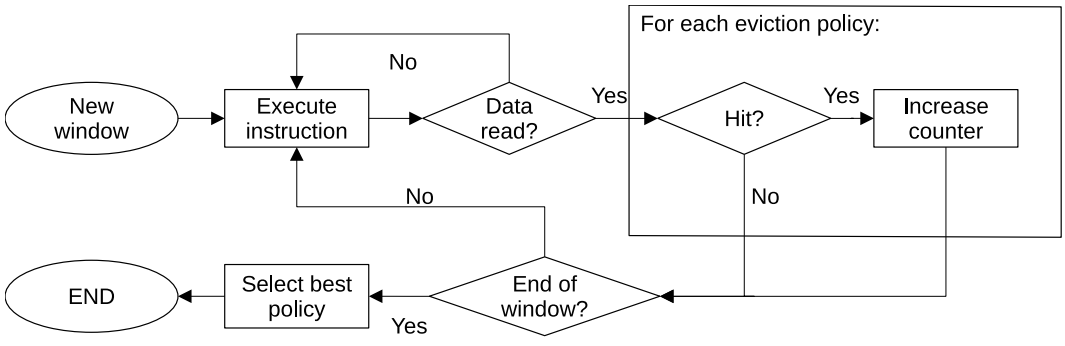
This is the first automated method presented in this work, contrary to the *naive* and *developer* approaches it searches for access patterns in the memory accesses. The idea behind the offline automated approaches is to check the memory accesses sequences during the execution of the code and analyse the behavior of the traditional policies with them, selecting the best policy for each sequence. As is not possible to know the optimal size of a memory access sequence that works better with a policy, the approaches face this inconvenient in different ways. For instance, the *Fixed-Window* method checks a fixed number of memory accesses, that considers as unique sequences, whereas the *Sliding-Window* or the *Online* approaches process some previous memory accesses that incrementally are tested with traditional policies, triggering the change of policy in the cache when a defined threshold is reached.

The Fixed-Window approach works offline, cloning the cache memory structure selected in the framework for each traditional policy. The presented version of the framework handles 4 policies LRU, BIP, FIFO, and RANDOM. The Cachegrind has as result 4 virtual cache memories with the same characteristics. After the generation of the caches the framework runs the code simulating the memory accesses in all the cache memories simultaneously. This method offers a parameter to fix the windows size, the window represents the number of memory accesses considered as a sequence to be examined. During the computation of the memory accesses within the window, every cache memory copy will count the number of hits and misses generated regarding its eviction policy. The process will conclude when the number of memory accesses defined by the parameter are reached, in that moment the hit counters of the 4 policies

are compared and the framework selects the policy with the greatest performance (i.e, the one with the highest cache hit ratio) as the policy to be used in the code inserting the instruction in the line executed at the beginning of the current window. Figure 4.3 presents an overview of the Fixed-Window approach for the process of a window. Whenever there is new data access during the window, it is applied to each eviction policy individually, considering their current status of the caches tag directory. If the memory access is a cache hit, then a counter for the current eviction policy is increased. When the window finalizes, then the counters are compared and the one with the highest value is chosen. The size of the window is smaller than the overall number of memory accesses, hence several windows will be processed during the execution of the code. After every single window, the Fixed-Window approach copies the cache of the selected policy to every other cache memories clones, starting the process again having 4 exact copies of the cache memory. The operation is repeated until the code is finalized, having at the end a traditional policy selected for every window in the code.

The Fixed-Window approach could be implemented online, but it would lead to a substantial hardware overhead because every traditional policy, 4 policies for the current implementation, needs a tag directory and a hit counter bounded to it. It also will be partially divergent in relation to the moment the policies are switched, in the offline mode the policy change is done before the window and in the online mode it should be changed after the window is processed. The online use of the approach will have also an execution overhead due to the copy of the tag directories to the not selected policies and the updating work of the 4 directories while the window is processed.

Figure 4.3 – Fixed-Window approach overview.



Source: Author (2023).

In this approach, the size of the window is the only parameter the developer must set and tweak. This represents the biggest disadvantage of this method as this parameter directly impacts the quality of the eviction decisions and it cannot be changed during the offline process execution. While coarser granularity may include code blocks that could work better with another policy different from the policy selected for the whole window, small window sizes are likely to result in many policy changes. The window size

has another restriction, it must be smaller than the policy counter to avoid overflows during the execution of the method. For instance, if the windows has the maximal allowed size, i.e. the maximal value of the counter, and all the memory accesses result in hits, the counter will reach its maximal value. Having a window size bigger than the counter and having a policy with only cache hits could result in the selection of a policy which is not the recommended for the processed sequence.

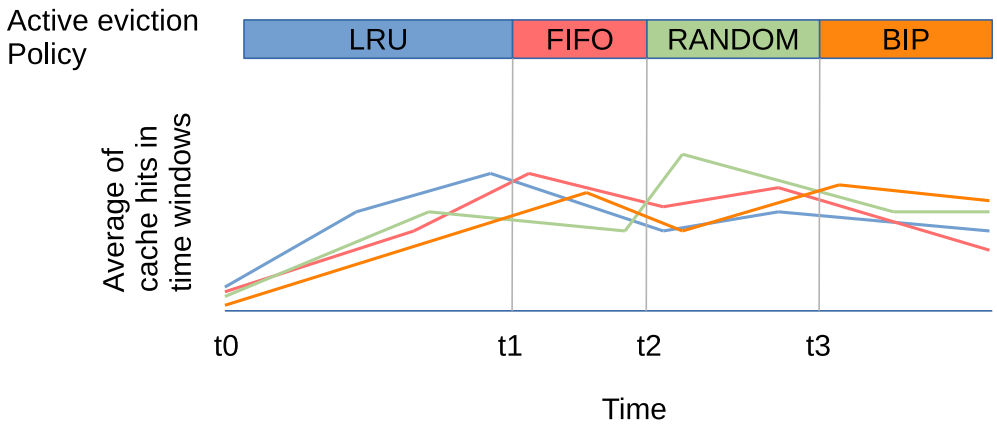
4.2.4 Sliding-Window Approach

The Sliding-Window method is an evolution of the previously presented *Fixed-Window* approach. It is also an offline and automated approach based on windows and handling of memory accesses sequences. Contrary to the *Fixed-Window*, the *Sliding-Window* does not compare behavior of policies in memory accesses sequence, it compare hit ratio trends.

The *Sliding-Window* approach, similarly to the *Fixed-Window*, also uses dedicated caches for each policy and defines a window size from a input parameter. However, here the window defines the number of previous memory accesses used to calculate the average of cache hits, i.e. hit ratio. The framework executes the code offline, simultaneously in the 4 cache memory clones, and uses the same 4 eviction policies given in the *Fixed-Window* method (LRU, FIFO, RANDOM, and BIP), for every single memory access the hit ratio of the previous window is calculated. The algorithm in the framework compares the hit ratios of all the cache copies and selects the best ratio between them. After that, it fixes the policy that will be used generating the output that will be used in the hardware at a later stage. Figure 4.4 shows the result of the Sliding-Window approach using a forged data example. The y-axis represents the average of cache hits considering time windows and the x-axis represents the time. The four colored lines represent the average of cache hits calculated for each data access and policy (LRU, FIFO, RANDOM, and BIP). For instance, in the time instant t1, there is a transition from LRU to FIFO, as the average of hits using FIFO in that data access overtakes the averages of the other three policies. Likewise, in the time instant t2, RANDOM is chosen over the other policies, and finally, in t3 there is a change from RANDOM to BIP.

The technique is equivalent to run the same code parallel using all the traditional policies and switching between them in the real cache memory. The framework uses 5 cache instances altogether, 4 of them with traditional policies but only saving metadata and 1 with metadata and data where the policy is replaced. The difference between the *Sliding-Window* and the *Online approach* is the aging of the information used to take the decision of changing to a new policy, this approach applies aging to the process while the *Online approach* does not. In the current method, only the hits occurred during the processed window are compared, ignoring older hits and behaviors.

Figure 4.4 – Example of eviction policy transitions with the sliding-window approach.



Source: Author (2023).

There are two parameters needed in this approach, the window size, and the threshold necessary to change between policies. In contrast to the *Fixed-Window* approach, the *Sliding-Window* may change the eviction policy with a higher frequency. This could be disadvantageous because it might cause an eviction policy change in every single data access, to avoid this, we use the threshold to establish the boundaries of policy changes. Moreover, the manner this approach works makes it a candidate to be implemented online, even resulting in a big hardware overhead. In a online implementation, the method would need a tag directory for each traditional policy, together with a hit counter that take aging in consideration. Since the comparison between policies is done using the same window size, it could be implemented comparing the hit counters getting the best performing policy.

4.2.5 Online Selection Method

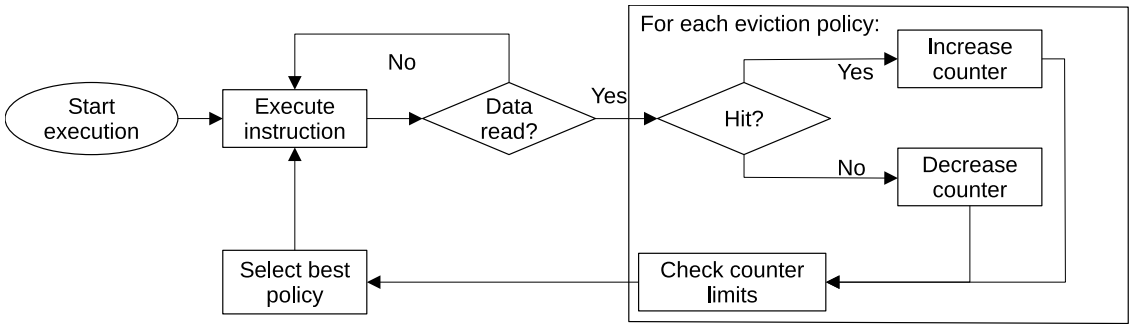
The *Online Selection* approach is an automated method, that despite its name, may be used online or offline. The motivation of the method is the utilization of hits trends but reducing the hardware overhead of the *Sliding-Window* technique. As consequence, it does not consider the aging of the hits, but just the amount of them.

In this approach, each policy has its cache metadata directory and a counter attached to it (similar to the DIP policy proposed by Qureshi et al. (2007)). The counter is defined with a maximal, a minimal, and a starting value. During the program execution, each policy modifies its cache metadata simultaneously according to the executed instruction. A cache hit increments the counter by one. Inversely, a cache miss decrements the counter by one. The difference between our *Online Selection* method and DIP (QURESHI et al., 2007) is that DIP is limited to only two policies, whereas our online selection method can use more than two. This minor modification, however, completely changes the way the counter is used. For instance, DIP has a unique counter that is

shared by the two policies. Here, we have a dedicated counter per policy. DIP uses the medium value of the counter (for example, 512 when the counter size is 1024), while our method uses a pre-defined threshold.

The eviction policy is selected by taking the option with the greatest counter. To avoid excessive changes among policies, a threshold-based mechanism is introduced in the method. The current policy is replaced by another one only if the new one surpasses the counter value of the current one by the threshold. Figure 4.5 shows an overview of the online selection approach. Each eviction policy is applied for every instruction causing data access. If a cache hit is encountered, then the corresponding counter is incremented, otherwise, it is decrement. Then, the approach compares the counters against the threshold to select the best policy.

Figure 4.5 – Overview of the online selection approach.



Source: Author (2023).

The advantage of this approach lays in the simplicity of its implementation and the minor computing needed. The determination of the eviction policy can be done offline (through simulation) as well as online in the CPU by having a per-policy counter. It is an evolution of the DIP-Global proposed by Qureshi et al. (2007) that increase the hardware overhead proportionally to the number of policies embedded on it. If we implement the DIP-Global with more than 2 eviction policies and aging to the counters we jump back to the already explained *Sliding-Window* approach.

4.2.6 Set Dueling

In (QURESHI et al., 2007) the authors present DIP Set-Dueling (DIP-SD) as a solution to apply BIP and LRU to the same cache memory in runtime without a considerable hardware overhead. This method allows the use of 2 eviction policies for the same memory while the memory may switch between them automatically according to the evolution of the working set. The limitation of this technique underlie in the limitation of the number of eviction policies used. In this work the Set Dueling was adapted to include a third eviction policy, LRU, BIP, and FIFO.

Qureshi et al. (2007) use the central limit theorem and statistic tables to demonstrate that choosing between 32 and 64 sets is enough to determine in run time the best

policy between two policies with a probability bigger than 95%. The authors expose that considering the number of misses per set in a cache memory as a independent random variable, exist a number **n** of sets small to avoid affecting the mean and standard deviation of the variable but at the same time sufficiently large such it could be approximated to a Gaussian distribution by the central limit theorem. They consider that it could be done with two subsets of the cache sets with the same size, and it could exist a new Gaussian random variable that represent the difference between total number of misses of the selected sets of both policies. The probability of choose the better policy based in the difference of the total number of misses is calculated by:

$$P(Best) = 1 - P(Z > \sqrt{n} \cdot r), \text{ where } r = \frac{\Delta}{\sigma} \quad (4.1)$$

$$\Delta = |\mu_1 - \mu_2| \quad (4.2)$$

$$\mu_1 = \text{Mean of number of misses in set using only first policy} \quad (4.3)$$

$$\mu_2 = \text{Mean of number of misses in set using only second policy} \quad (4.4)$$

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2} \quad (4.5)$$

$$\sigma_1 = \text{Standard deviation of number of misses in sets using only first policy} \quad (4.6)$$

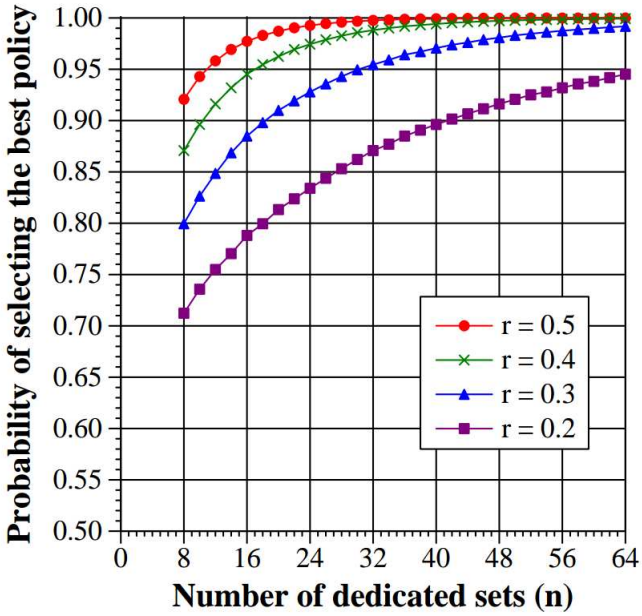
$$\sigma_2 = \text{Standard deviation of number of misses in sets using only second policy} \quad (4.7)$$

Figure 4.6 shows the probabilities of getting the best policy obtained from standard statistical tables depending on the number of sets used for each policy. The Y axis shows the probability and the X axis the number of sets used by policy. Each line in the graph represents the relation between the mean and standard deviation of the difference between the number of misses per set when only one policy is used in the code. Qureshi et al. (2007) claim that most of the benchmarks used in their experiments have a relation, **r**, bigger than 0.2. Considering the worst case, r=0.2 and 32 sets, the probability of choosing the best policy based on the number of misses of the selected sets is bigger than 85%.

As we do with the *Online Selection Method*, we propose to use more than two policies in the Set-Dueling approach. Following the same logic presented by Qureshi et al. (2007), it is possible to admit that more policies may be added to the method choosing between the best one using some sets dedicated to it. However, some restrictions appear when the number of policies increases. New policies added to the Set-Dueling method increment the number of sets dedicated to a single policy, and this affects the final hit ratio of the cache, because it has more sets that are not following the best policy. Furthermore, the number of sets dedicated to fixed policies should be small, since it does not affect the mean and standard deviation of the number of misses obtained in the following sets when compared with a cache where only a single policy is used.

We implement three policies in the Set-Dueling approach and we create three

Figure 4.6 – Relation between probability of selecting best policy with Set Dueling and the dedicated number of sets.



Source: Qureshi et al. (2007).

cache hit counters, one per policy. The counters show the number of hits that a policy has in the latest memory accesses occurred in its memory sets. Because the counters may overflow during the process, the counter only considers the n latest accesses. In our approach, we compare the values of the counters, which returns the best policy. The resulting best policy decides the policy that is used in the following sets. Because we use the same logic as in (QURESHI et al., 2007) three times, the final probability of choosing the best policy is the combination of the probability of each comparison, $P(Best_{Overall}) = P(A-B) \times P(C-B) \times P(A-C)$. For instance, considering that $r = 0.3$ for the comparisons (A-B, C-B and A-C), and 32 sets per policy according to Figure 4.6, $P(A-B)$, $P(C-B)$ and $P(A-C)$ are 95% making the overall probability of choosing the best policy 85.7%.

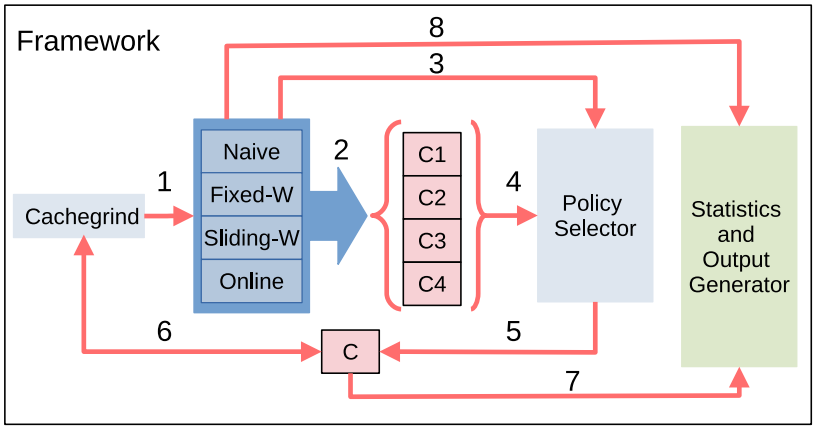
4.3 FRAMEWORK IMPLEMENTATION

We implement the approaches, described in Section 4.2, in a framework to select the best cache eviction policy (the one that maximizes the cache hit ratio). We extended Cachegrind, one of the tools composing the instrumentation framework Valgrind (SEWARD; NETHERCOTE; WEIDENDORFER, 2008), where we integrated all the approaches explained in this work. Cachegrind is a cache profiler that allows the user to simulate a program interacting with the cache memory. Parameters such as the size, associativity, and line size of the cache levels can be parameterized. In our extension, we consider the first level (L1) of both instruction and data caches, and the

last-level of the cache memory (LLC). Because Cachegrind does not support third or fourth-level caches these memories were excluded from our work. Cachegrind returns the statistics related to the cache memory after simulating the execution of a program. It determines the instruction and data cache memory reads status, providing the L1 and LLC misses and the total number of reads, which allows the user to obtain the overall memory hits and misses. Similarly, Cachegrind offers the statistics of the data cache writes, giving the misses and hits generated during the writing process.

The original Cachegrind supports only the LRU cache eviction policy. We extended it by also implementing FIFO, RANDOM, and BIP policies, as well as the described approaches. We added a command line parameter to select one of the approaches described in Section 4.2 to be executed during the program simulation. Figure 4.7 shows the internal structure and logic of the proposed framework extension. Depending on the selected approach (1), it creates diverse temporary cache memories (2) to be used individually by each eviction policy depending on the selected approach. These caches are represented in Figure 4.7 as C1, C2, C3, and C4. The selected approach also determines how the best policy is selected (3) based on the statuses of the temporary caches (4). The final cache memory, C, is generated using the policy selected by the policy selector during the execution of the code (5). The framework uses the final cache C to generate the statistics and outputs at the end of the execution of the program (6). The framework provides an overview of the process (7) depending on the approach (8) to know which policy was chosen by the framework during the execution of the program and to compare the performance of the outcome with the individual policies. Our framework with the modified Valgrind/Cachegrind code is available online.[1]

Figure 4.7 – Overview of the proposed framework for cache eviction policy selection.



Source: Author (2023).

In the next sections, we discuss the implementation details of each approach.

[1] <https://github.com/donxergio/cachegrind.git>.

We close the section by discussing the differences among the approaches, the output generated by the framework, and implementation aspects.

4.3.1 Developer Approach

Valgrind provides an interface, named client request, that allows user programs to pass queries and information to Valgrind and its current tool (Cachegrind in our case). We extended the client request interface to support policy changing at run-time. We added a new macro, `CHANGE_EVICT_POLICY(policy_id)`, which passes the new chosen policy to cachegrind, as exemplified in Figure 4.8 (lines 5 and 7).

Figure 4.8 – Developer approach code implementation.

```
1   #include <valgrind.h>
2   #include <cachegrind.h>
3
4   int main() {
5       CHANGE_EVICT_POLICY(FIFO_POLICY);
6       ...
7       CHANGE_EVICT_POLICY(RANDOM_POLICY);
8   }
```

Source: Author (2023).

To ensure smooth integration of this new approach within Cachegrind, we introduced a control variable specifically designed to keep track of the currently active eviction policy. This variable serves as a single point of truth within the program, dictating which eviction policy should be applied at any given time. As Cachegrind runs, it listens for incoming client requests. When a policy-changing request arrives via our extended client request interface, the control variable is updated accordingly. This effectively switches the eviction policy to the newly specified one without requiring the tool to be restarted or the profiled program to be relaunched. Every time Cachegrind encounters a memory access operation while profiling the target application, it first consults the control variable to determine the current eviction policy. Depending on the value stored in this variable, Cachegrind selects the appropriate algorithm for cache eviction. By doing so, we allow for dynamic, real-time changes to the cache eviction policy without affecting the ongoing profiling session.

4.3.2 Naive Approach

The **Naive Approach** employs a pseudo-random number generator that comes integrated with Valgrind. This generator utilizes stored data in memory and transforms it into UINT (Unsigned Integer) values for subsequent use. While the memory addresses from which the data are retrieved remain consistent, the UINT values can vary depending on the state of the memory during program execution.

Figure 4.9 highlights the section of code responsible for calling Valgrind's method to procure a pseudo-random number, which is notably executed at line 5. To obtain a manageable output, the modulo operator is invoked to convert the UINT value into a number range between 0 and 3. This numeric output then serves as an index to select the next eviction policy that will be activated.

Figure 4.9 – Naive approach code section that selects the next eviction policy.

```
1  ...
2   if (current_cache_replacement_policy == NAIVE)
3   {
4       // Must be adapted to the number of policies
5       current_adaptative_cache_replacement_policy = VG_(random)(NULL) % 4;
6   }
7
8   switch (current_adaptative_cache_replacement_policy)
9   {
10  case LRU_POLICY:
11      cachesim_setref_is_miss = &cachesim_setref_is_miss_lru;
12      policies[0].uses++;
13      break;
14  case RANDOM_POLICY:
15      policies[3].uses++;
16      cachesim_setref_is_miss = &cachesim_setref_is_miss_random;
17      break;
18  case FIFO_POLICY:
19      policies[1].uses++;
20      cachesim_setref_is_miss = &cachesim_setref_is_miss_fifo;
21      break;
22  case BIP_POLICY:
23      policies[2].uses++;
24      cachesim_setref_is_miss = &cachesim_setref_is_miss_bip;
25      break;
26  default:
27      cachesim_setref_is_miss = &cachesim_setref_is_miss_lru;
28      policies[0].uses++;
29      break;
30  }
31  ...
```

Source: Author (2023).

Once the policy index is determined, a switch statement in the code proceeds to assign the function address responsible for the corresponding policy behavior. This assignment targets a policy function, referred to as (**cachesim_setref_is_miss**). The system is designed in a way that it can easily accommodate additional policies; one would only need to update the switch statement and adjust the modulo operation accordingly.

This **Naive Approach** can be utilized in both online and offline modes, offering a degree of flexibility. Moreover, it comes with the benefit of incurring zero hardware overhead. However, its primary drawback lies in its unpredictable nature. The randomness introduced by the pseudo-random number generator does not ensure any form of optimization. The policies selected could, at times, be sub-optimal, leading to inconsistent performance metrics.

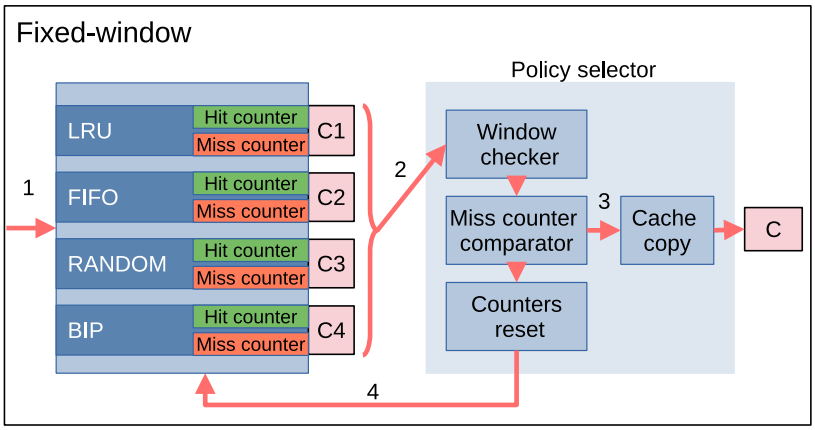
By employing this approach, developers are given a method that is easy to implement and adapt but must also grapple with the trade-off of unpredictability. Therefore, while this approach offers simplicity and minimal resource requirements, it may not be suitable for all scenarios.

4.3.3 Fixed-Window Approach

The **Fixed-Window Approach** represents a method for dynamic cache policy selection, taking into consideration the real-time performance metrics of four well-known cache management policies: LRU, FIFO, RANDOM, BIP.

Figure 4.10 offers a graphical representation detailing the internal mechanics of this approach. Initially, incoming instructions (annotated as (1) in the figure) are dispatched to specialized policy blocks. Each policy block comprises two counters one for tracking hits and another for tracking misses. These counters are intricately connected to separate, individual cache memories labeled as C1 for LRU, C2 for FIFO, C3 for RANDOM, and C4 for BIP.

Figure 4.10 – Fixed-window approach implementation overview.



Source: Author (2023).

The framework is designed to continually monitor the status of these individual caches during code execution. Specifically, it checks whether the requested data is available in the cache, updating the hit and miss counters as appropriate. An arrow marked as (2) in Figure 4.10 illustrates the flow of this information to a central component called the Policy Selector.

The role of the Policy Selector is twofold. Firstly, it amasses and scrutinizes the performance metrics for each individual policy. Secondly, it manages the window, checking if the pre-defined number of instructions has been processed. Once the window concludes, the Policy Selector embarks on a comparative analysis of the miss counters associated with each policy. The aim here is to identify the policy with the lowest number of misses, thereby inferring it as the most efficient under the current conditions.

After identifying the most efficient policy, the data corresponding to this "winning" window is transferred to the final, operational cache, indicated as (3) in Figure 4.10. Subsequently, all counters are reset to their initial state, thus setting the stage for the next window to restart its operation.

This approach provides a dynamic and adaptive solution to cache management, constantly adjusting to the optimal policy based on real-time performance metrics. However, it's essential to note that the granularity of the window can have a significant impact on the system's adaptability and overall performance. Therefore, selecting an appropriate window size parameter, which defines the number of instruction processed before the selection of the best policy, is crucial for balancing adaptability and computational overhead.

Figure 4.11 shows the pseudocode of the implementation of this approach.

Figure 4.11 – Fixed-widows approach pseudocode.

```
1  Process all policies ()
2  data_blocks++;
3  IF (policy is FIXED_WINDOW)
4          {
5              // Check if the window was already processed
6              IF (data_blocks == WindowSizeParameter)
7              {
8                  /*Copy the best policy cache to all other caches*/
9                  FOR all policies:
10                 {
11                 copy_cache(bestpolicy.InstructionMemory, other_policy[x].
                    InstructionMemory)
12                 copy_cache(bestpolicy.DataMemory, other_policy[x].DataMemory)
13                 copy_cache(bestpolicy.LastLevelMemory, other_policy[x].
                    LastLevelMemory)
14                 }
15                 // Clean last window to start a new one
16                 bestpolicy.Memories = 0
17                 // Restart processing
18                 data_blocks=0;
19             }
20         }
```

Source: Author (2024).

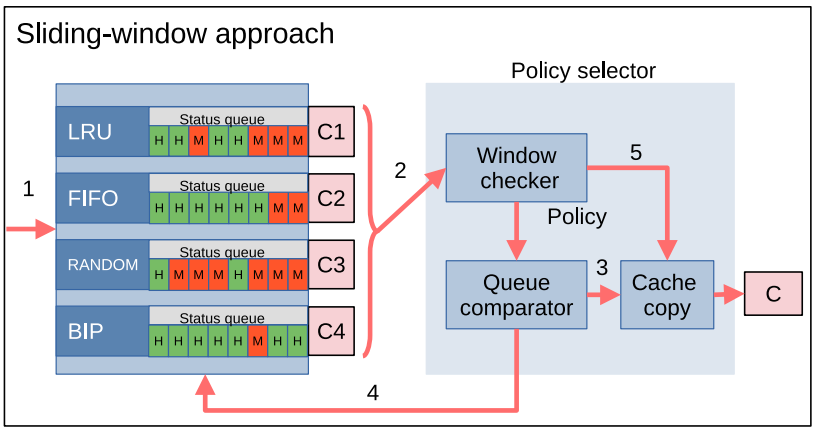
4.3.4 Sliding-Window Approach

The **Sliding-Window Approach** is another methodology that we have integrated into Cachegrind, alongside the previously described Fixed-Window Approach. This approach also considers the four primary cache eviction policies: LRU, FIFO, RANDOM, and BIP. Like its Fixed-Window counterpart, each policy block contains its unique cache memory (C1 for LRU, C2 for FIFO, C3 for RANDOM, and C4 for BIP) and a fixed-size status queue. The framework uses this queue to record the outcomes of cache accesses, classified as either 'hit' or 'miss.'

As illustrated in Figure 4.12, incoming instructions (labeled as (1) in the figure) are dispatched to these specialized policy blocks. The Policy Selector is invoked for

each executed instruction to both scrutinize the state of the window and examine the status queues of individual policy blocks. For every instruction, the four policy queues are compared, and the framework selects the policy with the least number of misses for use in the final operational cache (labeled as (3) in the figure). Misses are visualized as red squares with the letter 'M' in the diagram. In case of a tie in the number of hits across multiple policies, a default policy is employed.

Figure 4.12 – Sliding-window approach implementation overview.



Source: Author (2023).

Though the Fixed-Window and Sliding-Window Approaches share some commonalities, such as the usage of time windows for policy selection, they differ in significant ways. The Fixed-Window Approach waits for the entire window to elapse before computing the most optimal policy, effectively introducing policy changes only at predefined intervals. For example, if the time window is set at 5000 instructions, policy modifications occur every 5000 instructions.

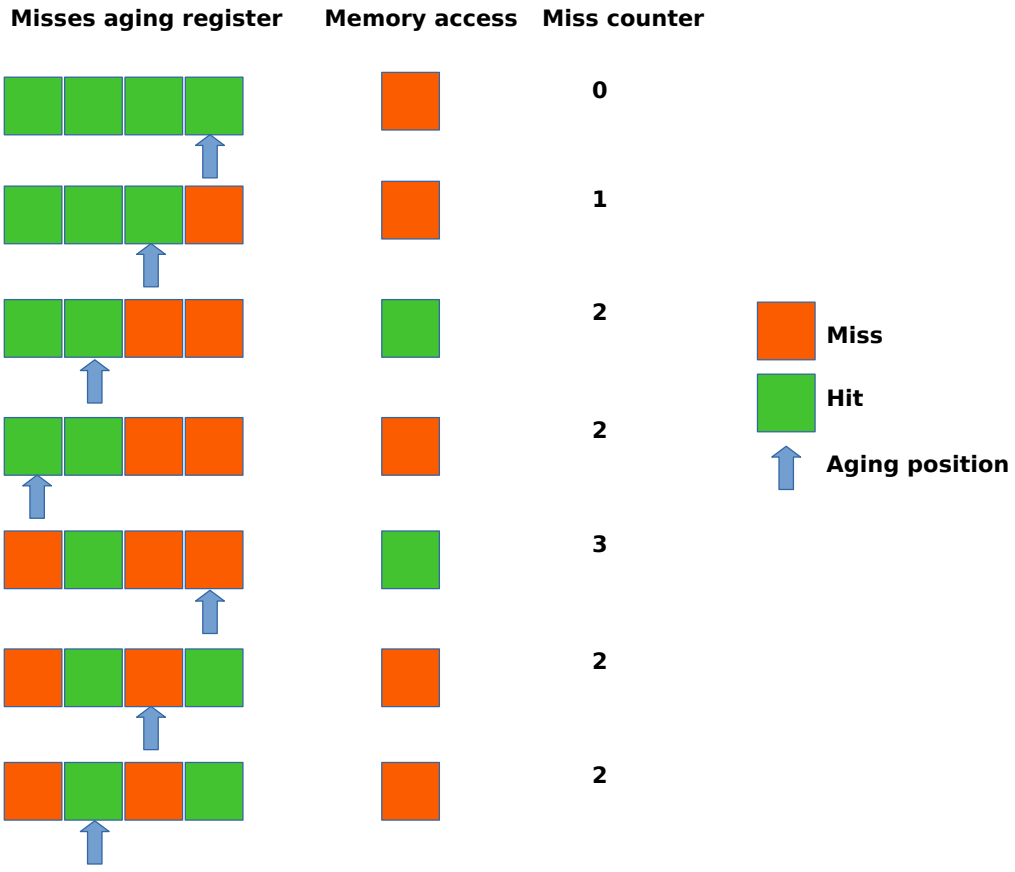
In contrast, the Sliding-Window Approach permits more frequent policy adjustments within the same time window. It does so by evaluating the efficiency of each policy on an instruction-by-instruction basis. When configured with identical time windows, the Sliding-Window Approach is anticipated to outperform its Fixed-Window counterpart. This is due to its ability to adapt more quickly to different sections of code, leading to overall improved performance, albeit at the cost of additional overhead for frequent policy switching.

A remarkable feature of the Sliding-Window Approach is its nuanced consideration of the aging of the hits ratio. Unlike other approaches, the counters employed here work together with a register tailored to match the window size. For instance, a window of size 16 would necessitate a 16-bit register and a counter for each policy. This counter-register configuration enhances the algorithm's adaptability and responsiveness to temporal variations in code behavior.

Figure 4.13 depicts an example of use of the miss counter and register where we implement the aging technique. Initially the register has all his bit to 0, which represents

a hit. The blue arrow represents the bit position of the register where the aging is applied. The bit pointed by the arrow will be updated with the result of the latest memory access. Before the update of the register we compare the previous value of the register bit, a change in the value of the register updates the miss counter. When the bit changes from 0 to 1 (hit to miss) the counter is increased by 1, and when the bit changes from 1 to 0 (miss to hit) the counter is decremented by 1. This implementation limits the counter between the number of bits of the register and 0.

Figure 4.13 – Register usage for the aging implementation.



Source: Author (2023).

Figure 4.14 shows the pseudocode of the implementation of aging control.

4.3.5 Online Selection Approach

The **Online Selection Method** constitutes another enhancement that we have engineered into Cachegrind. This method involves augmenting each policy block with a specialized miss counter, as illustrated in Figure 4.15. Whenever Cachegrind processes an incoming instruction, the corresponding policy blocks are activated, and their associated miss counters are updated if a cache miss occurs.

As depicted in Figure 4.15, arrow (1) indicates the incoming instructions to the policy blocks from Cachegrind. For every individual instruction, the Policy Selector then

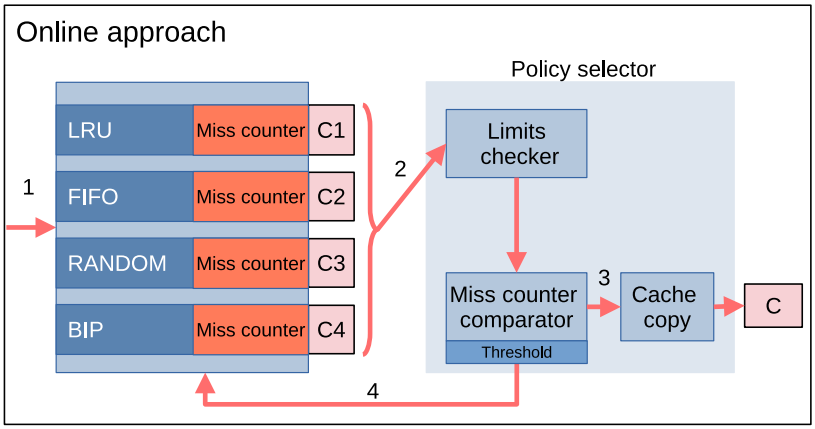
Figure 4.14 – Aging control pseudocode.

```
1     int temp = 0;
2     IF (Function disable)
3     {
4        return last_value;
5     }
6        IF(last_value is 1){
7          Increase number of misses
8        }
9        else{
10          Decrease number of misses
11       }
12       IF (last_value != values[Historic_index])
13       {
14          // The value to be returned should be updated
15          IF (last_value is 1)
16          {
17             Increase historic_miss_counter; // New miss
18
19          }
20          ELSE
21          {
22             Decrease historic_miss_counter; // New hit
23          }
24          values[Historic_index] = last_value;
25       }
26       Increase Historic_index
27       IF (Historic_index >= historic_max)
28       {
29          Historic_index = 0;
30       }
31
32       return historic_miss_counter;
```

Source: Author (2024).

examines the four miss counters (indicated by arrow (2)). Subsequently, it selects for the final cache (labeled as C in the figure) the policy corresponding to the smallest miss counter (indicated by arrow (3)).

Figure 4.15 – Online approach implementation overview.



Source: Author (2023).

A distinctive feature of this method is the incorporation of upper and lower bounds for the counters and a configurable threshold to modulate the frequency of

policy changes. This threshold mechanism mitigates the risk of erratic policy switching by requiring a substantial differential between counters before triggering a switch. In other words, a new policy will only be activated if its counter value is not only smaller than that of the current policy but also exceeds a predefined threshold.

To elucidate, let's consider an example involving a threshold of 5 units. Suppose after the execution of instruction n, the current selected eviction policy is BIP with a miss counter value of 200, while FIFO, RANDOM, and LRU have miss counter values of 250, 250, and 204 respectively. If instructions $n+1$ through $n+5$ result in misses for BIP, its miss counter will increment to 205. Simultaneously, if these same instructions are hits in the LRU policy block, its miss counter would decrement to 199. After the execution of instruction $n+5$, the differential between LRU and BIP counters is 6, which surpasses the preset threshold of 5 units. Consequently, the Online Selection Method would switch to the LRU eviction policy for subsequent instruction execution.

This mechanism, by dynamically adjusting to the program's behavior, not only optimizes cache performance but also minimizes the overhead introduced by frequent policy changes. The Online Selection Method thus strikes a balanced trade-off between adaptability and stability, making it particularly suitable for complex workloads.

Figure 4.16 shows the pseudocode of the implementation of Online approach.

Figure 4.16 – Online approach pseudocode.

```
1 check_misses()
2 update historic values()
3 // Apply the threshold before switching
4 FOR all policies{
5     IF (Misses[Policy] < Misses[Best_Policy] - Threshold)
6     {
7         Best_Policy = Policy;
8     }
9 }
```

Source: Author (2024).

4.3.6 Set Dueling Approach

In contrast to the original Set Dueling mechanism presented in (QURESHI et al., 2007), our implementation within the Cachegrind framework focuses on selecting among three specific policies: LRU, FIFO, and BIP. To accommodate this, we have introduced unique counters for each policy, augmented with aging registers, akin to the methodology used in the Sliding-Windows approach, as discussed in Section 4.3.4.

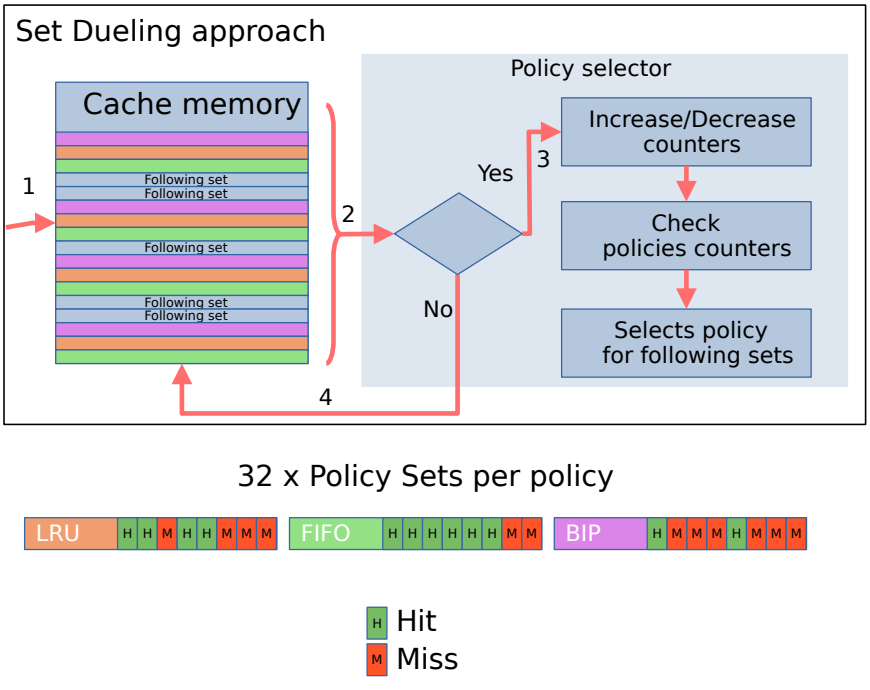
Initially, the cache memory is partitioned into four distinct sections. The first three sections, each comprising 32 sets (totaling 96 sets), are individually linked to one of the three eviction policies (LRU, FIFO, BIP). These sets are referred to as "policy sets." The remaining sets in the cache memory serve as "following sets" that will adopt the policy deemed most efficient at any given time.

For each eviction policy, we maintain a miss counter paired with an aging register. These counters and registers are responsible for tracking the number of cache misses that occur when memory accesses are directed to the sets assigned to each policy. After every memory access where a "policy set" is used, the aging registers are compared, and the policy associated with the smallest counter value is selected for the "following sets."

It is worth noting that the assignment of cache sets to specific policies is performed only once, at the beginning of the execution process. This ensures that all experimental runs adhere to a consistent setup, thereby guaranteeing reproducibility.

Figure 4.17 illustrates the operational flow of our Set Dueling implementation. Upon processing a new memory access (1), the cache set associated with the eviction policy tied to the accessed set is triggered (2). If the accessed set is a "policy set" (3), its corresponding counter is updated based on the logic elaborated in Figure 4.13. Concurrently, the "following sets" have their eviction policies updated based on the counter with the smallest value. On the other hand, if the accessed set is a "following set" (4), the counters remain unaffected, and the most recently determined optimal policy is applied for cache eviction.

Figure 4.17 – Set dueling with 3 policies implementation overview.



Source: Author (2023).

Figure 4.18 shows the pseudocode of the implementation of Set Dueling with 3 policies approach.

Figure 4.18 – Set Dueling approach pseudocode.

```
IF set IS from Policy1
{
   Check_miss_or_hit(Policy1)
   Update_Historic_Values(Policy1)
}
IF set IS from Policy2
{
   Check_miss_or_hit(Policy2)
   Update_Historic_Values(Policy2)
}
IF set IS from Policy3
{
   Check_miss_or_hit(Policy3)
   Update_Historic_Values(Policy3)
}
IF set IS following_set
{
   IF (historic_miss_counter[1] > historic_miss_counter[2] && historic_miss_counter
    [1] > historic_miss_counter[3] )
      Check_miss_or_hit(Policy1)
   ELSE IF (historic_miss_counter[2] > historic_miss_counter[1] &&
    historic_miss_counter[2] > historic_miss_counter[3] )
      Check_miss_or_hit(Policy2)
   ELSE IF (historic_miss_counter[3] > historic_miss_counter[1] &&
    historic_miss_counter[3] > historic_miss_counter[2] )
      Check_miss_or_hit(Policy3)
   ELSE
      Check_miss_or_hit(Policy1)
}
```

Source: Author (2024).

4.4 CONCLUSION

In this chapter, we have laid the groundwork for our study on cache replacement policy selection. We began by outlining the assumptions and system model (4.1), providing a clear framework within which our research will be conducted. Next, we introduced the proposed approaches for cache policy selection (4.2), presenting each method individually, including the Developer Approach, Naive Search Approach, Fixed-Window Approach, Sliding-Window Approach, Online Selection Method, and Set Dueling.

Furthermore, we detailed the implementation of these methods within our framework (4.3), offering technical insights into their integration and operation. By providing a systematic overview of our methodology and the tools at our disposal, we have established a solid foundation for our subsequent analysis and experimentation.

Moving forward, we will apply these methods within our experimental setup to evaluate their effectiveness in improving cache performance. Through rigorous testing and analysis, we aim to identify the most efficient cache replacement policy selection strategies, ultimately contributing to advancements in system optimization and performance enhancement.

5 EVALUATION

This Chapter is devoted to the comprehensive evaluation of the four distinct approaches to cache eviction policies investigated in this thesis: Fixed-Window, Sliding-Window, Online Selection, and Set Dueling. The goal of this evaluation is multi-fold:

1. To assess the performance improvements these methods bring over traditional eviction policies, focusing on various metrics such as execution time, cache hits, and cache misses.
2. To understand the computational and memory overheads associated with each method, thereby providing a holistic view of the trade-offs involved.
3. To compare the approaches against each other, highlighting their respective strengths and weaknesses in different scenarios.
4. To explore how the efficiency of these methods varies with different benchmarks and inputs, thereby providing insights into their adaptability and robustness.

All evaluations are conducted using a consistent methodology, defined benchmarks, and a reproducible test environment. We also consider the statistical significance of the results to ensure that the observations are robust and reliable.

The Chapter is organized as follows. Section 5.1 details the experimental setup, including the benchmarks used and the hardware and software configuration. Sections 5.2 and 5.3 show how the traditional policies and the Naive and Developer approaches are used in our research. Sections 5.4, 5.5, 5.6, and 5.7 provide in-depth evaluations for the Fixed-Window, Sliding-Window, Online Selection, and Set Dueling approaches, respectively. Finally, Section 5.8 offers a comparative analysis of the approaches, summarizing key findings and implications.

5.1 EXPERIMENTAL SETUP

For the purpose of evaluating the four approaches proposed for cache eviction policy selection (Fixed-Window, Sliding-Window, Online Selection, and Set Dueling), it was crucial to employ benchmarks that are representative of real-world computational tasks. To this end, we selected the **San Diego CortexSuite** benchmark suite as our primary testbed. CortexSuite is a comprehensive suite of benchmarks that encompasses algorithms and datasets from diverse computing domains, such as machine learning, natural language processing, and computer vision (THOMAS et al., 2014).

The benchmarks chosen for this study are designed to put significant pressure on the memory hierarchy, thereby allowing us to comprehensively evaluate the performance of our eviction policy selection framework. Specifically, we have included the following benchmarks:

1. **Principle Component Analysis (PCA)**: A widely used technique in machine learning for dimensionality reduction.

2. **Singular Value Decomposition (SVD)**: Another method often used in machine learning and data science for feature extraction.

3. **Image Stitch**: Commonly utilized in image processing to combine multiple overlapping images into a panoramic image.

4. **Latent Dirichlet Allocation (LDA)**: A generative probabilistic model primarily used in natural language processing.

5. **SIFT (Scale-Invariant Feature Transform)**: An algorithm widely used for object recognition in computer vision tasks.

These benchmarks are particularly relevant for embedded systems, where memory efficiency is crucial for performance. They are often employed in application domains such as autonomous vehicles, robotics, and IoT devices.

To ensure the robustness and generalizability of our evaluation, we used multiple input sets for each benchmark. CortexSuite provides various standard input sets that differ in size and complexity. These are categorized as small, medium, and large for general-purpose algorithms. For image processing related algorithms, the inputs are categorized as CIF (Common Intermediate Format), QCIF (Quarter CIF), and HD (High Definition).

By varying the input data size and complexity, we aim to simulate real-world variability and to assess how adaptable each approach is under different memory pressure conditions. This strategy also allows us to explore any performance variations that might be attributable to the nature or size of the input data.

To conduct a comprehensive evaluation of our proposed framework, we executed each benchmark on our modified Cachegrind simulator under various settings and parameters. This allowed us to assess the sensitivity of our eviction policy selection approaches to different cache configurations and operating conditions.

Cache Size Variations We experimented with multiple cache sizes to analyze the impact of cache capacity on performance. Specifically, we considered cache sizes of 4, 8, 16, 32 KB[1].

Eviction Policies We evaluated the performance of four standard cache eviction policies: LRU (Least Recently Used), FIFO (First-In, First-Out), RANDOM, and BIP (Bimodal Insertion Policy). For the BIP policy, we followed the configuration in prior work (QURESHI et al., 2007; ARAUJO et al., 2021), setting the bimodal probability parameter (ϵ) to 1/64.

[1] Cache size here refers to the size of the cache partition allocated to a particular task.

Cache Line and Associativity The cache line size was set to 32 bytes, and we used a 4-way set associative cache for all our experiments.

Window size For the Fixed-Window approach, we considered various window sizes ranging from 128 to 16384 cache data accesses. These configurations are tested to choose the best configuration.

Similarly, for the Sliding-Window approach, we used the same range of window sizes as in the Fixed-Window approach.

For the online selection approach, we set the maximal counter values from 128 to 16384 and a minimum value of 0. We also varied the threshold settings, with values ranging from 1 to 2048.

Baseline In our research, we adopted the LRU (Least Recently Used) cache eviction policy as a foundational baseline. Given its widespread use and prominence in processors, LRU provides a crucial frame of reference for our comparative analysis.

By systematically varying these parameters, we aimed to cover a broad spectrum of potential real-world scenarios, thereby ensuring that our findings are both robust and generalizable.

Using the obtained number of cache misses, together with extra information available from the Cachegrind executions, it is possible to translate the obtained reduction on cache misses into gains in terms of execution time for each benchmark and experiment configuration. For doing that, we used parameters of different processors to calculate the execution times of the benchmarks as done in (ARAUJO et al., 2021):

$$\begin{aligned} Exec\,Time = (I \times CPI) + \\ (D\_MISSES \times MISS\_PENALTY) + \\ ((D\_REFS - D\_MISSES) \times HIT\_PENALTY) \end{aligned} \tag{5.1}$$

Where I is the number of executed instructions, CPI is the cycles per instruction, D_MISSES is the number of data cache misses, $MISS\_PENALTY$ is the cache miss penalty (in processor cycles), D_REFS the number of data references, D_MISSES is the number of cache data misses, and $HIT\_PENALTY$ is the cache hit penalty (in processor cycles). I, D_MISSES, and D_REFS are obtained from the Cachegrind framework output, while the other parameters are defined according to the processor cache memory architecture. Table 5.1 shows the cache-related parameters of the considered processors in this evaluation.

5.2 TRADITIONAL POLICIES

Traditional eviction policies, such as LRU, FIFO, RANDOM, and BIP, have been widely adopted in various computational systems due to their general efficacy across diverse applications. To assess the efficacy of our proposed approaches, it's imperative

Table 5.1 – Parameters of the considered processors.

Processor	Parameters
x86 bosch pentium (WONG; BETZ; ROSE, 2016)	2 instructions per cycle, 3 cycles for a cache hit, and 44 cycles for a cache miss
Intel i7 (HENNESSY; PATTERSON, 2011)	4 inst. per cycle, 4 cycles for L1 hit, 10 for L2 hit, 35 for L3 hit, 100 cycles for DRAM leading to 135 miss penalty
ARM Cortex A8 (HENNESSY; PATTERSON, 2011)	2 instructions per cycle, 1 cycle for L1 hit, 11 cycles for L1 miss, and 60 cycles for L2 miss
ARM Cortex A53 (BANSAL et al., 2018)	2 instructions per cycle, 4 cycles for L1 hit, 19 cycles for L2 hit, and 181 cycles for L2 miss
Related work (QURESHI et al., 2007)	4 instructions per cycle, 6 cycles for a cache hit, and 270 cycles for a cache miss

to comprehend the memory access patterns and performance benchmarks set by these traditional policies. In essence, they serve as a reference point, enabling us to discern the effectiveness of our new methods.

Employing the same benchmarks as with our proposed approaches, we executed the tests for each of the traditional policies. The cache size was varied, consistent with the procedure detailed in Section 5.1. As anticipated, our results exhibited that as cache size increased, the miss rate decreased across all benchmarks and policies. This phenomenon is more pronounced at the start, with smaller cache sizes witnessing higher miss rates, primarily due to their limited storage capacity.

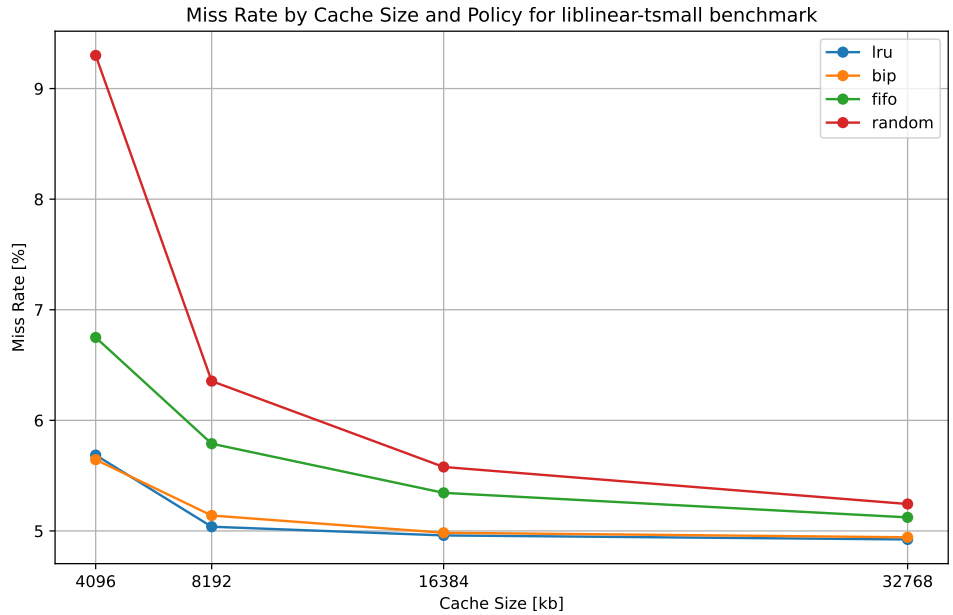
Upon careful analysis, it was observed that beyond certain cache sizes, the marginal reduction in the miss rate becomes negligible. Such points, as shown in Figures 5.1 and 5.2, occur after 8KB and 16KB respectively. This can be attributed to the intrinsic memory footprint of the benchmarks. When the cache is sufficiently large to accommodate the benchmark's working dataset, the miss rate remains to its minimal value for that eviction policy.

An important observation from our results was the persistent, minimal miss rate, irrespective of the eviction policy. This base miss rate stems from the inherent nature of the benchmarks and their data manipulation routines. During execution, as benchmarks rotate or replace their datasets, the cache contends with the need for renewal, leading to inevitable misses. These misses, born out of the dynamics of data usage within algorithms, are inherent and cannot be completely circumvented by any eviction policy.

Upon analyzing the benchmarks, we observed a clear pattern: some benchmarks favor specific eviction policies over others. This relationship highlights how different memory access patterns can influence the effectiveness of cache eviction methods.

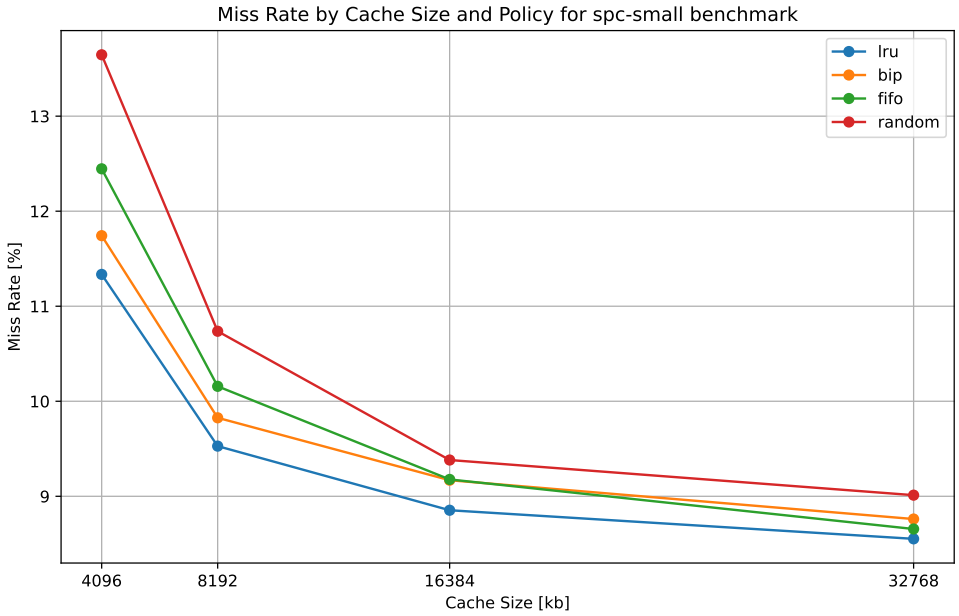
Benchmarks such as those depicted in Figures 5.3 and 5.4 exhibit strong affinity towards the LRU policy. This is evident from the fact that the LRU policy consistently delivers the lowest miss rate throughout the evaluation, regardless of cache size variations. Such benchmarks are termed as "LRU-friendly" because they frequently access

Figure 5.1 – Miss rate evolution for several cache sizes using traditional policies with the liblinear-tsmall benchmark.



Source: Author (2023).

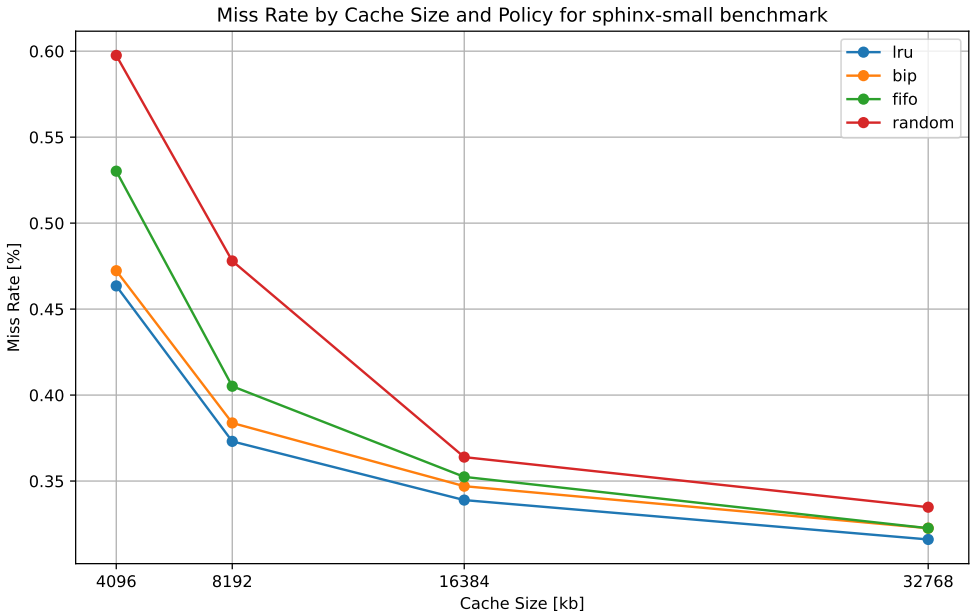
Figure 5.2 – Miss rate evolution for several cache sizes using traditional policies with the spc-small benchmark.



Source: Author (2023).

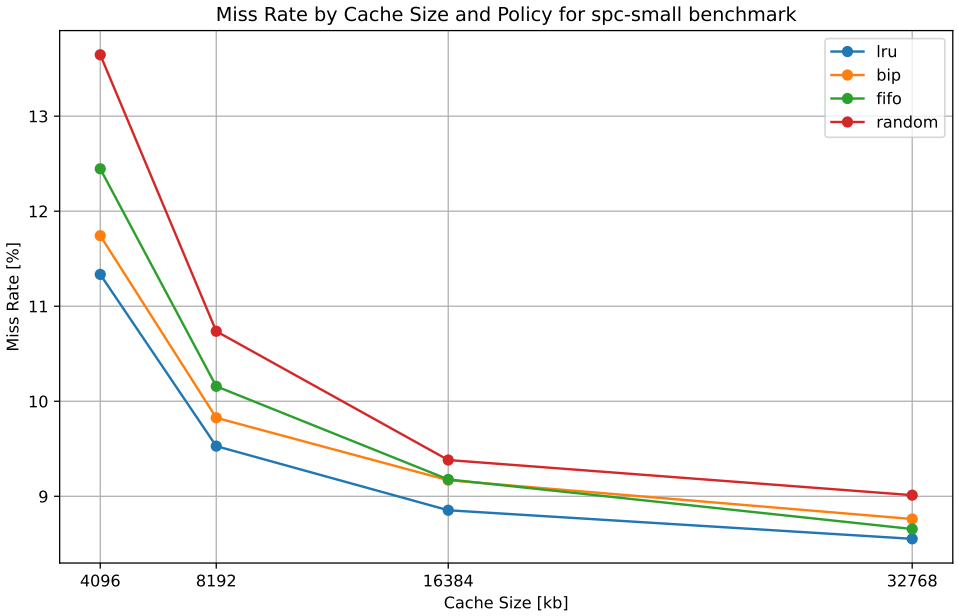
a relatively recent subset of their data, and thus, evicting the least recently used items proves most efficient.

Figure 5.3 – Miss rate evolution for diverse cache sizes using traditional policies with sphinx-small benchmark.



Source: Author (2023).

Figure 5.4 – Miss rate evolution for diverse cache sizes using traditional policies with spc-small benchmark.



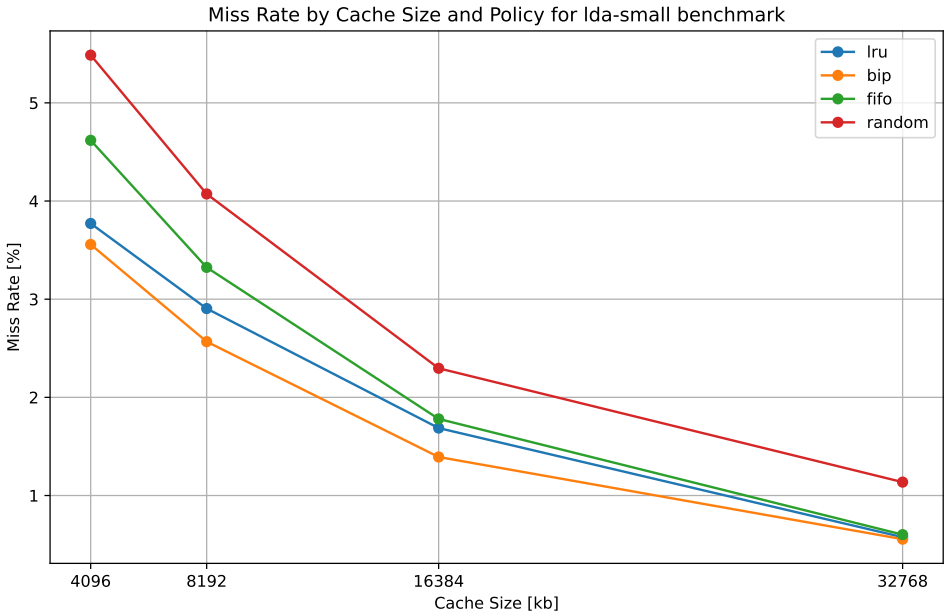
Source: Author (2023).

Conversely, benchmarks like `lda-small` (see Figure 5.5) do not mirror the same affinity towards LRU. Their memory access patterns might be less predictable or might

not necessarily prioritize recently accessed data, making policies like LRU less optimal.

Moreover, certain benchmarks, such as `pca-small` in Figure 5.6, display marked improvements with alternative eviction strategies like BIP, especially at specific cache sizes. For instance, when the cache size is set to 32KB for the `pca-small` benchmark, even though it is smaller than its working set, the BIP policy is adept at retaining essential memory lines in the cache. This results in a significant performance boost compared to the LRU policy.

Figure 5.5 – Miss rate evolution for diverse cache sizes using traditional policies with lda-small benchmark.



Source: Author (2023).

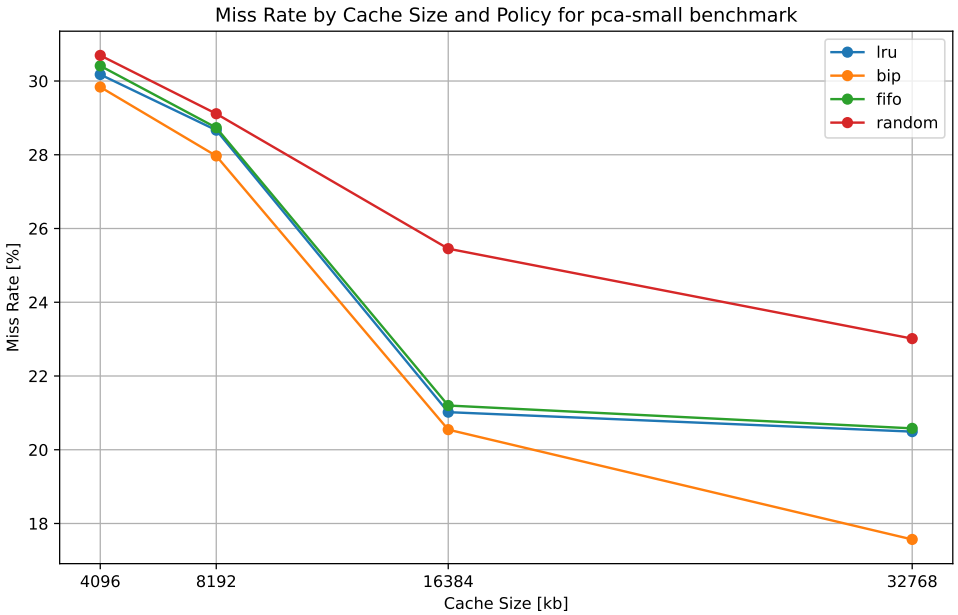
Given the performance metrics established by traditional policies, our overarching goal remains clear: to surpass these benchmarks and further minimize cache misses. Should our proposed methods not yield better results, it would lead us to critically re-evaluate and potentially discard them as viable alternatives.

In the upcoming sections, we present a comparative analysis of our proposed approaches against these traditional benchmarks, laying out a clear picture of their relative strengths, weaknesses, and areas of potential improvement.

5.3 DEVELOPER AND NAIVE APPROACHES

The **Developer** and **Naive** approaches serve as interesting outliers in our evaluation methodology. The Developer approach leans on human expertise and insights into specific sections of the code, thereby representing a best-case scenario of cache policy tuning. The Naive approach, in contrast, symbolizes a worst-case scenario where

Figure 5.6 – Miss rate evolution for diverse cache sizes using traditional policies with pca-small benchmark.



Source: Author (2023).

the selection is entirely random. Given the significant differences in their operational mechanisms compared to automated approaches, we found it beneficial to separate their evaluation from that of the Fixed-Window, Sliding-Window, Online, and Set Dueling methods.

In our experiments, we selected policies for the Developer approach, paying particular attention to three primary sections of the code:

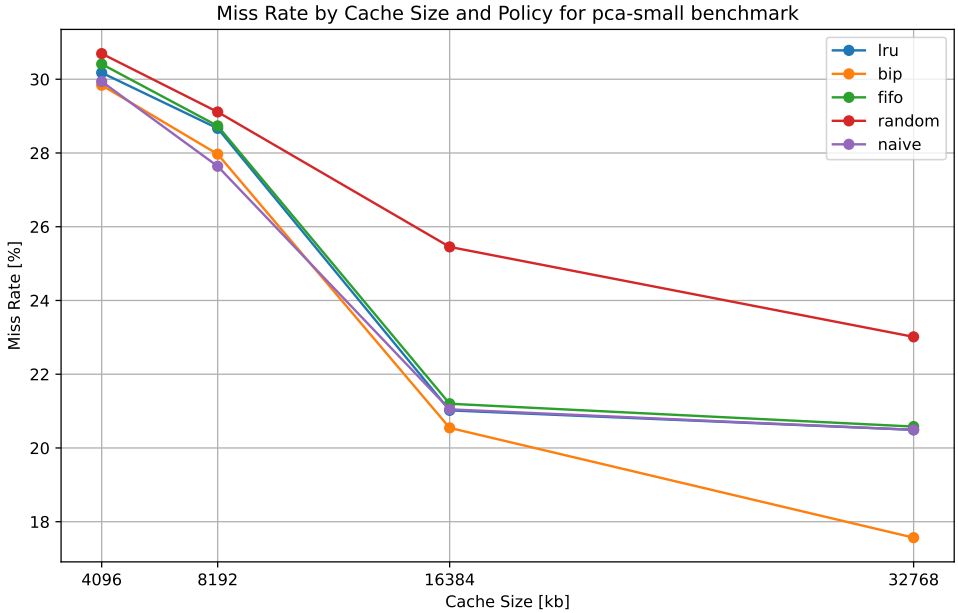
1. The initial part of the benchmark, where cache warming occurs.
2. Looping statements, which are often the hotspots for cache misses.
3. Code blocks where arrays and matrices are manipulated.

By fine-tuning the policies for these specific segments, we achieved significant reductions in cache misses. For example, running `pca-small` with a cache size of 128 KB, we were able to reduce cache misses by 10% compared to using LRU alone, by alternating between LRU and BIP.

Though we did not include the Naive approach in our extensive evaluations, it's worth mentioning that its purely random selection of eviction policies makes it an interesting case for establishing a lower-bound baseline. Since this approach doesn't involve any tuning or intelligent decision-making, any performance gain observed in other methods over the Naive approach can be attributed to their randomness.

In Figure 5.7, the naive approach's performance is compared to traditional policies. While the naive method falls behind BIP for cache sizes of 4kb, 16kb, and 32kb, it outperforms BIP at 8kb. The fluctuation of the miss rate obtained after the execution of the same benchmark using the **Naive approach** in Figure 5.8 highlights the unpredictable nature of the naive approach due to its randomness.

Figure 5.7 – Miss rate evolution for diverse cache sizes using naive approach with pca-small benchmark.

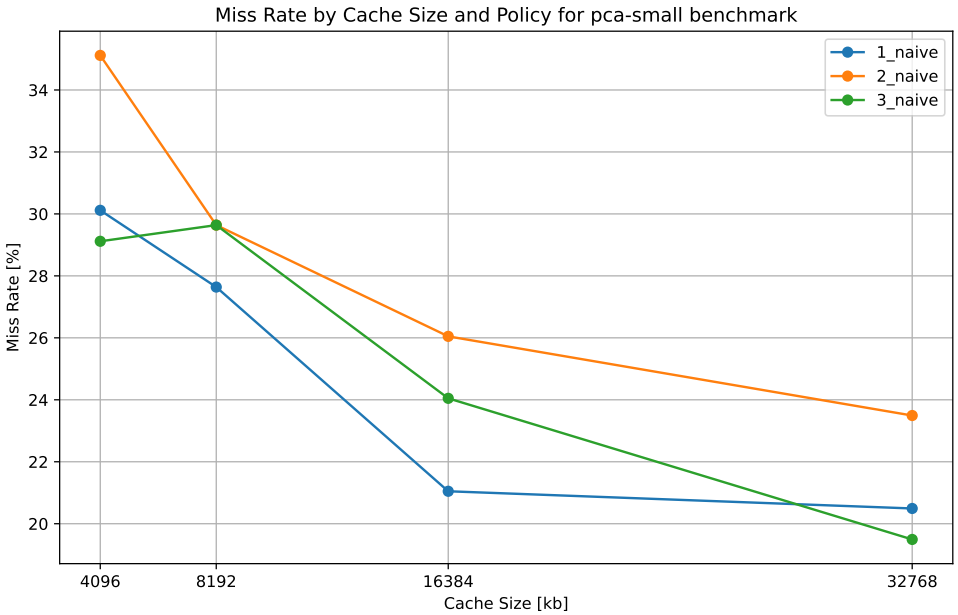


Source: Author (2023).

5.4 FIXED WINDOW EVALUATION

The **Fixed Window** approach offers a distinct methodology to assess memory access patterns. In this method, memory accesses are segmented into fixed-size windows. Each window of memory accesses is concurrently executed using all four traditional eviction policies (LRU, BIP, RANDOM, and FIFO). This concurrent execution takes place in four separate cache clones. Once a window completes processing, the results from the four policies are compared. The cache clone exhibiting the lowest miss ratio is then replicated to the main cache memory and to the other three cache clones. This ensures that subsequent memory access windows start from a consistent state. This iterative process continues until the program execution concludes. Due to the inherent implementation structure of this approach, its online use is unfeasible, leading to considerable overhead.

Figure 5.8 – Miss rate evolution for diverse cache sizes using 3 executions of naive approach with pca-small benchmark.



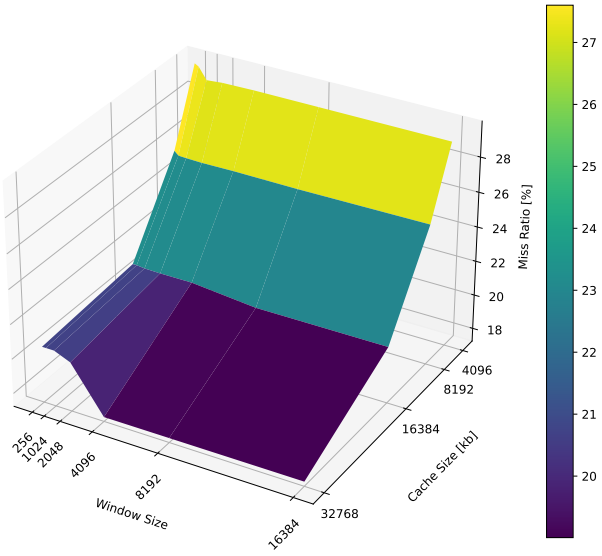
Source: Author (2023).

We subjected various benchmarks to this methodology, intending to juxtapose the results across different window sizes. The aim was to ascertain the optimal configuration and explore a universal parameter applicable across diverse cache sizes.

A clear correlation between the window size and the resultant miss rate was observed. For window sizes spanning between 2 and 256, the outcomes closely maintain the miss ratio invariable as depicted in the figure 5.11. Yet, when window sizes were within the 256 to 16384 range, discernible enhancements emerged. Notably, the choice of optimal window size was intrinsically linked to the specific benchmark and its respective working set. As showcased in Figure 5.9, a appreciable reduction in the miss ratio was achieved when the window size exceeded 4096 memory accesses. In juxtaposition, Figure 5.10 reveals that peak results were realized at a window size of 1024. Interestingly, in both instances, expanding the window further did not confer additional benefits for a consistent cache size.

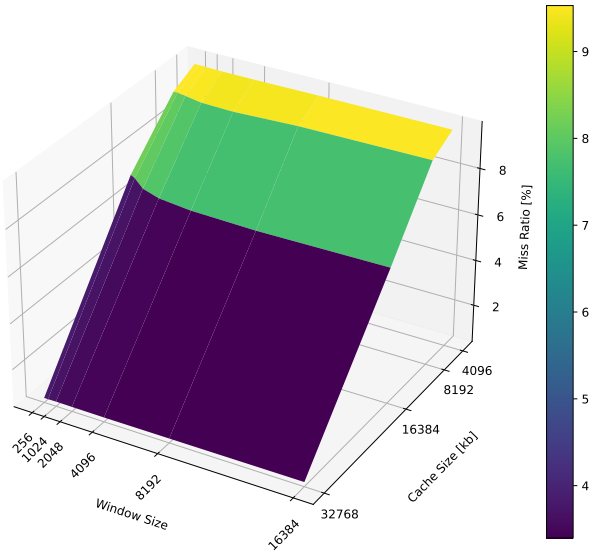
As highlighted in Section 5.2, there is an inverse relationship between cache size and miss ratio, with larger cache sizes generally resulting in reduced miss ratios. Through our research, we discerned that the window size parameter holds the potential to further optimize these outcomes, trimming the miss ratio even in scenarios with the largest cache sizes, as illustrated in Figures 5.12 and 5.9. However, it's noteworthy that when the cache size surpasses the working set, the window size parameter loses its potency in influencing the miss ratio. This phenomenon is clearly exemplified in Figure

Figure 5.9 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with pca-small benchmark.



Source: Author (2023).

Figure 5.10 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with rbm-small benchmark.

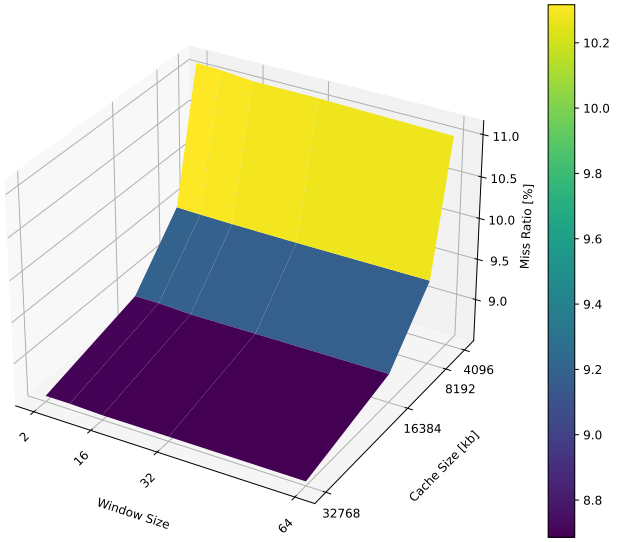


Source: Author (2023).

5.13.

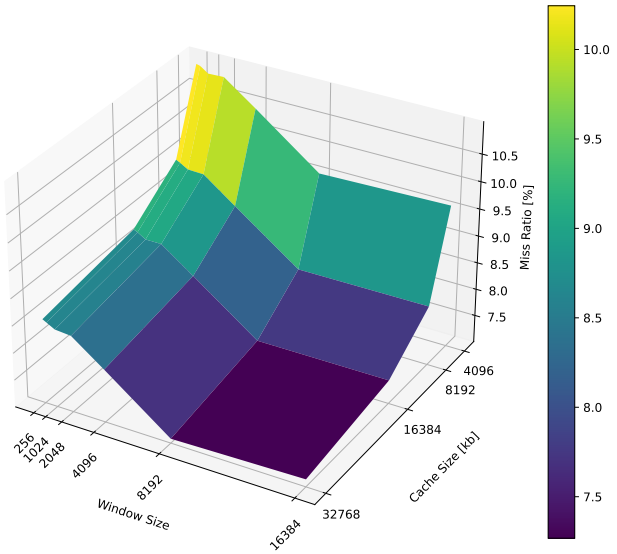
In our experiments, we evaluated the efficacy of the **Fixed Window** approach

Figure 5.11 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with spc-small benchmark (low range of windows).



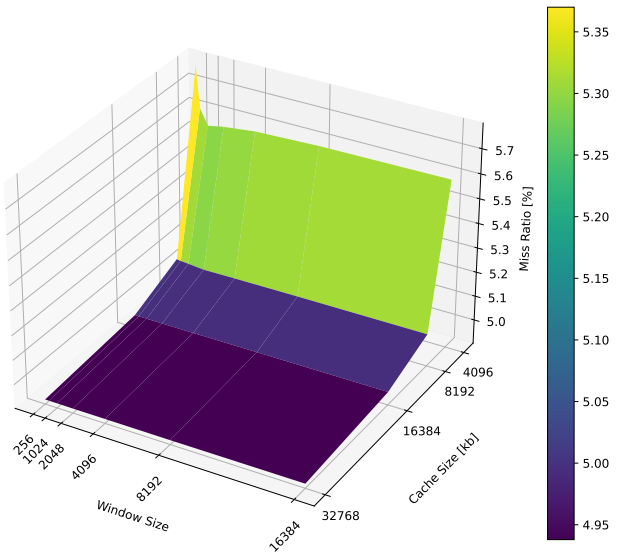
Source: Author (2023).

Figure 5.12 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with spc-small benchmark.



Source: Author (2023).

Figure 5.13 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with liblinear-tsmall benchmark.



Source: Author (2023).

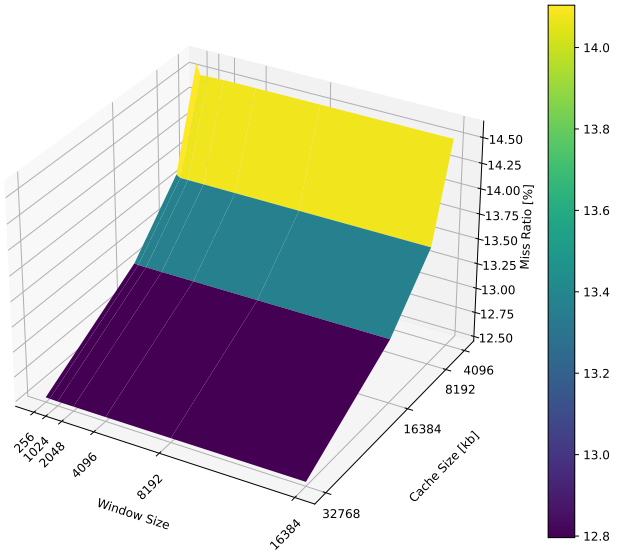
by employing benchmarks with larger working sets, including liblinear-tlarge, kmeans-large, disparity-vga, and pca-large. The outcomes, particularly illustrated in Figure 5.14, indicate that the benefits of the **Fixed Window** strategy diminish when the working sets exceed the cache capacity.

However, this trend was not uniform. Certain scenarios, as highlighted in Figure 5.15, revealed that augmenting the window size could indeed confer modest enhancements. Based on these insights, we adopted a window comprising 4096 memory accesses for subsequent analyses with other strategies. This decision was predicated on the observed incremental improvements and the pragmatic considerations in orchestrating the experiments with varying window dimensions.

5.5 SLIDING WINDOW EVALUATION

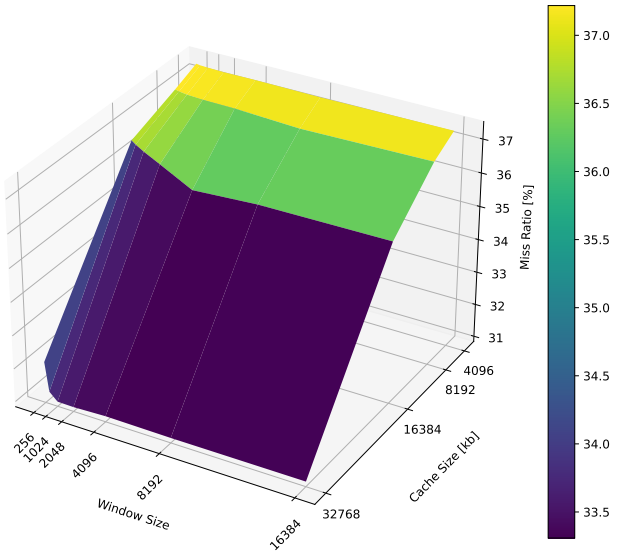
The **Sliding Windows** approach is grounded in two pivotal parameters: the threshold and the window size. The former serves as a determinant for the minimum disparity in miss counts between two eviction policies that must be met prior to initiating a policy switch. For instance, consider a scenario where LRU has a lower miss ratio compared to BIP, FIFO, and RANDOM, being LRU the designated active eviction policy. If, during the program's runtime, BIP's miss ratio reduces, surpassing the others, the transition to mark BIP as the active policy would only be triggered if the disparity between BIP's and LRU's miss counters exceeds the predefined threshold.

Figure 5.14 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with liblinear-tlarge benchmark.



Source: Author (2023).

Figure 5.15 – Miss rate evolution for diverse cache sizes and windows configurations using Fixed Windows approach with pca-large benchmark.



Source: Author (2023).

Throughout our experimentation, we assessed how these parameters modulate the miss ratio across diverse benchmarks. Our goal was to discern the configuration

most apt for the majority of scenarios.

The threshold must inherently be less than the windows size. This is because the tally of misses within the window is stored, setting the upper limit for the possible number of misses to the window size. As a consequence, the threshold's upper bound becomes window size $-\ 1$.

Our findings affirmed that, while the window size's primary role is to constrain the threshold, the threshold itself is the pillar that predominantly influences the performance of the **Sliding approach**. As depicted in Figure 5.16, lower threshold values notably diminish the miss ratio for certain benchmarks (such as pca-small and rbm-small). In contrast, larger thresholds commence underperforming beyond a specific point. On assessing the entirety of our results, we concluded that, on average, a threshold of 512 offers the most optimal outcomes. Opting for an even lower threshold would theoretically ensure a minimal miss ratio. However, it comes with the caveat of inducing frequent policy shifts. Such volatility could inadvertently lead to the eviction of pertinent data. This phenomenon, especially evident during transitions from policies like LRU or BIP to FIFO, was further elaborated in Section 2.6.

Having established one variable in the **Sliding Window** approach, our attention turned to the second parameter: the windows size. Our experimentation with varying this dimension under the *Cortex suite* of benchmarks echoed the findings observed with the **Fixed Windows** strategy. Specifically, we noted a consistent reduction in the miss rate corresponding with increases in cache memory size. Furthermore, within these conditions, the majority of benchmarks exhibited minimal fluctuation in miss rates across different windows sizes, assuming the cache memory size was held constant. This trend is exemplified in Figure 5.17, which demonstrates a uniform miss ratio across various windows sizes when analyzed within the same cache memory parameter.

However, this was not a universal phenomenon. Certain specific benchmarks, as outlined in Figures 5.18 and 5.19, bucked this trend, showing a tangible decrease in miss ratios as the windows sizes expanded, albeit only up to a point beyond which the rates stabilized.

Drawing from these analytical insights, we aligned on a windows size of 4096 for subsequent comparative evaluations with other strategies, mirroring our decision in the **Fixed Window** methodology. Consequently, the optimal parameter configuration identified for the **Sliding Window** approach, harmonizing performance efficiency and operational pragmatism, comprised a threshold of 512 coupled with a windows size of 4096.

5.6 ONLINE SELECTION EVALUATION

Comparable to the **Sliding Windows** approach, the **Online selection** method involves multiple parameters that influence the resultant outcomes. The underlying

concept of this approach is to concurrently execute conventional cache eviction policies, assess their miss ratios, and designate the policy that surpasses others in performance. Although it bears resemblances to the sliding approach, there exist pivotal distinctions.

In the **Online selection** method, a counter is utilized to monitor cache hits and misses. Unlike aging mechanisms used in the **Sliding Windows** approach, this method confines the counter within predefined maximum and minimum thresholds. When comparing the efficacy of different eviction policies, the transition between selected policies is triggered by a threshold applied to their respective counters, thus averting frequent policy alterations.

The two parameters pivotal to this approach are the maximum and minimum counter values, in conjunction with the threshold for policy transition. In our preliminary experiments with the **Online selection** approach, our objective was to ascertain the optimal combination of these parameters across a spectrum of benchmarks. Given its resemblance to the **Sliding Windows** method, we employed the same benchmarks for our investigation.

The graphs depicted in Figure 5.20 illustrate that larger thresholds yield results comparable to or even worse than those obtained with smaller thresholds, a phenomenon that was anticipated. Opting for a larger threshold compels the approach to discern significant disparities in the behavior of two policies within a short span of memory accesses, rendering detection more intricate and thereby reducing the frequency of policy changes. Conversely, smaller threshold values afford finer granularity, enabling the **Online selection** to swiftly switch between policies in response to fluctuations in miss ratio.

In light of these observations, we elected to conduct further experiments with smaller threshold values, which had not been explored in the context of the **Sliding Windows** approach. We incrementally varied the threshold from 1 to 512, doubling its value in each iteration. The results are presented in Figure 5.21. Notably, the miss rate exhibits significant variability across benchmarks and window sizes when the threshold is less than 32, attributable to the frequent randomness of changes during code execution. Our investigations revealed that thresholds around 64 generally yielded optimal performance.

Moreover, during the course of our experiments, we noted that the efficacy of the **Online selection** method is contingent upon the maximum and minimum values of the counter employed for selection, represented as "Window" in the graphs. To explore this dependency further, we plotted the miss rate of benchmarks with the threshold fixed at 64 while varying the window size. This analysis, depicted in Figure 5.22, revealed that employing a maximal counter value of 1024 yielded optimal results in most cases.

This circumstance can be attributed to the saturation of the counter. When cache accesses favor one of the policies, its miss counter may reach the minimum. Even

as memory accesses continue resulting in cache hits, the counter value remains unchanged. Consequently, this constrains the disparity between policies exhibiting poor and superior performance. For instance, with the selected parameters, the worst-case scenario separates the best and worst policies by 1024 accesses. Furthermore, in the event of a shift in memory access patterns, wherein a previously inferior policy becomes more suitable, it would take 512 accesses plus half the threshold for it to become the selected policy. The utilization of larger counter limits disregards policy changes in memory access patterns where rapid policy transitions are necessary, as evidenced in Figure 5.22 with small cache sizes, where larger maximum counter values distinctly underperform.

5.7 SET DUELING EVALUATION

The **Set Dueling** method is designed to handle the complexities of managing multiple eviction policies simultaneously. It accomplishes this by partitioning the cache memory into distinct sets and assigning each set to a particular eviction policy. In our analysis, we have chosen to allocate 32 cache sets per policy, leaving the remaining sets available for the policy that exhibits the most favorable performance. This approach allows us to dynamically adapt to changing access patterns and select the most appropriate eviction policy for each subset of memory.

However, it's important to note that the number of sets allocated to each policy remains constant regardless of the size of the cache memory. This decision was made intentionally to simplify the experimental setup and maintain consistency across different configurations. However, this approach may lead to suboptimal performance, especially in scenarios where the cache memory size is relatively small.

To illustrate, consider a cache memory size of 4 KB in our experimental setup. With a line size of 32 bytes and 4 ways associativity, the cache is divided into a total of 128 sets. When we select the best-performing policy, we allocate 32 sets to it, along with the subsequent 32 sets that follow the same policy. This leaves us with 64 sets that are then allocated to the remaining two traditional eviction policies, which may not necessarily be the most suitable choices for the current memory access patterns. Consequently, in this scenario, only half of the cache memory is effectively optimized, potentially resulting in suboptimal performance.

Table 5.2 provides a detailed breakdown of the number of sets allocated to each policy across various memory configurations, ranging from 4 KB to 32 KB. This allows for a comprehensive understanding of how the allocation of cache sets impacts the overall performance of the **Set Dueling** method across different cache sizes.

Table 5.3 provides an insightful analysis of the percentage of memory that remains unoptimized when employing the **Set Dueling** method across different cache sizes. This metric underscores our observation that the effectiveness of the approach

Table 5.2 – Number of sets depending on cache configuration.

Cache Size	Number of Ways	Line Size		
		16 bytes/line	32 bytes/line	64 bytes/line
4 KB	4	256 sets	128 sets	64 sets
	8	128 sets	64 sets	32 sets
	16	64 sets	32 sets	16 sets
8 KB	4	512 sets	256 sets	128 sets
	8	256 sets	128 sets	64 sets
	16	128 sets	64 sets	32 sets
16 KB	4	1024 sets	512 sets	256 sets
	8	512 sets	256 sets	128 sets
	16	256 sets	128 sets	64 sets
32 KB	4	2048 sets	1024 sets	512 sets
	8	1024 sets	512 sets	256 sets
	16	512 sets	256 sets	128 sets

tends to improve with larger cache sizes.

Table 5.3 – Suboptimal percentage of memory, considering 32 sets per policy.

Cache Size	Number of Ways	Percentage of suboptimal sets		
		16 bytes/line	32 bytes/line	64 bytes/line
4 KB	4	25%	50%	-%
	8	50%	-%	-%
	16	-%	-%	-%
8 KB	4	12.5%	25%	50%
	8	25%	50%	-%
	16	50%	-%	-%
16 KB	4	6.25%	12.5%	25%
	8	12.5%	25%	50%
	16	25%	50%	-%
32 KB	4	3.125%	6.25%	12.5%
	8	6.25%	12.5%	25%
	16	12.5%	25%	50%

As cache sizes increase, the proportion of memory allocated to suboptimal policies decreases. This trend can be attributed to the greater flexibility afforded by larger cache sizes, allowing for more sets to be allocated to the most effective eviction policies. Consequently, with a larger percentage of memory dedicated to optimal policies, the overall performance of the **Set Dueling** method tends to improve.

Conversely, in scenarios where cache sizes are smaller, a larger proportion of memory may be allocated to suboptimal policies due to the limited number of available cache sets. This can result in decreased performance efficiency as the method may struggle to adapt to varying access patterns effectively.

Figure 5.23 delves into the influence of the number of sets on the miss rate obtained with the same benchmark, shedding light on how this factor impacts the performance of the **Set Dueling** method across different cache line sizes.

One notable observation from the Figure 5.23 is that implementing the **Set**

Dueling method for a cache line size of 64 bytes poses challenges due to the insufficient number of sets available. As a result, the method cannot be adequately implemented, leading to random performance outcomes when compared with a cache line size of 32 bytes. However, for cache sizes of 8 KB, 16 KB, and 32 KB, the method is appropriately implemented, allowing for a more accurate determination of the best policy.

When examining specific benchmarks such as *kmeans-small*, we notice a reversal in results between different cache line sizes, particularly evident in the 4 KB cache. This highlights the sensitivity of the method to changes in cache configuration and the intricate interplay between cache line size, number of sets, and benchmark characteristics.

Similarly, in the case of *pca-small* and *svd3-small* benchmarks, we observe instances where the **Set Dueling** method outperforms the expected performance based solely on cache line size. This phenomenon underscores the ability of the method to leverage a larger number of sets to overcome limitations posed by cache line size and achieve superior performance outcomes.

In the **Set Dueling** method, the window size stands as the sole parameter open for adjustment. This parameter delineates the number of preceding memory accesses utilized to ascertain the policy exhibiting superior performance. We anticipate that altering the window size will yield varying results contingent upon the benchmark utilized. Specifically, we expect the method to yield optimal outcomes when the window size is smaller than the size of the memory access pattern best suited to a particular policy.

Consider, for instance, two consecutive memory access patterns, each comprising 300 memory accesses, favoring BIP and LRU. If we set the window size to 300, the method would transition from the previously selected policy to BIP upon reaching the 151st memory access within the BIP memory area. It would continue selecting BIP as the optimal policy until reaching the 151st memory access within the LRU memory area. In essence, it would accurately identify the correct policy for only 149 memory accesses. Consequently, reducing the window size might not necessarily enhance performance. This is because a minimum number of accesses is required to discern a memory access pattern, and this requisite varies for each pattern. Opting for a smaller window could lead to unnecessary policy changes.

Another crucial aspect to consider in this approach is the random allocation of fixed policy sets and subsequent sets. In instances where the window size is small and the pattern exclusively utilizes sets associated with a non-optimal policy, the method may fail to identify the best policy accurately.

Determining the optimal parameter that strikes the best balance between its value and resulting performance necessitates an empirical approach. In our case, we opted to conduct extensive experimentation by employing the **Set Dueling** method across a spectrum of benchmarks. We systematically varied the window size across

different memory sizes and compared the resulting miss ratios to discern the most effective configuration.

Following this rigorous experimentation, we discovered that a window size of 4096 consistently yielded favorable results across the majority of benchmarks, irrespective of the cache size being tested. This window size demonstrated robust performance characteristics, effectively capturing and adapting to diverse memory access patterns present in the benchmarks.

Figure 5.24 illustrates a notable observation regarding the impact of window size on the performance of the **Set Dueling** method, particularly in the context of varying memory working set. This observation underscores the dynamic nature of the relationship between window size and performance across different benchmarks and memory sizes.

Specifically, we observed a distinct pattern in the performance of the kmeans benchmark across different memory sizes and window sizes. For a memory size of 4096 bytes, the *kmeans* benchmark exhibited superior performance with smaller window sizes when utilizing a smaller memory working set (*kmeans-small*). Conversely, for larger memory working sets (*kmeans-medium*), the performance improved with larger window sizes, up to a window size of 16384.

Figure 5.25 shows another intriguing aspect of the **Set Dueling** method's performance variability concerning window size, particularly in the context of the *lda* benchmark across various memory footprints.

When examining the *lda* benchmark with a cache size of 32 KB and three different memory working sets (small, medium, and large), we made a noteworthy observation. Specifically, for the medium memory working set case, we observed that changes in window size resulted in discernible performance improvements. However, this effect was not as pronounced or clearly detected for the other memory working sets.

These divergences in optimal window size highlights the nuanced interplay between memory access patterns, memory working sets, and the effectiveness of the **Set Dueling** method. It underscores the importance of adaptability and flexibility in selecting an appropriate window size to suit the specific characteristics of each benchmark and memory configuration.

5.8 COMPARATIVE ANALYSIS

Once we have conducted experiments testing various parameter values for the proposed methods, the next step involves comparing them with traditional policies. The overarching goal is to showcase how these approaches enhance cache memory performance by adaptively selecting the traditional policy that best aligns with the memory access pattern of the benchmark under scrutiny.

To facilitate a comprehensive comparison, we executed the same set of bench-

marks, specifically the *Cortex suite* benchmarks, across all presented policies. Subsequently, we utilized the results obtained from the **LCU** policy as a baseline for normalization purposes. This standardized baseline allows for a fair and meaningful comparison of the performance improvements achieved by the proposed methods relative to traditional policies.

By leveraging a common set of benchmarks and a normalized baseline, we can effectively evaluate and quantify the efficacy of the proposed approaches in enhancing cache memory performance. This comparative analysis enables us to discern the extent to which adaptive policy selection strategies contribute to improved cache management and overall system efficiency across diverse workloads.

The parameters utilized in this comparative analysis for each method are succinctly summarized in Table 5.4. This tabular representation provides a comprehensive overview of the specific configurations employed for each method, facilitating a clear understanding of the experimental setup.

Table 5.4 – Parameters used in every method for comparison with traditional policies

Method	Windows size	Threshold	Min. Cache size	Max. Cache size
Fixed Window	4096	-	4KB	32KB
Sliding Window	4096	512	4KB	32KB
Online selection	1024	64	4KB	32KB
Set Dueling	4096	-	4KB	32KB

The figures in the Annex A provide a comparative analysis of the miss ratio normalized with respect to the LRU policy for both traditional eviction policies and the proposed approaches for combining traditional policies based on the memory access pattern.

Upon examining figures in Annex A, which compare **LRU** with **Random**, **FIFO**, and **BIP**, respectively, it becomes apparent that **LRU** does not consistently outperform other policies. Notably, the **Random** policy generally exhibits higher miss ratios compared to **LRU**, although for the `kmeans-large` benchmark, it reduces the number of misses for cache sizes of 4 KB and 8 KB. This anomaly aside, **Random** typically results in a greater number of misses compared to **LRU** across most scenarios.

Similarly, while the **FIFO** policy generally performs worse than **LRU**, there are instances, such as the `me-large` benchmark for a 4 KB cache and the `me-medium` benchmark for an 8 KB cache, where **FIFO** outperforms **LRU**. On the other hand, the **BIP** policy demonstrates more cases where it performs better than **LRU**. Noteworthy instances include the `kmeans-medium` and `kmeans-small` benchmarks for 4 KB memories, `lda-large`, `lda-medium`, and `lda-small` for memories of 8 KB, 16 KB, and 32 KB, as well as `pca-medium` and `pca-small` for 32 KB caches.

These results suggest that the proposed approaches have the potential to improve the miss ratio of traditional policies by effectively detecting differences in perfor-

mance among different policies for the same benchmark. Consequently, the correct selection of policy has the potential to reduce the number of misses, thereby enhancing cache memory performance overall.

In figures in Annex A, for Online, Set Dueling, Sliding Window and Fix Window, we present the results obtained with the proposed approaches. Overall, all the proposed approaches exhibit lower miss rates than **LRU**, indicating an improvement in cache memory performance. However, there are some specific benchmarks where the miss ratio is worse than that of **LRU**.

Figures in the Annex A for the Online approach demonstrate how the **Online** approach effectively reduces the miss ratio of **LRU** across all studied cache sizes for several benchmarks. Notably, there are cases for cache sizes of 32 KB where the improvement compared with **LRU** exceeds 20%, this could be observed in Figure A.12. This improvement is primarily attributed to the method's capability to switch between **LRU** and **BIP** dynamically. For instance, the benchmarks `pca-large`, `pca-medium`, and `pca-small` exhibit miss ratios slightly better than those obtained with the traditional **BIP** policy. Particularly intriguing is the case of the benchmark `liblinear-medium` for a 32 KB cache, where the **Online** method showcases around a 20% improvement compared with **LRU**. Interestingly, this improvement is not observed when using the **BIP** policy alone. This indicates that the method doesn't merely switch between policies throughout the entire execution of the benchmark but adapts dynamically, leading to improvements in both **LRU** and **BIP** performance over the course of program execution.

These findings underscore the efficacy of the **Online** approach in enhancing cache memory performance by dynamically selecting the most suitable policy based on the prevailing memory access patterns. The ability to switch between policies at different stages of program execution enables the method to adapt effectively to varying workload characteristics, ultimately leading to improved cache utilization and reduced miss rates.

The cases where the approach performs worse than **LRU** are located in caches of 4 KB for the benchmarks `liblinear-medium` and `liblinear-small`. Curiously, this combination results in miss ratios higher than those obtained with **LRU** for all the proposed approaches. The **BIP** policy also encounters issues with the `liblinear-large` benchmark for the same cache size. This could be explained by premature changes to other policies with the approaches. If this occurs, cache addresses could be replaced following the newly selected policy, but these addresses would not be replaced if **LRU** were used. A specific selection of windows and threshold parameters for the approaches could potentially resolve the issues observed with the mentioned benchmarks.

For the **Set Dueling** approach with 3 policies, we concluded that the results are similar to those obtained with the **Online** method. However, for a cache size of 4 KB, the

method performs worse than **Online** and, in several cases, even worse than **LRU**. This discrepancy can be attributed to the number of sets used for policies that do not perform well enough to be selected. Further details on this aspect can be found in section 5.7, where we elaborate on how the cache size and cache line size impact the expected results of this approach. Consequently, this method is not recommended for small cache sizes, as it generally performs worse than **Online** despite having significantly lower hardware overhead in its implementation. Thus, it may be considered a better online method due to its reduced hardware complexity.

When compared with **BIP**, which is also a viable alternative to **LRU** in certain benchmarks, we observe that **Set Dueling** is capable of mitigating some cases where **BIP** performs significantly worse than **LRU**. For instance, in the benchmarks `rbm-large` and `rbm-medium` with a cache size of 16 KB, where **BIP** performs 20% worse than **LRU**, **Set Dueling** outperforms both traditional policies.

For the offline methods, **Sliding Window** and **Fixed Window**, our findings revealed that the fixed version performs significantly better across the studied benchmarks. It demonstrates the capability to reduce the miss rate of **LRU** by up to 40% in certain cases, such as `me-medium` for a cache size of 8 KB, `me-large` for 16 KB caches, or `pca-medium` for caches with 32 KB. This could be observed in Figure 5.27. Even though **Sliding Window** occasionally yields better results than **LRU**, its improvement is not comparable to the enhancements achieved by the fixed version of the Window method. Moreover, the results of **Sliding Window** are comparable with those of online approaches, rendering it a less favorable option due to the time required to execute it before analyzing the software on real hardware.

The proposed experiments and their results highlight a significant disparity between the utilization of a single eviction policy and the optimal approach, wherein the policy is dynamically switched based on the heuristic policy with the best result for each data access. The approaches outlined in this work consistently yield superior results compared to traditional policies across the majority of cases. In section 5.9, we provided detailed insights into various factors that should be taken into account to achieve more accurate outcomes, such as the effects of inputs and loops in the code. Although these aspects were not the primary focus of the experiments, they represent potential areas for exploration in future research endeavors.

5.9 DISCUSSION

When implementing dynamic cache eviction policies, several factors come into play that can significantly impact both performance and resource utilization. These include, but are not limited to, the nature of program inputs, the complexity introduced by loop structures, the additional cache space required, and the overhead associated with transitioning between policies. In this section, we delve into each of these aspects

to provide a comprehensive understanding of the challenges and considerations.

Impact of Program Inputs

In our experimental evaluation, we found that different input sets for the same benchmark programs could lead to varying performance metrics when using dynamic eviction policies. For instance, our experiments indicated that the `PCA` benchmark exhibited up to an 20% difference in miss ratio when run with medium-sized inputs compared to small-sized inputs. The development of a specialized tool to systematically compare these outcomes is earmarked for future work. This suggests that dynamic policy selection can be influenced by the specific computational and memory access patterns presented by different inputs.

Loop Complexity

Changing eviction policies within loop structures presents a unique set of challenges. While our framework does indicate the specific iteration at which a policy change occurs, realizing this in hardware would necessitate additional counters within the CPU architecture. One potential solution could be to treat loops as single computational blocks and determine the most efficient eviction policy for the entirety of the loop, thereby simplifying the hardware requirements.

Cache Space Considerations

Implementing multiple eviction policies inevitably increases the hardware footprint, particularly in terms of required cache space. However, policies like **LRU**, **FIFO**, and **BIP**, which maintain meta-information about recently accessed cache lines, could potentially share metadata structures, thereby minimizing the space overhead. In contrast, the **RANDOM** policy does not require any metadata storage. Additionally, cache duplication for each eviction policy, while conceptually straightforward, is not resource-efficient. Techniques like Set Dueling, as proposed in (QURESHI et al., 2007), can significantly reduce this overhead. This technique allocates a small number of tag sets to each candidate policy, and the policy that performs best on its assigned sets becomes the global policy for all the other sets.

Overhead in Policy Transitions

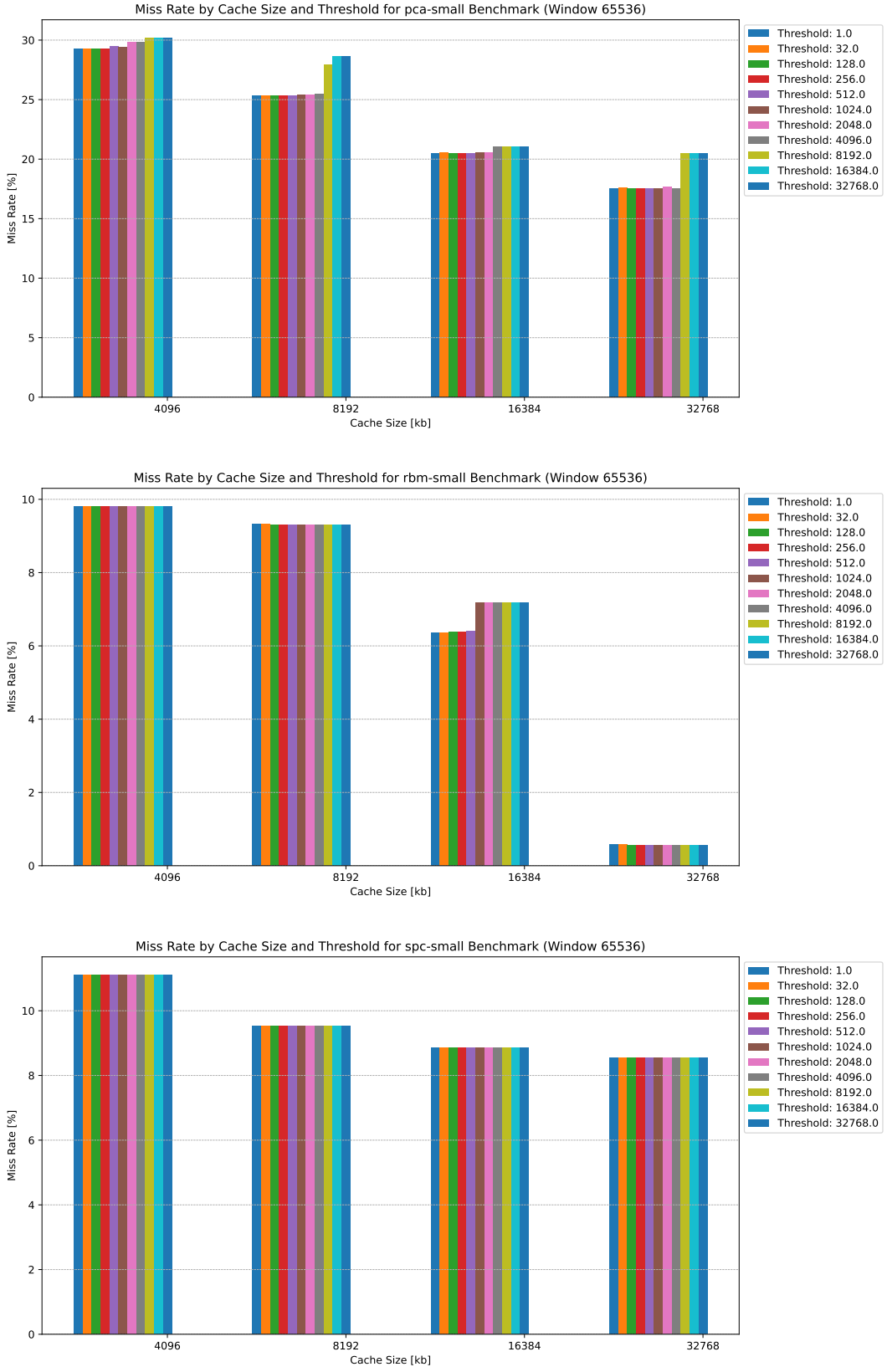
A further consideration is the overhead incurred during policy transitions. In our framework, transitioning from one policy to another entails copying over the metadata and cache state. For example, switching from **FIFO** to **LRU** would involve transferring all **FIFO** states and metadata to the **LRU** cache, this was explained in 2.6. While this approach simplifies the switching process, it imposes a non-negligible CPU overhead. For real hardware implementations, this overhead could be mitigated either by flushing the cache and metadata with each switch or by maintaining the existing cache state and metadata and merely updating the policy enforcement mechanism.

Simulations in offline methods The time required to execute the proposed offline approaches poses a significant challenge for certain benchmarks. We observed

that benchmarks like `liblinear-large` or `lda-large` demanded more than 6 hours for analysis by the **Fixed Window** approach under certain configurations. Particularly when the window size was small, the frequency of copying the entire cache memory to simulate the next window became excessively high. This operation imposes a significant computational burden on the CPU responsible for executing the simulation. One potential solution to address this lengthy processing time would be to adjust the size of the windows or to selectively apply our framework to specific portions of the codebase.

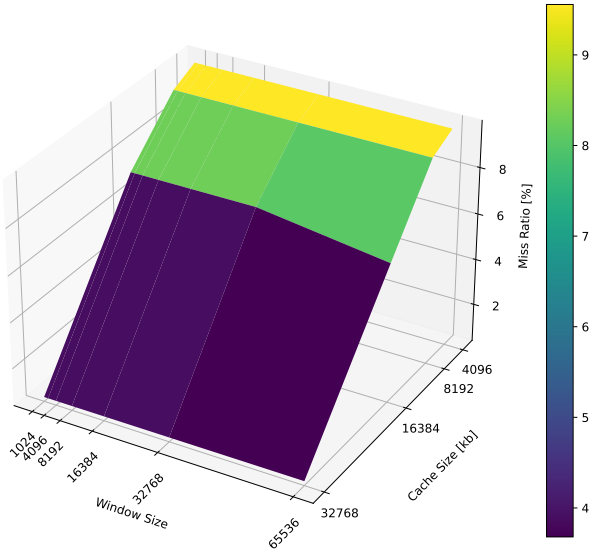
Selection of Right Parameters The results have demonstrated that obtaining a universal set of parameters applicable to every method across all benchmarks is not a straightforward task. This implies that even with real-world software, it is necessary to utilize the framework to determine the optimal configuration. However, this process may also present challenges due to the significant time required to execute the simulations.

Figure 5.16 – Miss rate evolution for diverse cache sizes and threshold configurations using Sliding Windows approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows.



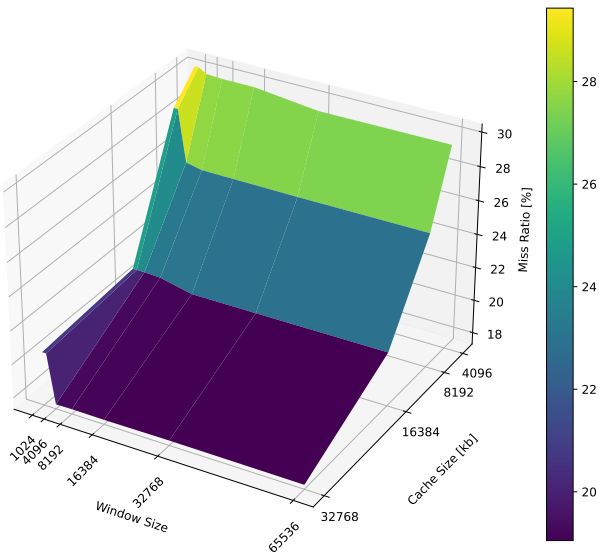
Source: Author (2023).

Figure 5.17 – Miss rate evolution for several cache sizes and windows configurations using Sliding Windows approach with rbm-small benchmark.



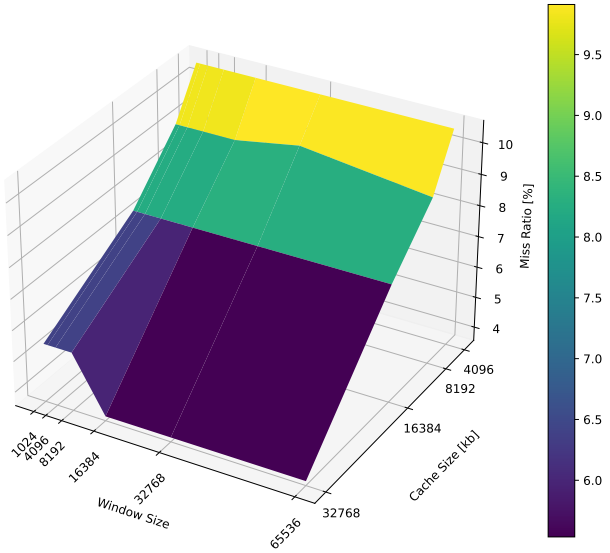
Source: Author (2023).

Figure 5.18 – Miss rate evolution for several cache sizes and windows configurations using Sliding Windows approach with pca-small benchmark.



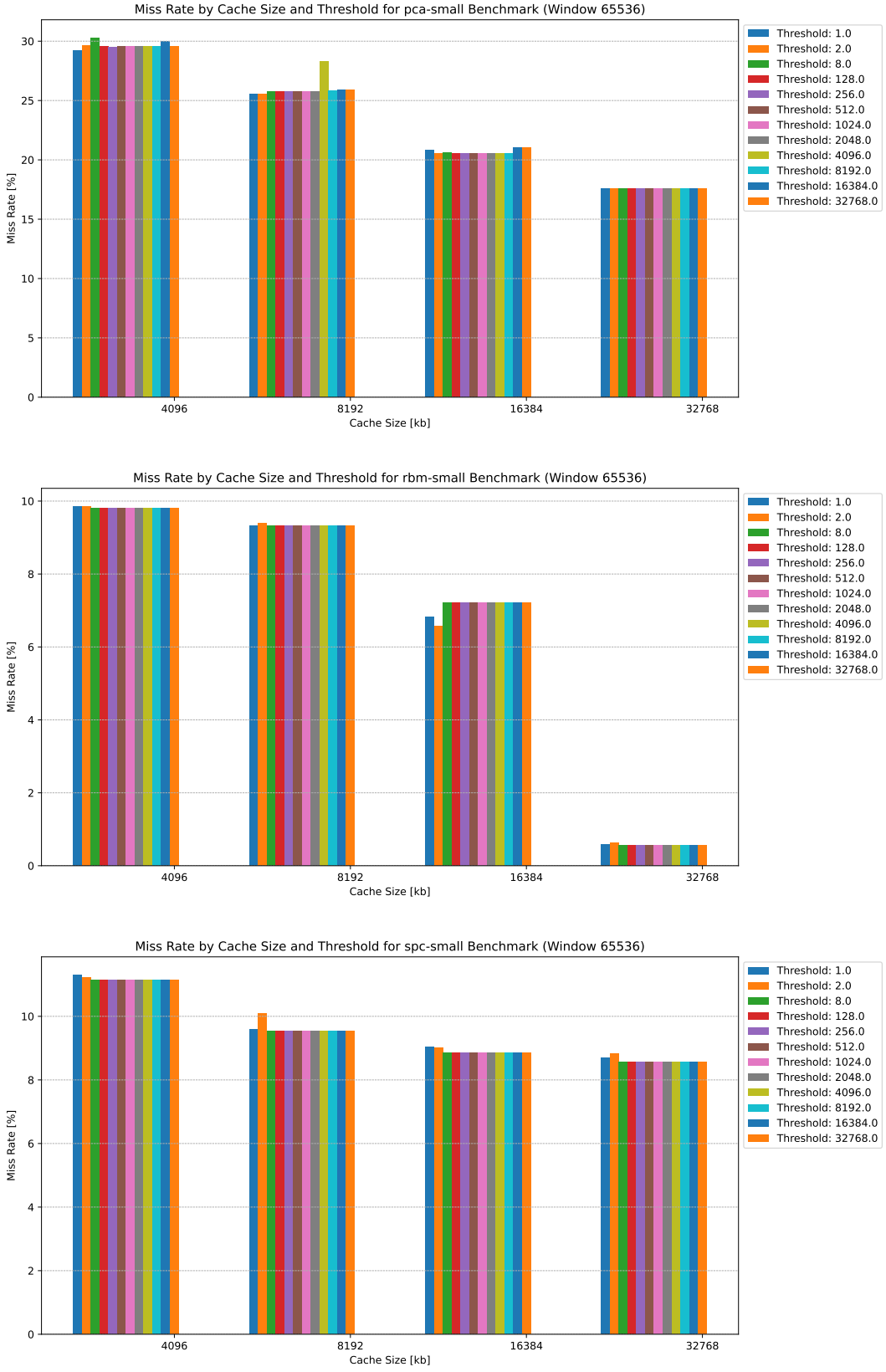
Source: Author (2023).

Figure 5.19 – Miss rate evolution for diverse cache sizes and windows configurations using Sliding Windows approach with srr-small benchmark.



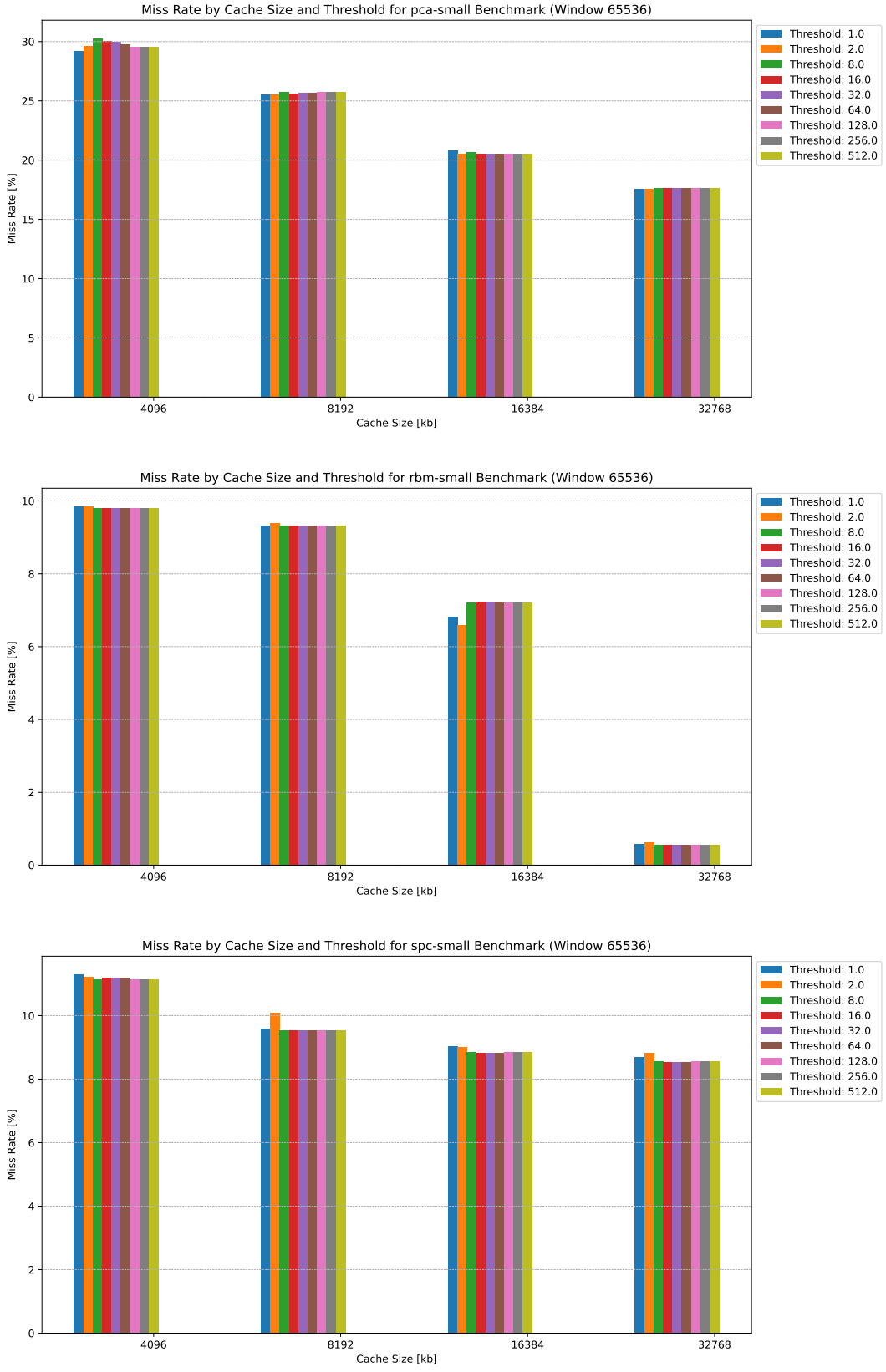
Source: Author (2023).

Figure 5.20 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows.



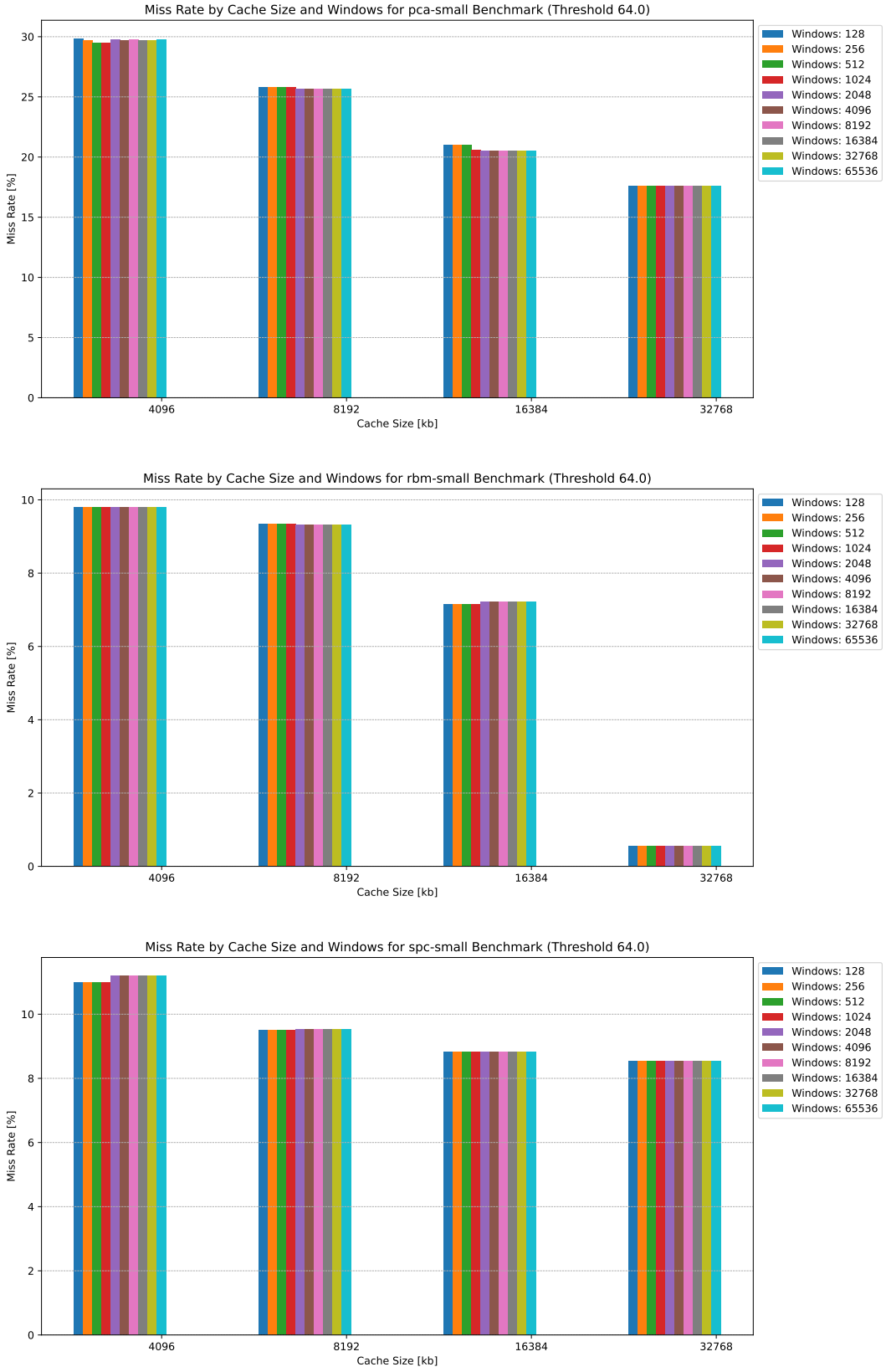
Source: Author (2024).

Figure 5.21 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows, focusing in small values of threshold.



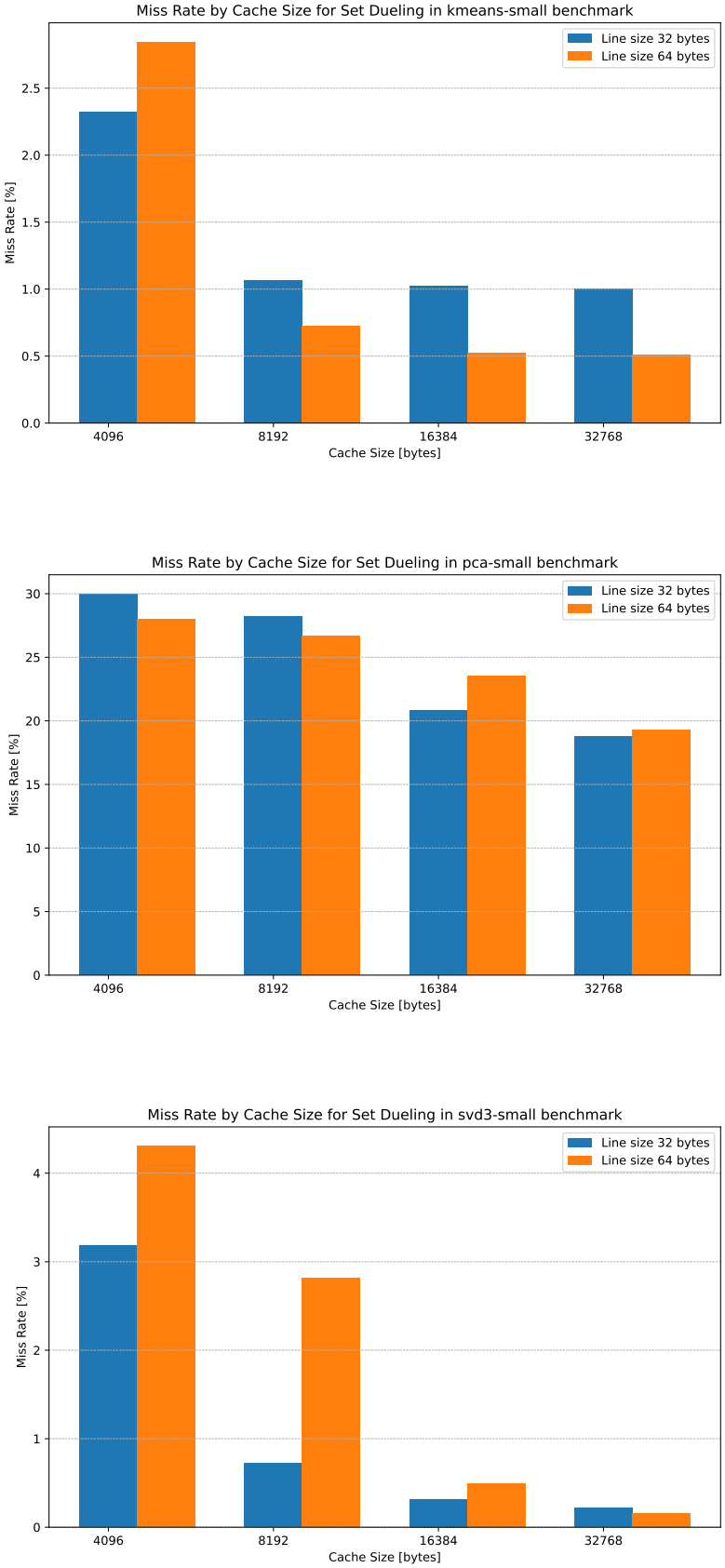
Source: Author (2024).

Figure 5.22 – Miss rate evolution for diverse cache sizes and threshold configurations using Online selection approach with pca-small, rbm-small, spc-small benchmarks with a fixed size of windows, focusing in small values of threshold.



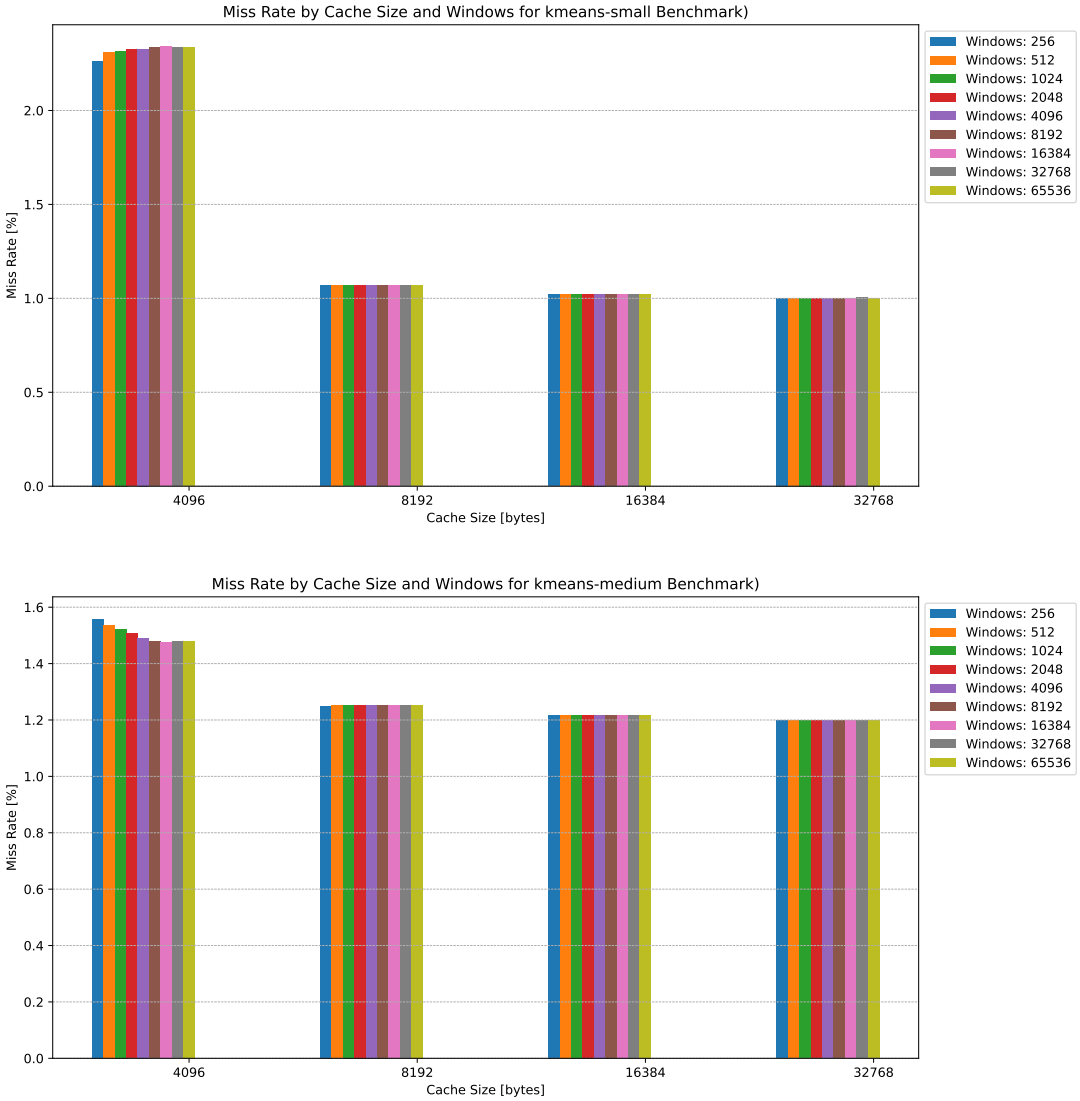
Source: Author (2024).

Figure 5.23 – Miss rate comparison for Set Dueling approach when using different memory line sizes (32 and 64 bytes).



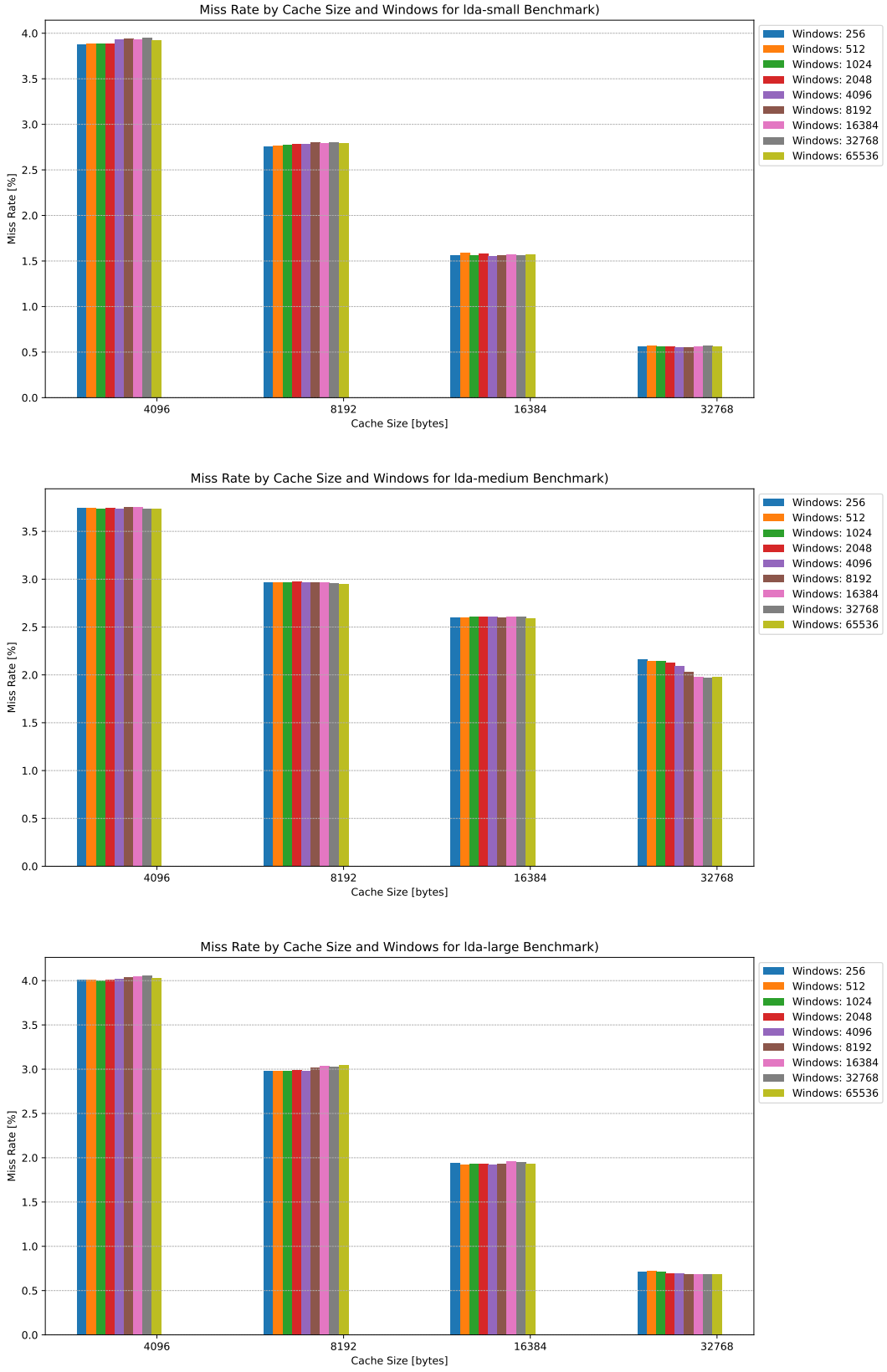
Source: Author (2024).

Figure 5.24 – Miss rate evolution for diverse cache sizes configurations using Set Dueling approach with kmeans benchmarks with a diverse size of windows.



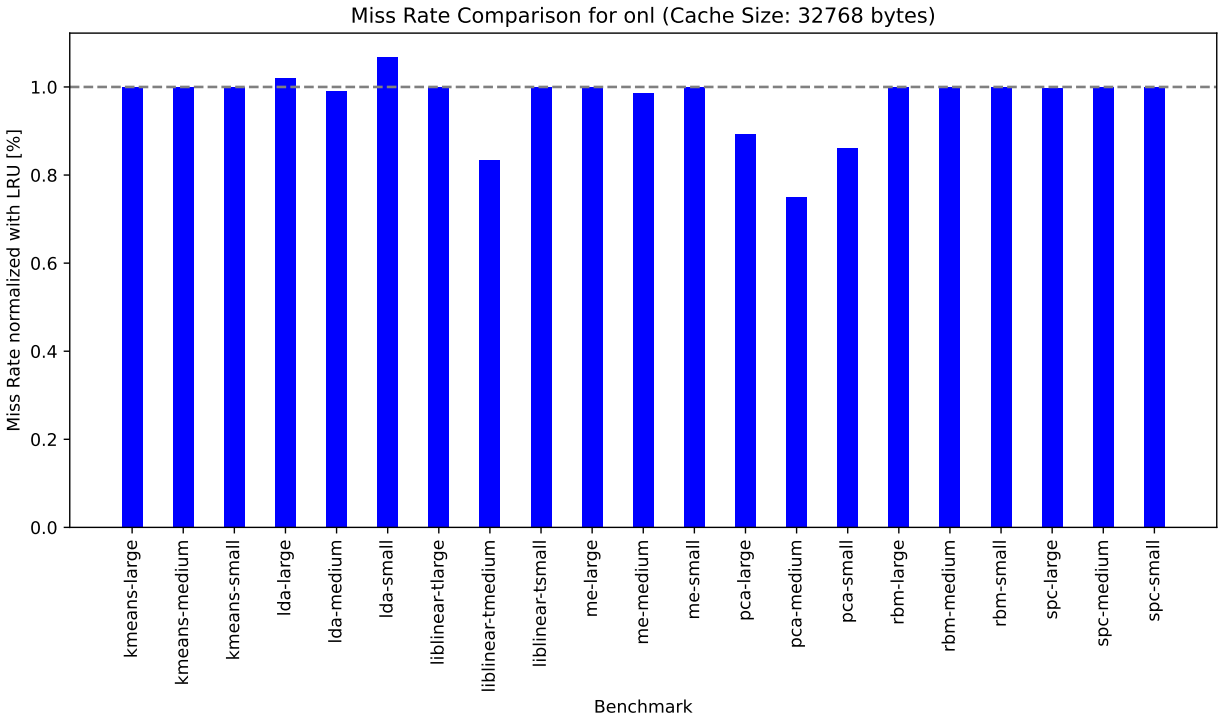
Source: Author (2024).

Figure 5.25 – Miss rate evolution for diverse cache sizes configurations using Set Dueling approach with lda benchmarks with a diverse size of windows.



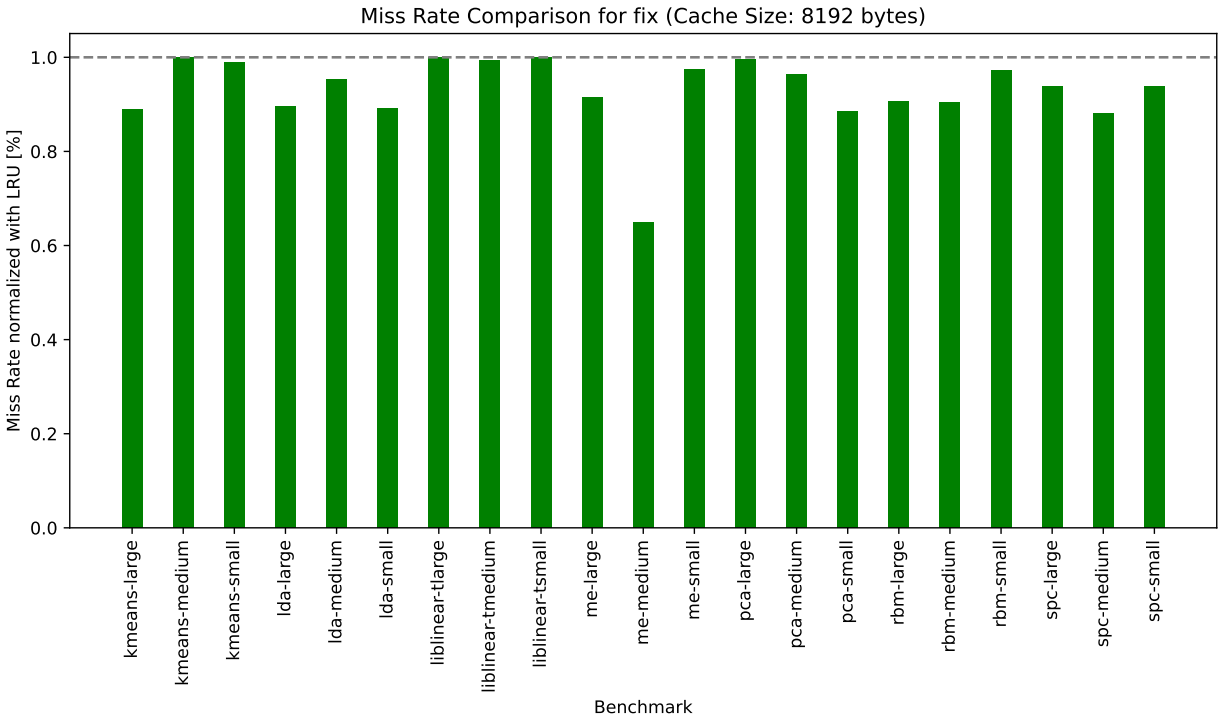
Source: Author (2024).

Figure 5.26 – Miss rate evolution using Online with Cortex benchmarks (Cache size 32768 bytes).



Source: Author (2024).

Figure 5.27 – Miss rate evolution using Fix Window with Cortex benchmarks (Cache size 8192 bytes).



Source: Author (2024).

6 CONCLUSION

The framework developed using the *Cachegrind* tool has proven to be robust and versatile, effectively facilitating the testing of code using all proposed methods, both online and offline. By leveraging this framework, researchers and developers can gain valuable insights into how different cache replacement policies impact code performance, enabling informed decisions on policy selection for specific code segments.

The alternative methods to traditional cache replacement policies proposed, have demonstrated significant promise in reducing the total number of cache misses across various benchmarks, mainly the Fixed-Window and Set Dueling. By considering the unique characteristics of each code segment and dynamically selecting the most suitable policy, these methods have showcased their ability to enhance cache efficiency and optimize code execution.

In particular, the online methods, **Set Dueling** and **Online Selection**, have shown exceptional potential for real-world implementation. Through comprehensive simulations, these methods have consistently outperformed **LRU**, indicating their viability for improving cache performance in practical hardware environments. The framework's ability to assess the potential benefits of these methods, even considering the associated hardware overhead, underscores its utility in guiding policy selection decisions.

Similarly, the offline methods have yielded substantial improvements in cache performance, surpassing 40% reduction in cache misses compared to **LRU** in certain scenarios. This highlights the importance of considering dynamic policy selection strategies, as opposed to relying solely on static policies, to effectively manage cache resources and optimize code execution.

6.1 IMPLICATIONS AND RECOMMENDATIONS

Despite the promising results, it is important to acknowledge certain limitations and considerations. The time-intensive nature of simulating offline methods for certain configurations poses a challenge, suggesting the need for optimization strategies to streamline simulation processes. Additionally, the fixed input nature of offline simulations may not fully capture the dynamic nature of real-world software execution, emphasizing the importance of further research into adaptive offline methods capable of responding to varying input data.

Furthermore, while some parameter configurations may exhibit universality across benchmarks, our findings suggest the necessity of fine-tuning parameters to optimize performance for specific code segments. Future research efforts could focus on developing more sophisticated parameter selection strategies tailored to individual benchmarks, thereby maximizing the effectiveness of the proposed methods in diverse computing environments.

6.2 LIMITATIONS AND FUTURE WORK

In conclusion, the findings of this study underscore the importance of dynamic cache management strategies in optimizing code performance. By leveraging a combination of online and offline methods, developers can achieve significant improvements in cache efficiency, ultimately enhancing overall system performance. However, ongoing research is needed to address existing limitations and further refine these methods for broader applicability across diverse computing contexts.

As future works we have identified:

- Integrating the generated framework into a more comprehensive framework that allows for a broader range of operations on the studied codes. For example:
  - Implementing functionality to detect memory access patterns and associate them with the most effective method, storing this information in a database for easier detection in future codes.
  - Developing a subfunctionality to analyze loops in the code and determine the method that performs best for them.
  - Enhancing the framework to simulate diverse inputs in the code, enabling more comprehensive testing and evaluation.
- Exploring additional factors and considerations that may influence cache management and code performance, such as cache hierarchies, multi-threading, and varying hardware architectures. By expanding the scope of analysis, researchers can gain deeper insights into the intricacies of cache behavior and develop more sophisticated optimization techniques.
- Conducting empirical studies to validate the effectiveness and scalability of the proposed methods in real-world computing environments. Collaborations with industry partners and deployment in production systems can provide valuable feedback and practical insights for further refinement and optimization.

These future research directions hold the potential to further advance the field of cache optimization and contribute to the development of more efficient and resilient computing systems. By addressing these challenges and embracing opportunities for innovation, researchers can continue to drive improvements in code performance and system efficiency.

BIBLIOGRAPHY

AJORPAZ, S. M. et al. Exploring predictive replacement policies for instruction cache and branch target buffer. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2018. p. 519–532. ISBN 9781538659847. ISSN 10636897.

AKANKSHA, J.; CALVIN, L. *Cache replacement policies*. [S.l.]: Morgan & Claypool Publishers, 2019. ISBN 1681735776, 9781681735771.

ARAUJO, B. A. et al. Implementation and evaluation of adaptive cache insertion policies for real-time systems. In: *Proc. of the XI SBESC*. [S.l.: s.n.], 2021. p. 1–8.

ARRIBAS, S. et al. Development of embedded software for data acquisition and calibration of automotive ecus. In: *29th International Symposium of Automotive Engineering (SIMEA 2022)*. São Paulo: [s.n.], 2022.

BANSAL, A. et al. Evaluating the memory subsystem of a configurable heterogeneous MPSoC. In: *Proc. of the 14th OSPERT*. [S.l.: s.n.], 2018. p. 55–60.

BEDRETCHUK, J. P. et al. Low-cost data acquisition system for automotive electronic control units. *Sensors*, v. 23, n. 4, 2023. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/23/4/2319>.

BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, v. 5, n. 2, p. 78–101, 1966.

GARCíA, S. A. et al. Improving the execution time of industrial applications through planned cache eviction policy selection. In: *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*. [S.l.: s.n.], 2023. p. 1–6.

GRACIOLI, G. et al. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, ACM, v. 48, n. 2, 2015.

HAQUE, M. S.; EASWARAN, A. Predictability and performance aware replacement policy pvisam for unified shared caches in real-time multicores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 37, n. 11, p. 2720–2731, 2018.

HECKMANN, R. et al. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, v. 91, n. 7, p. 1038–1054, 2003.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X.

JACOB, B.; WANG, D.; NG, S. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. ISBN 9780080553849. Disponível em: <https://books.google.com.br/books?id=SrP3aWed-esC>.

JAIN, A.; LIN, C. Back to the future: Leveraging belady's algorithm for improved cache replacement. In: *Proc. of 43rd ISCA*. [S.l.: s.n.], 2016. p. 78–89.

JALEEL, A. et al. High performance cache replacement using re-reference interval prediction (rrip). *Proceedings - International Symposium on Computer Architecture*, p. 60–71, 2010. ISSN 10636897.

KUMAR, S.; SINGH, P. K. An overview of modern cache memory and performance analysis of replacement policies. In: *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. [S.l.: s.n.], 2016. p. 210–214.

LEE, D. et al. Implementation and performance evaluation of the lrfu replacement policy. In: *Proceedings 23rd Euromicro Conference New Frontiers of Information Technology - Short Contributions* -. [S.l.: s.n.], 1997. p. 106–111.

MANCUSO, R.; YUN, H.; PUAUT, I. Impact of DM-LRU on WCET: A Static Analysis Approach. In: *Proc. of the 31st ECRTS*. Dagstuhl, Germany: [s.n.], 2019. v. 133, p. 17:1–17:25. ISBN 978-3-95977-110-8. ISSN 1868-8969.

MEGIDDO, N.; MODHA, D. S. ARC: A Self-Tuning, low overhead replacement cache. In: *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, 2003. Disponível em: <https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: The hardware/software interface*. [S.l.]: Morgan Kaufmann, 2017.

QURESHI, M. K. et al. Adaptive insertion policies for high performance caching. In: *Proc. of the 34th ISCA*. [S.l.: s.n.], 2007. p. 381–391.

REINEKE, J.; GRUND, D. Sensitivity of cache replacement policies. *Transactions on Embedded Computing Systems*, v. 12, 3 2013. ISSN 15399087.

REINEKE, J. et al. Timing predictability of cache replacement policies. *Real-Time Syst.*, Kluwer Academic Publishers, USA, v. 37, n. 2, p. 99–122, nov. 2007. ISSN 0922-6443.

SEGARRA, J.; Gran Tejero, R.; VIñALS, V. A generic framework to integrate data caches in the wcet analysis of real-time systems. *Journal of Systems Architecture*, v. 120, p. 102304, 2021. ISSN 1383-7621.

SEWARD, J.; NETHERCOTE, N.; WEIDENDORFER, J. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. [S.l.]: Network Theory Ltd., 2008. ISBN 0954612051.

SHI, Z. et al. Applying deep learning to the cache replacement problem. In: *Proc. of the 52nd MICRO*. [S.l.: s.n.], 2019. p. 413–425.

SRIVATSA, A. et al. Hyve: A hybrid voting-based eviction policy for caches. In: *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*. [S.l.: s.n.], 2020. p. 1–7.

SUN, B. et al. Minimizing cache usage for real-time systems. In: *Proceedings of the 31st International Conference on Real-Time Networks and Systems*. New York, NY, USA: Association for Computing Machinery, 2023. (RTNS '23), p. 200–211. ISBN 9781450399838. Disponível em: <https://doi.org/10.1145/3575757.3593651>.

THOMAS, S. et al. CortexSuite: A Synthetic Brain Benchmark Suite. In: *Proc. of the IISWC*. [S.l.: s.n.], 2014.

TOUZEAU, V. et al. Fast and exact analysis for lru caches. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 3, n. POPL, jan 2019.

VALSAN, P. K.; YUN, H.; FARSHCHI, F. Taming non-blocking caches to improve isolation in multicore real-time systems. In: *2016 IEEE RTAS*. [S.l.: s.n.], 2016. p. 1–12.

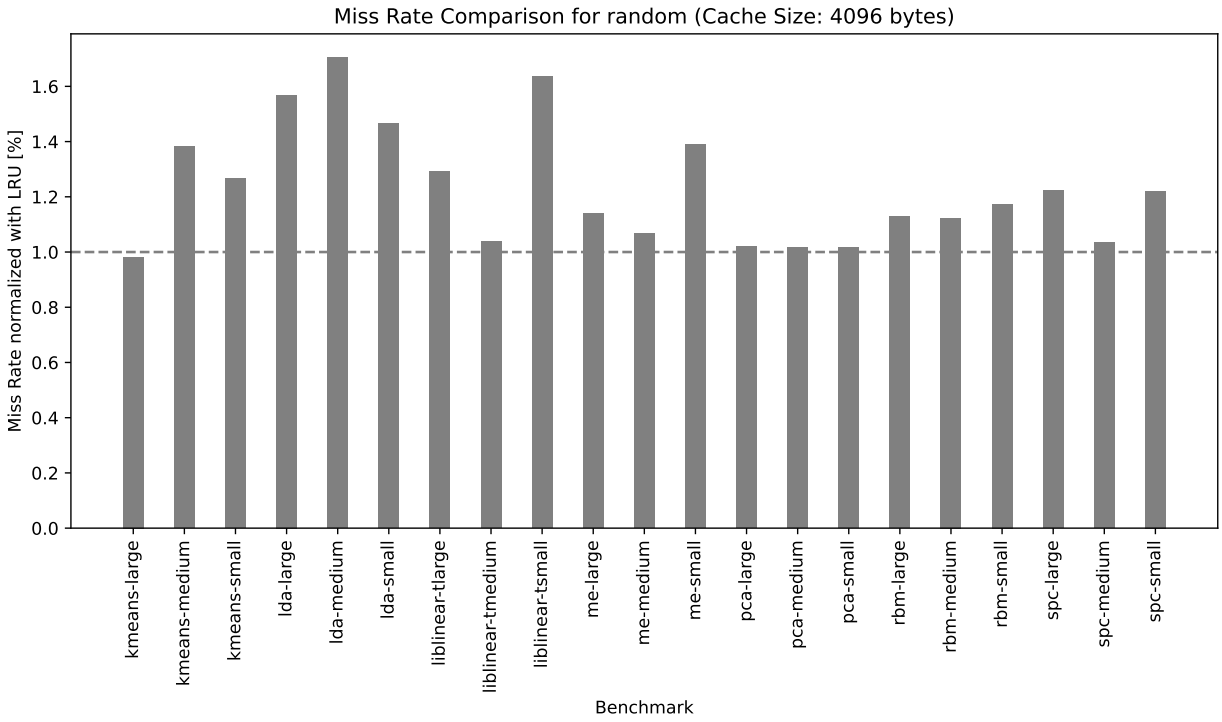
WANG, Z. et al. Using the compiler to improve cache replacement decisions. In: *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. [S.l.: s.n.], 2002. p. 199–208. ISSN 1089-795X.

WONG, H.; BETZ, V.; ROSE, J. Microarchitecture and circuits for a 200 mhz out-of-order soft processor memory system. *ACM Trans. Reconfigurable Technol. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 10, n. 1, dez. 2016. ISSN 1936-7406.

YUN, H. et al. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: *19th RTAS*. [S.l.: s.n.], 2013. p. 55–64.

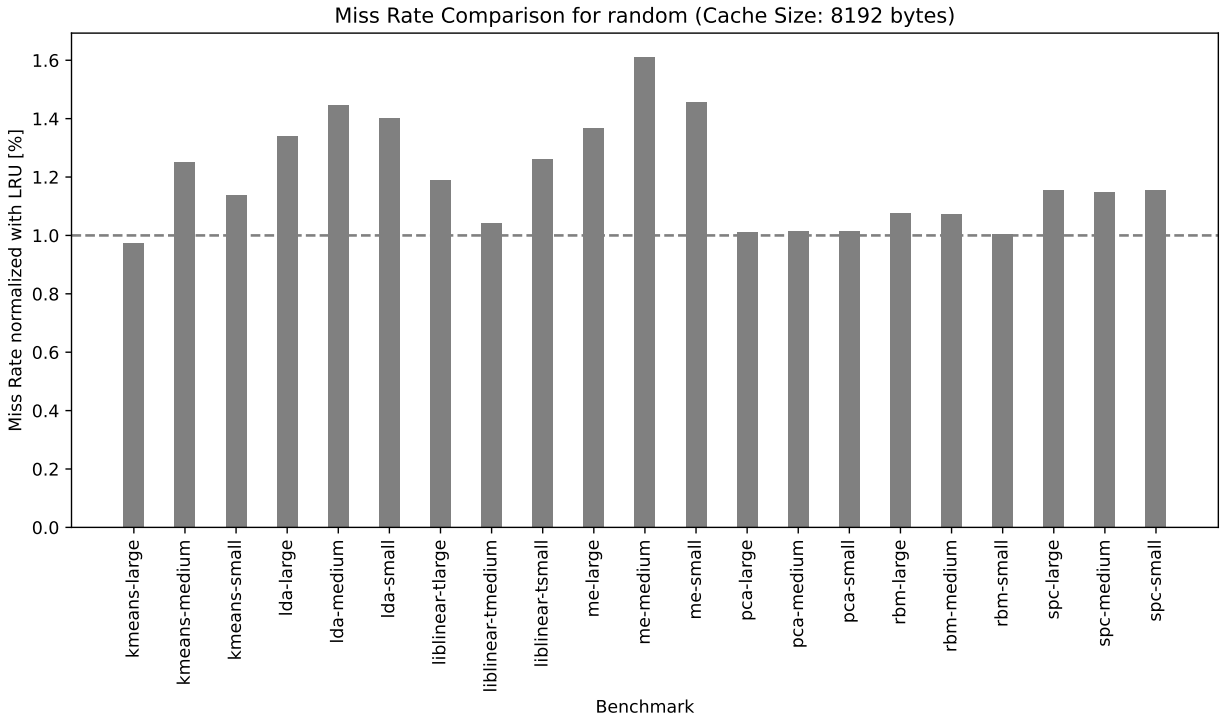
APPENDIX A – COMPARISON OF MISS RATE FOR DIVERSE BENCHMARKS

Figure A.1 – Miss rate evolution using Random with Cortex benchmarks (Cache size 4096 bytes).



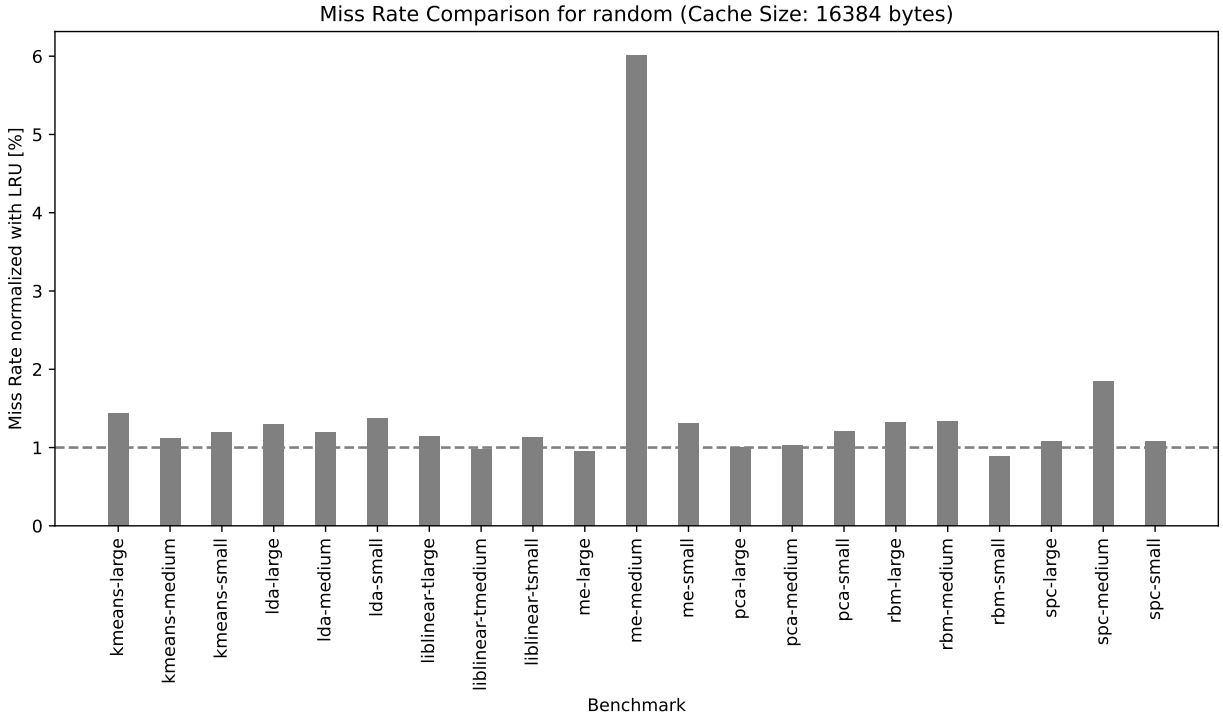
Source: Author (2024).

Figure A.2 – Miss rate evolution using Random with Cortex benchmarks (Cache size 8192 bytes).



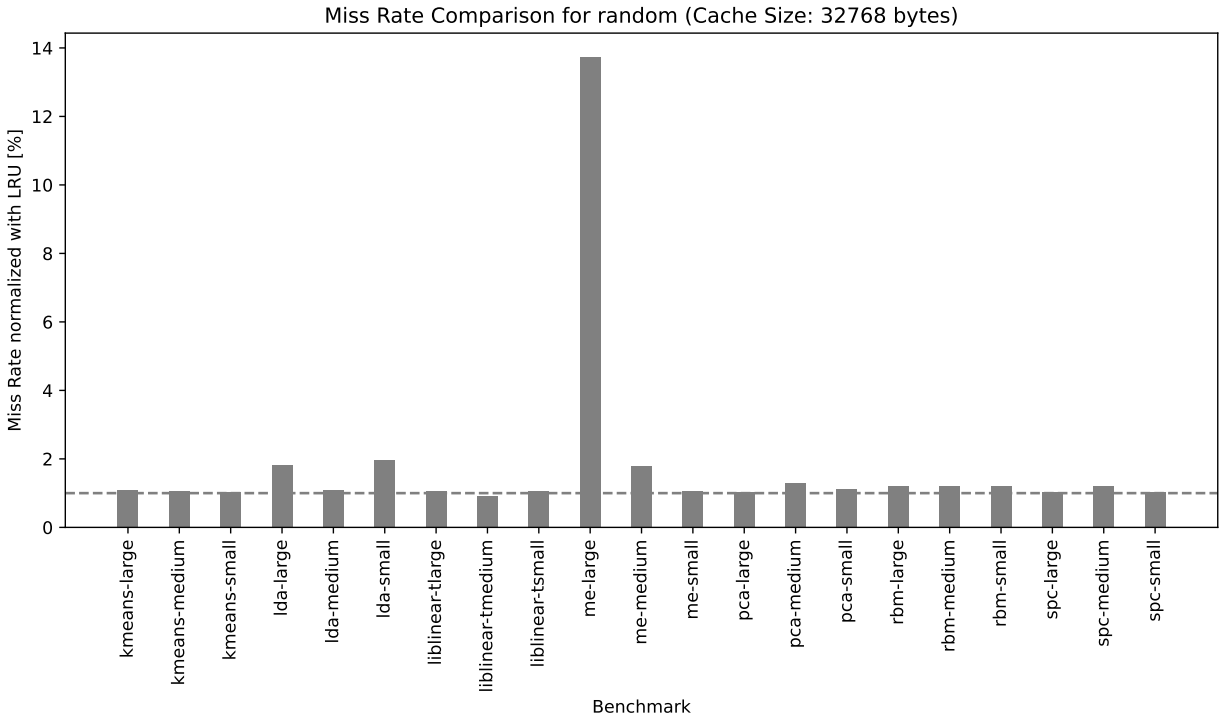
Source: Author (2024).

Figure A.3 – Miss rate evolution using Random with Cortex benchmarks (Cache size 16384 bytes).



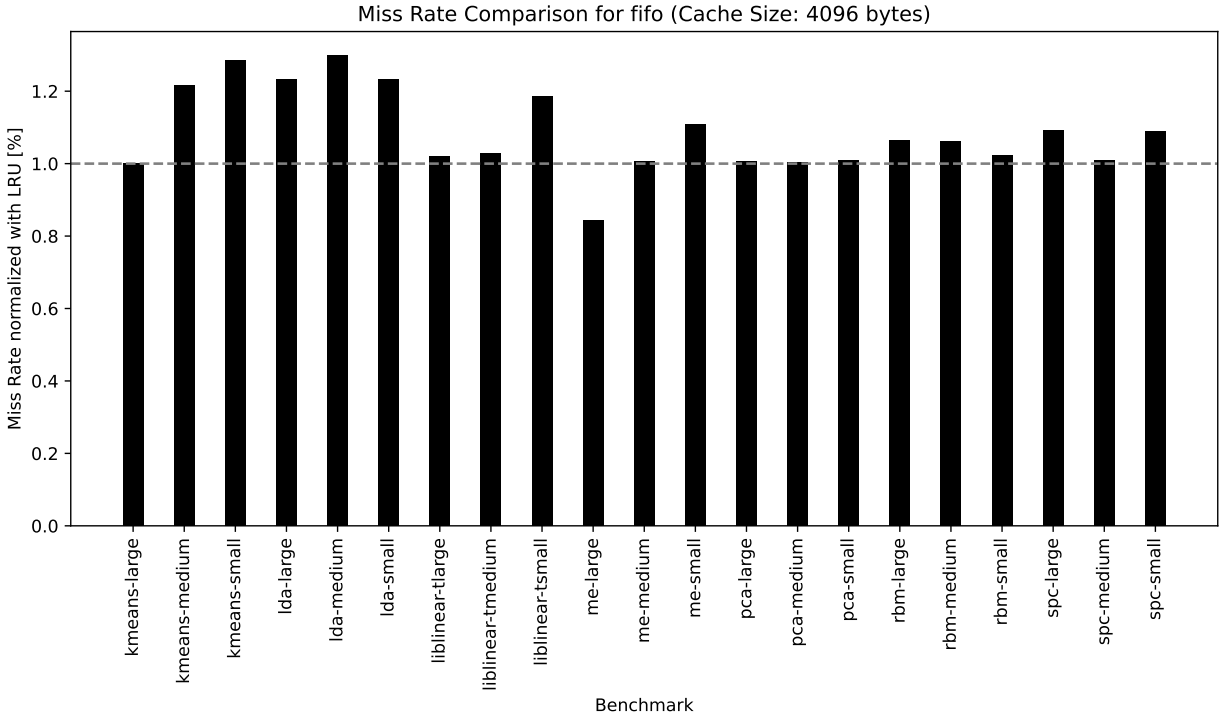
Source: Author (2024).

Figure A.4 – Miss rate evolution using Random with Cortex benchmarks (Cache size 32768 bytes).



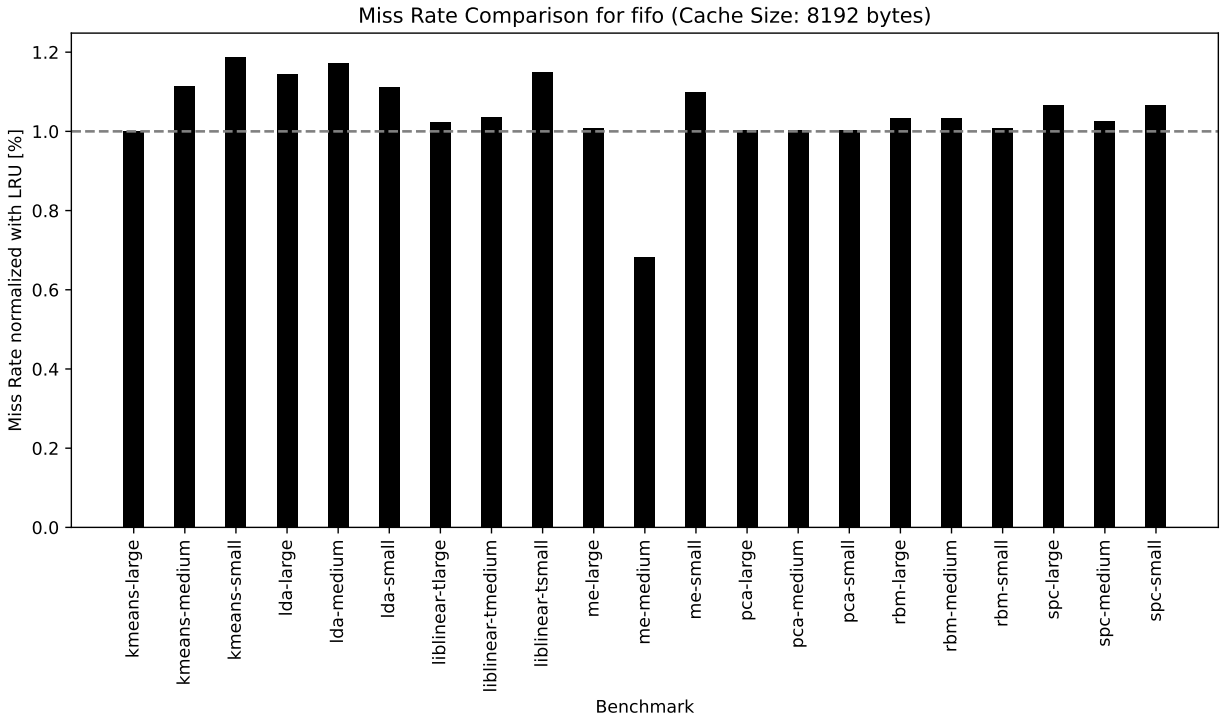
Source: Author (2024).

Figure A.5 – Miss rate evolution using FIFO with Cortex benchmarks (Cache size 4096 bytes).



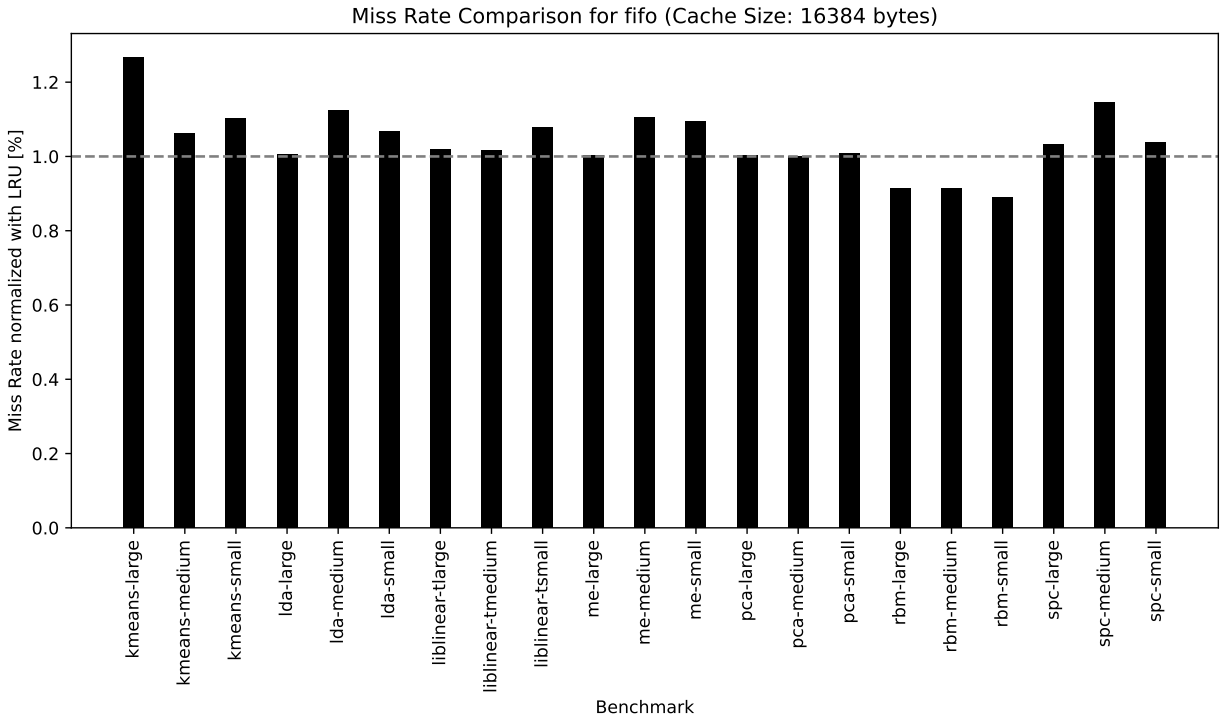
Source: Author (2024).

Figure A.6 – Miss rate evolution using FIFO with Cortex benchmarks (Cache size 8192 bytes).



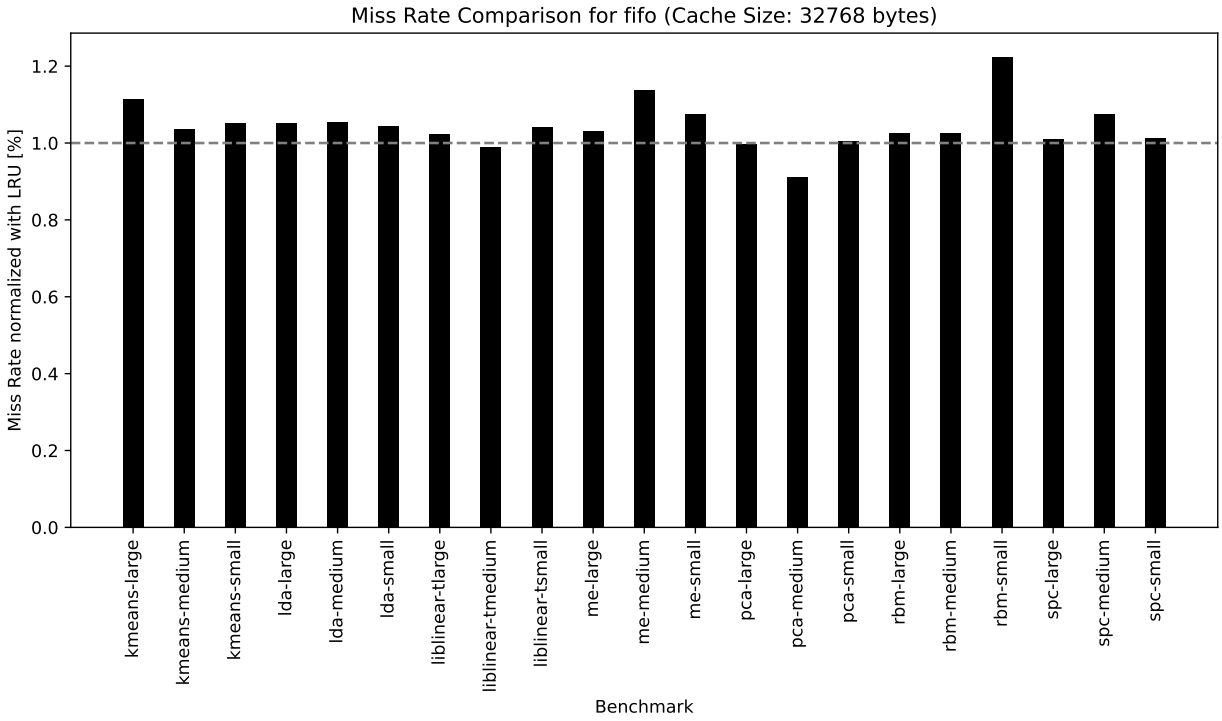
Source: Author (2024).

Figure A.7 – Miss rate evolution using FIFO with Cortex benchmarks (Cache size 16384 bytes).



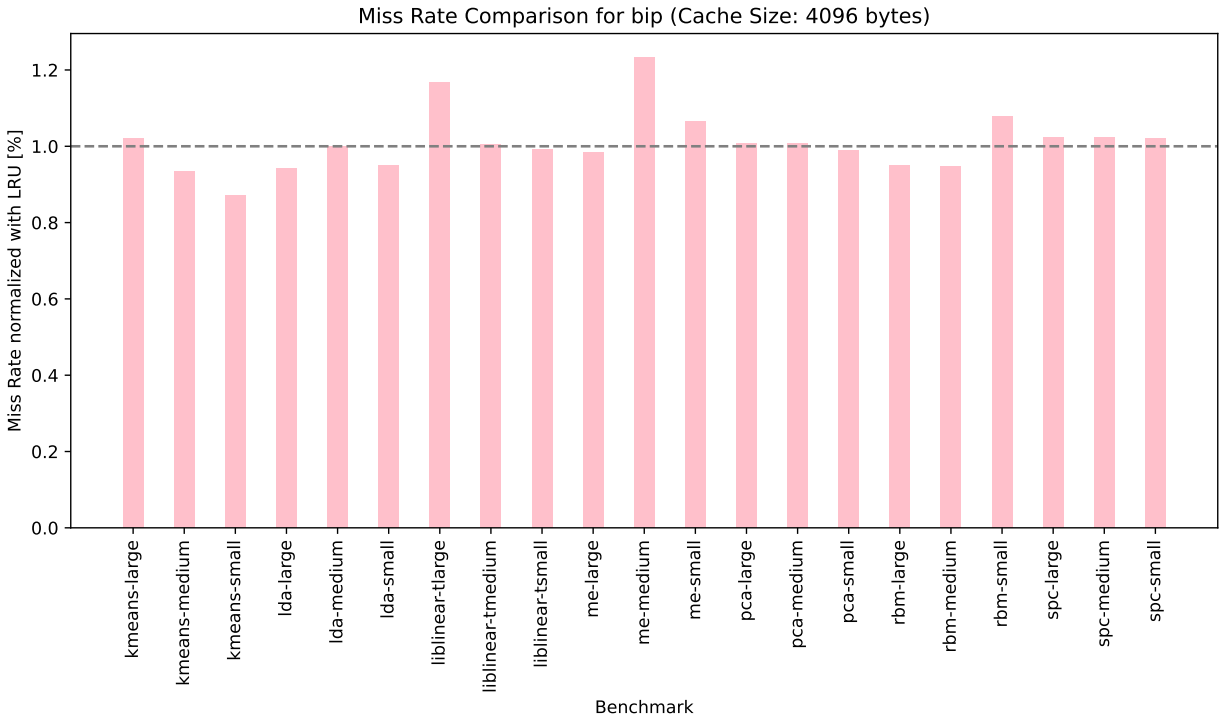
Source: Author (2024).

Figure A.8 – Miss rate evolution using FIFO with Cortex benchmarks (Cache size 32768 bytes).



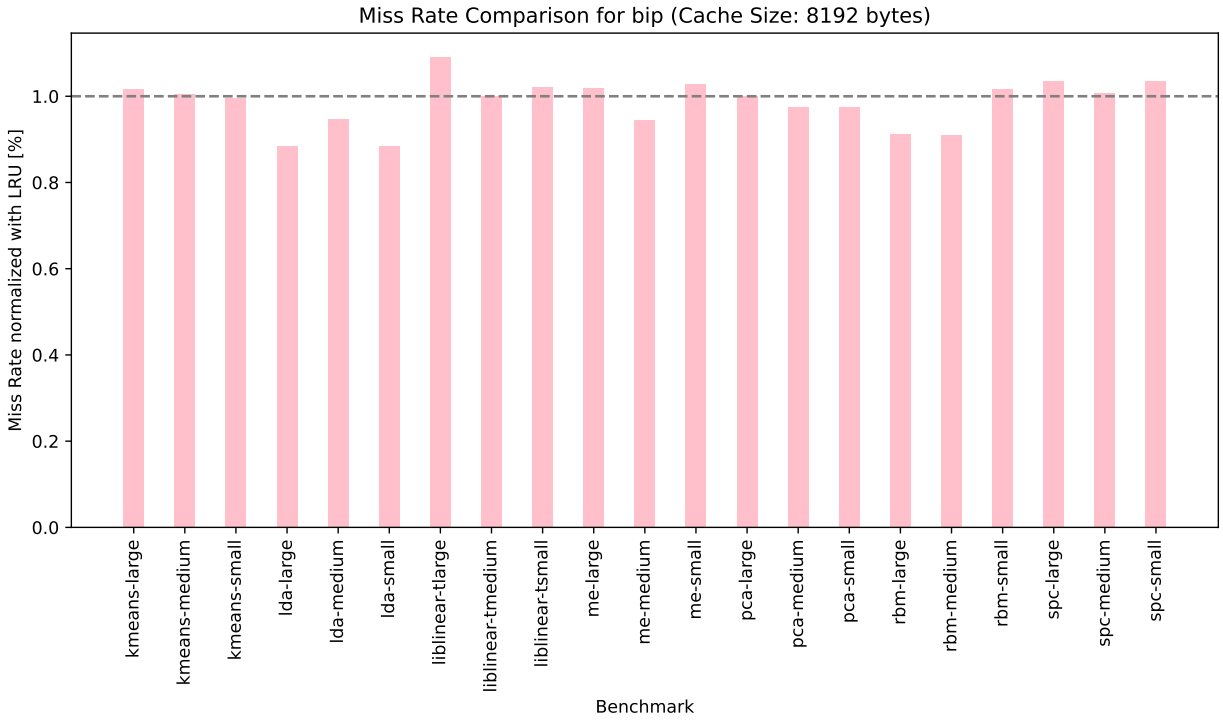
Source: Author (2024).

Figure A.9 – Miss rate evolution using BIP with Cortex benchmarks (Cache size 4096 bytes).



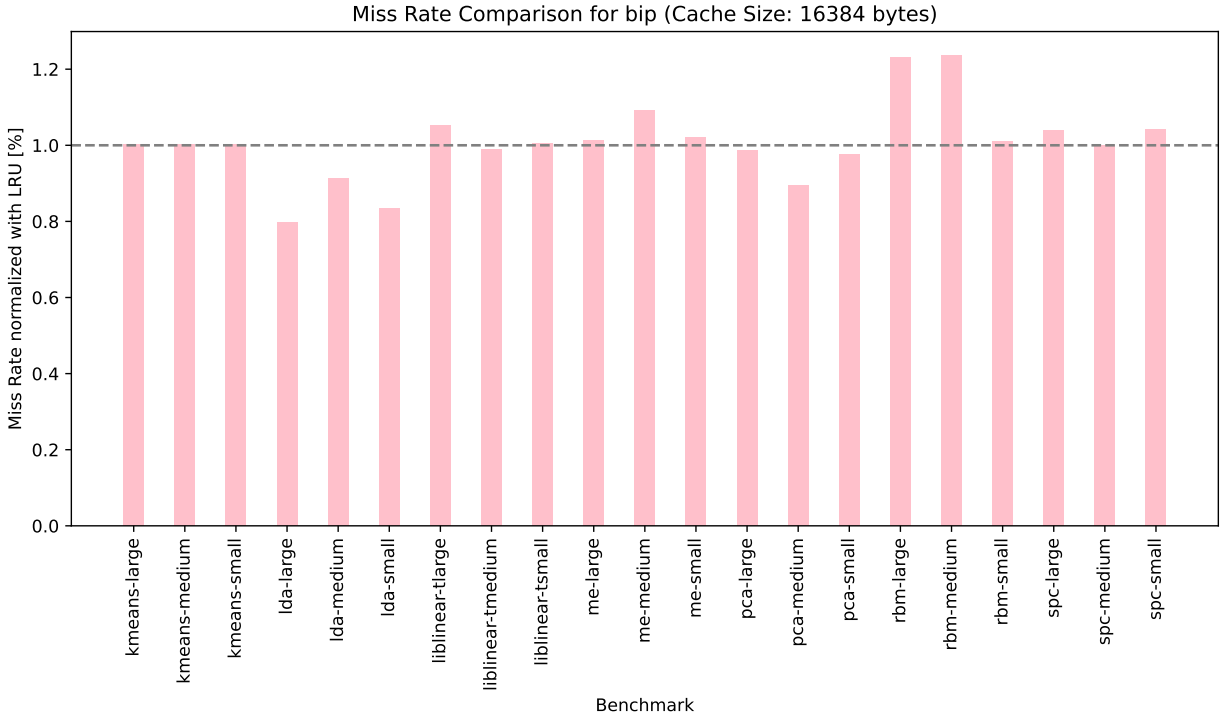
Source: Author (2024).

Figure A.10 – Miss rate evolution using BIP with Cortex benchmarks (Cache size 8192 bytes).



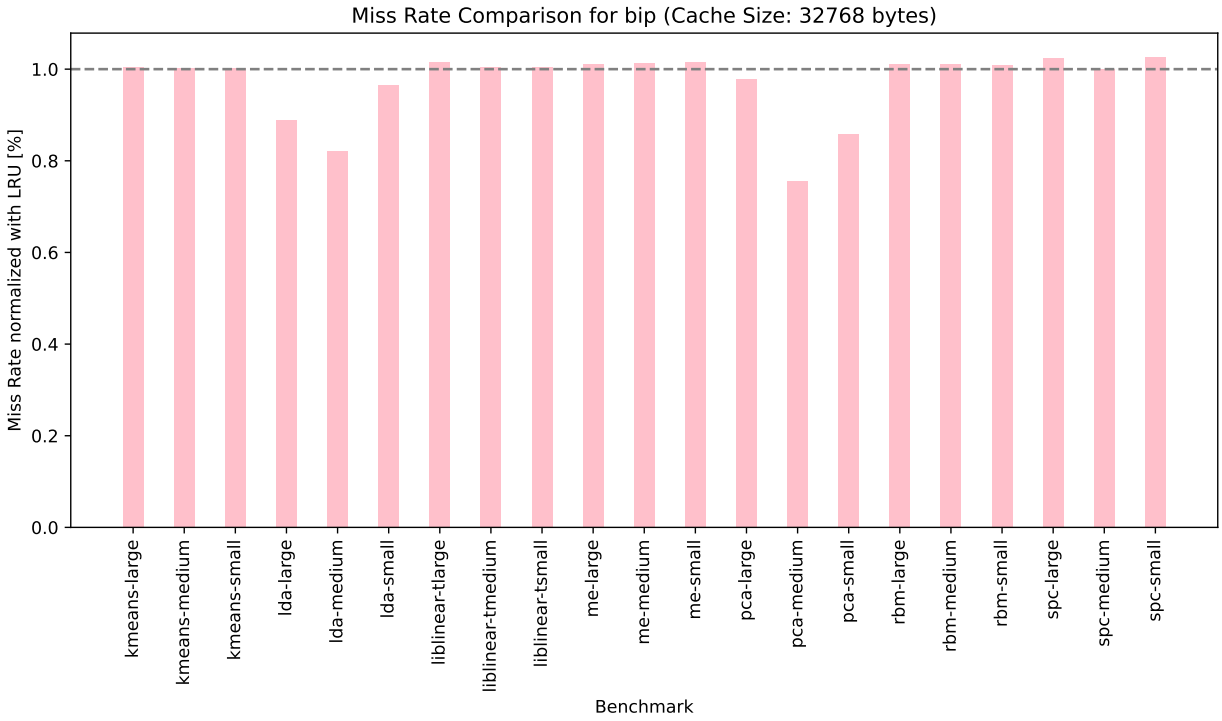
Source: Author (2024).

Figure A.11 – Miss rate evolution using BIP with Cortex benchmarks (Cache size 16384 bytes).



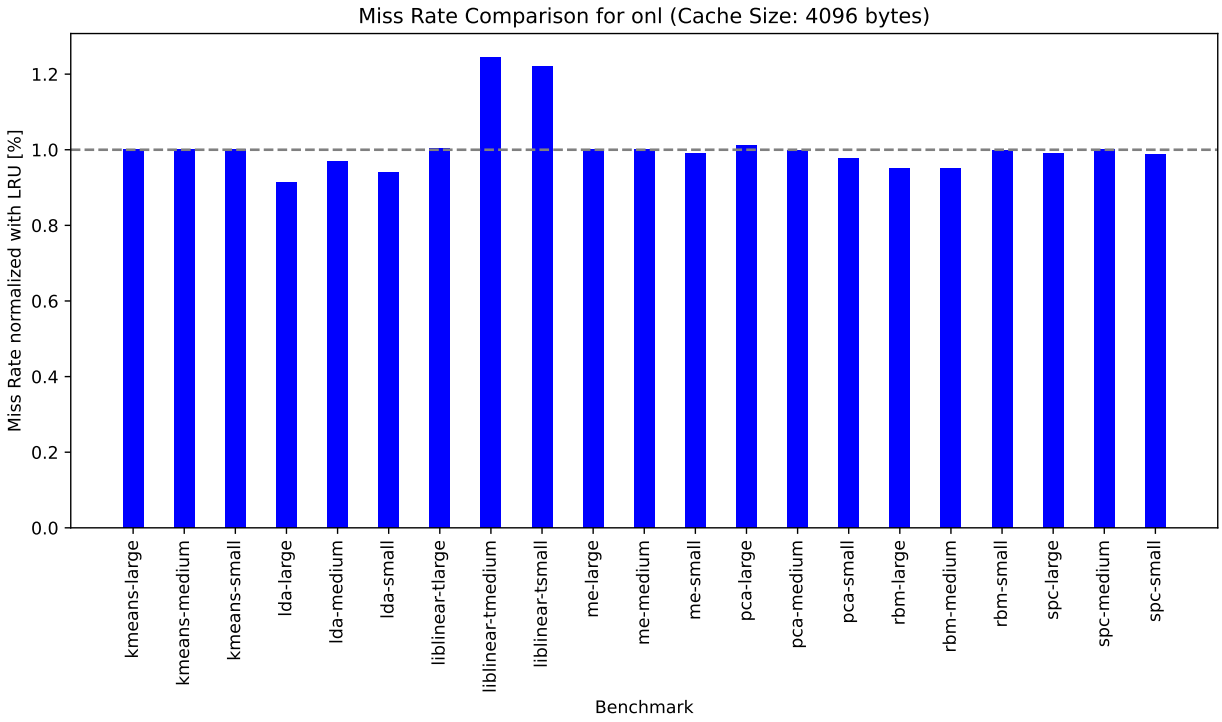
Source: Author (2024).

Figure A.12 – Miss rate evolution using BIP with Cortex benchmarks (Cache size 32768 bytes).



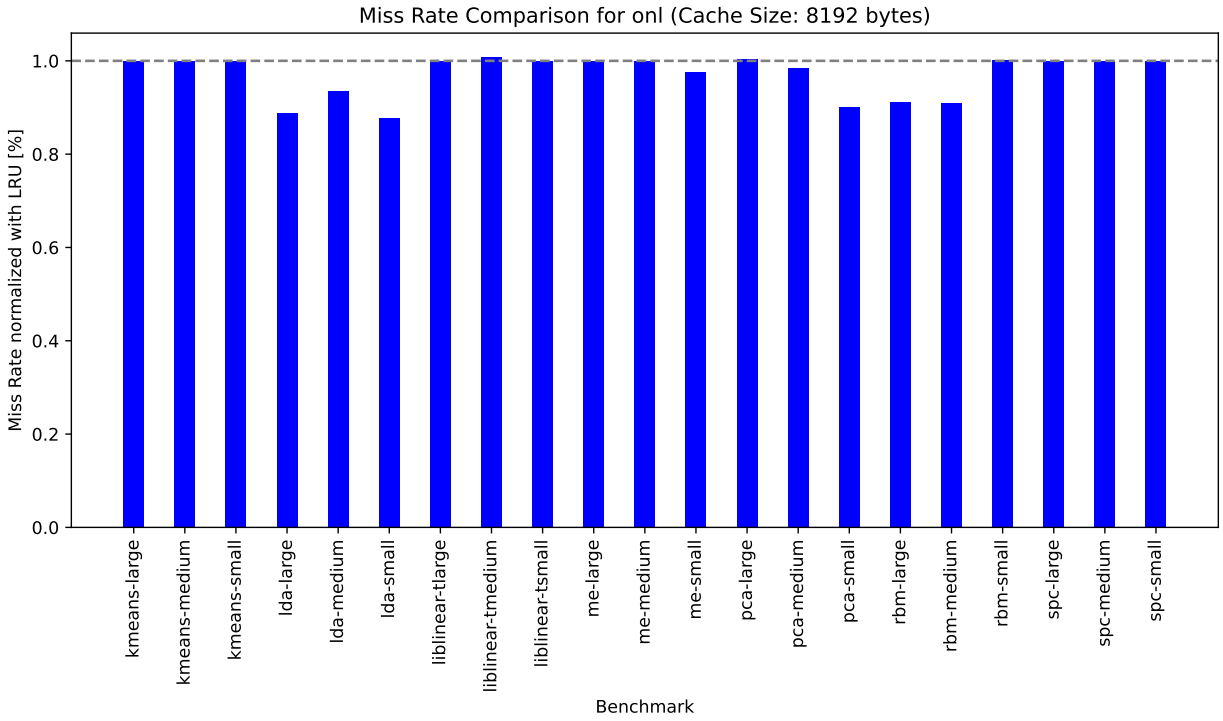
Source: Author (2024).

Figure A.13 – Miss rate evolution using Online with Cortex benchmarks (Cache size 4096 bytes).



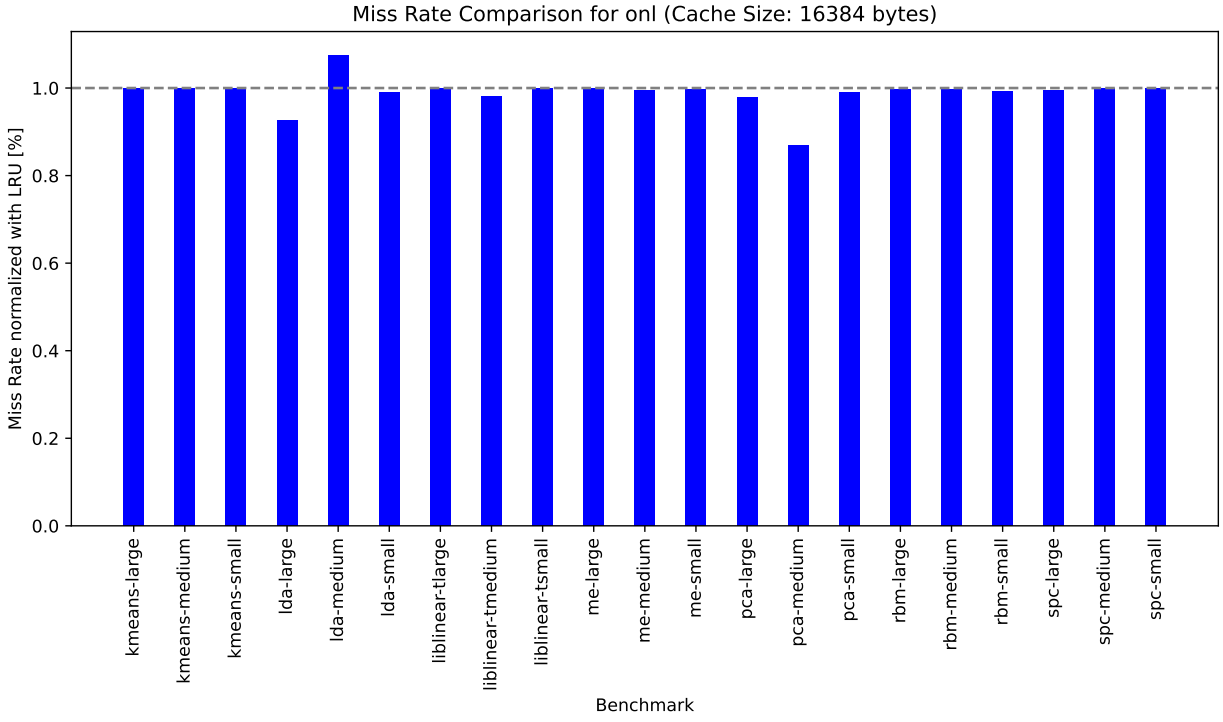
Source: Author (2024).

Figure A.14 – Miss rate evolution using Online with Cortex benchmarks (Cache size 8192 bytes).



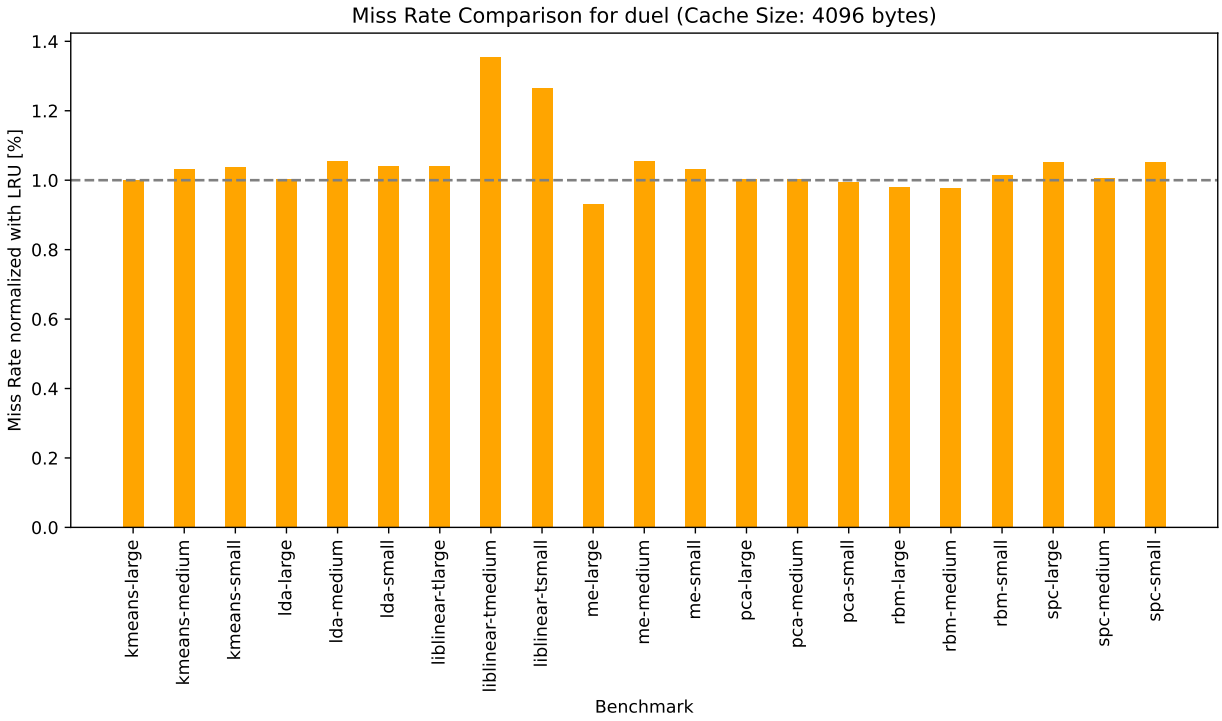
Source: Author (2024).

Figure A.15 – Miss rate evolution using Online with Cortex benchmarks (Cache size 16384 bytes).



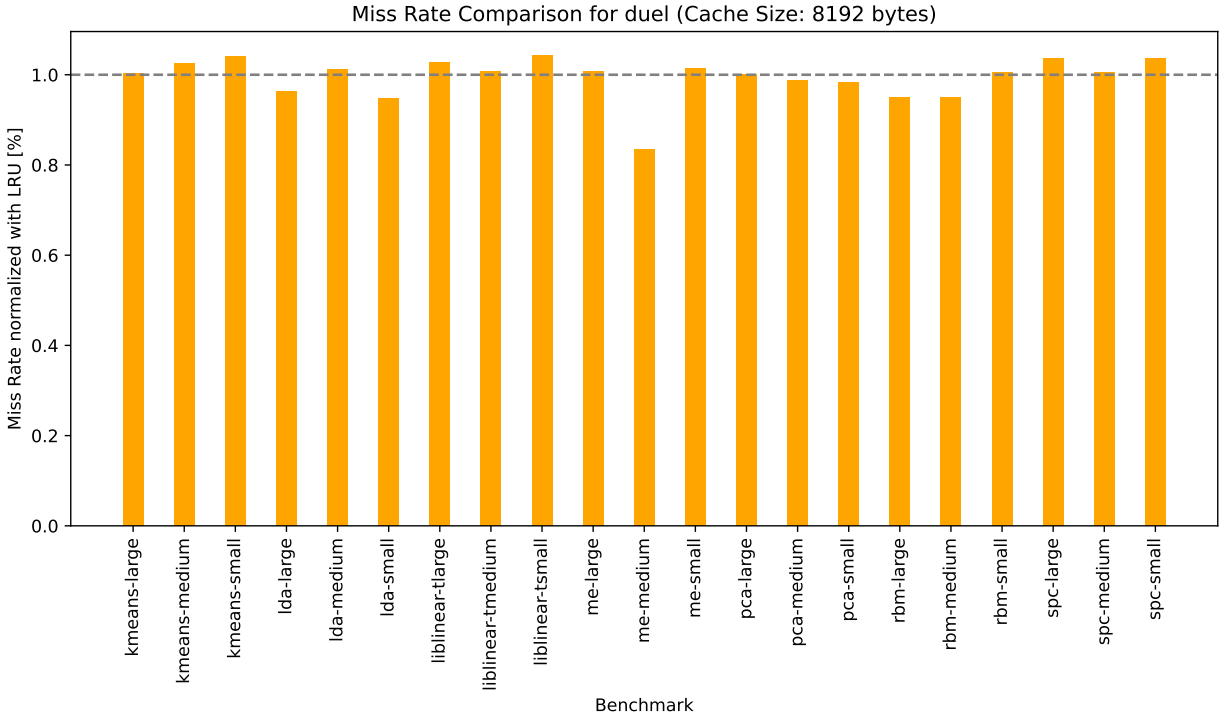
Source: Author (2024).

Figure A.16 – Miss rate evolution using Set Dueling with Cortex benchmarks (Cache size 4096 bytes).



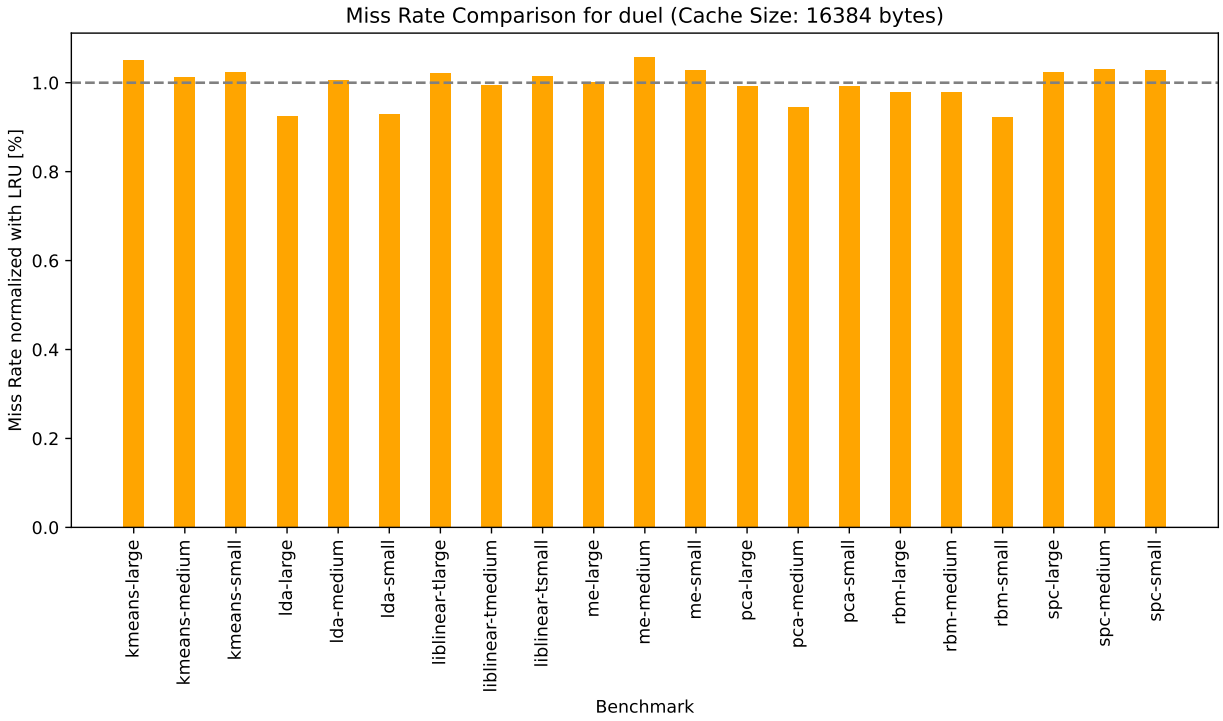
Source: Author (2024).

Figure A.17 – Miss rate evolution using Set Dueling with Cortex benchmarks (Cache size 8192 bytes).



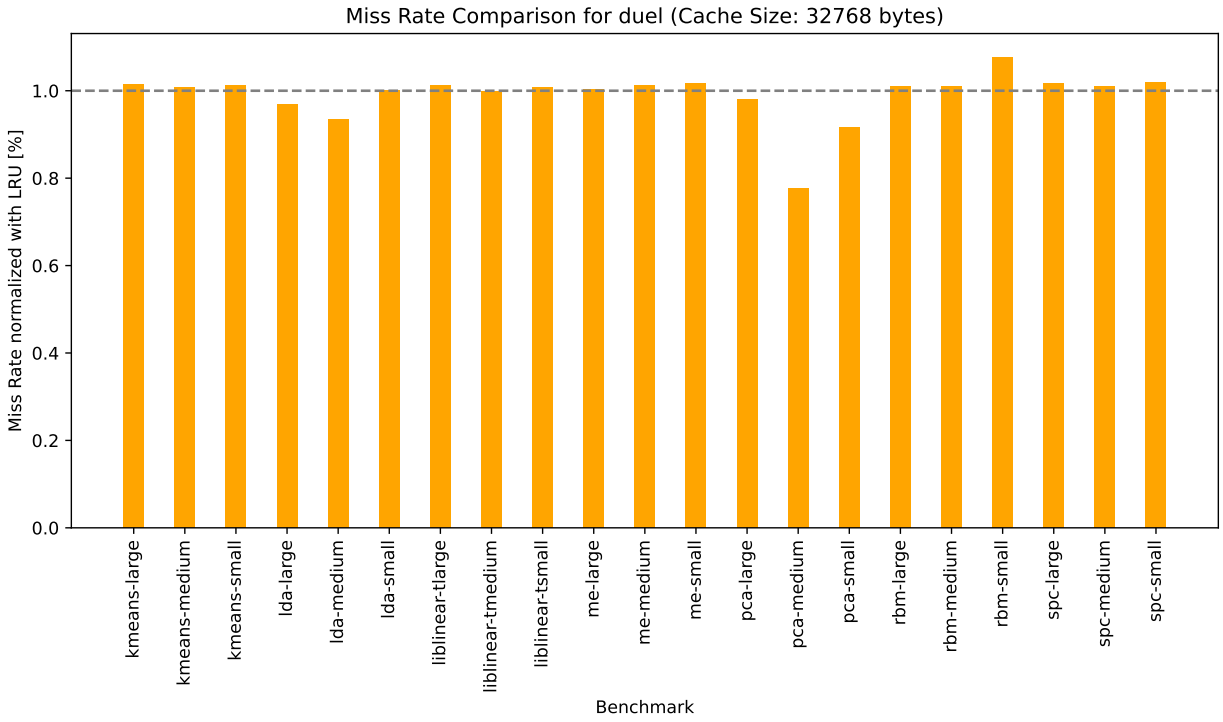
Source: Author (2024).

Figure A.18 – Miss rate evolution using Set Dueling with Cortex benchmarks (Cache size 16384 bytes).



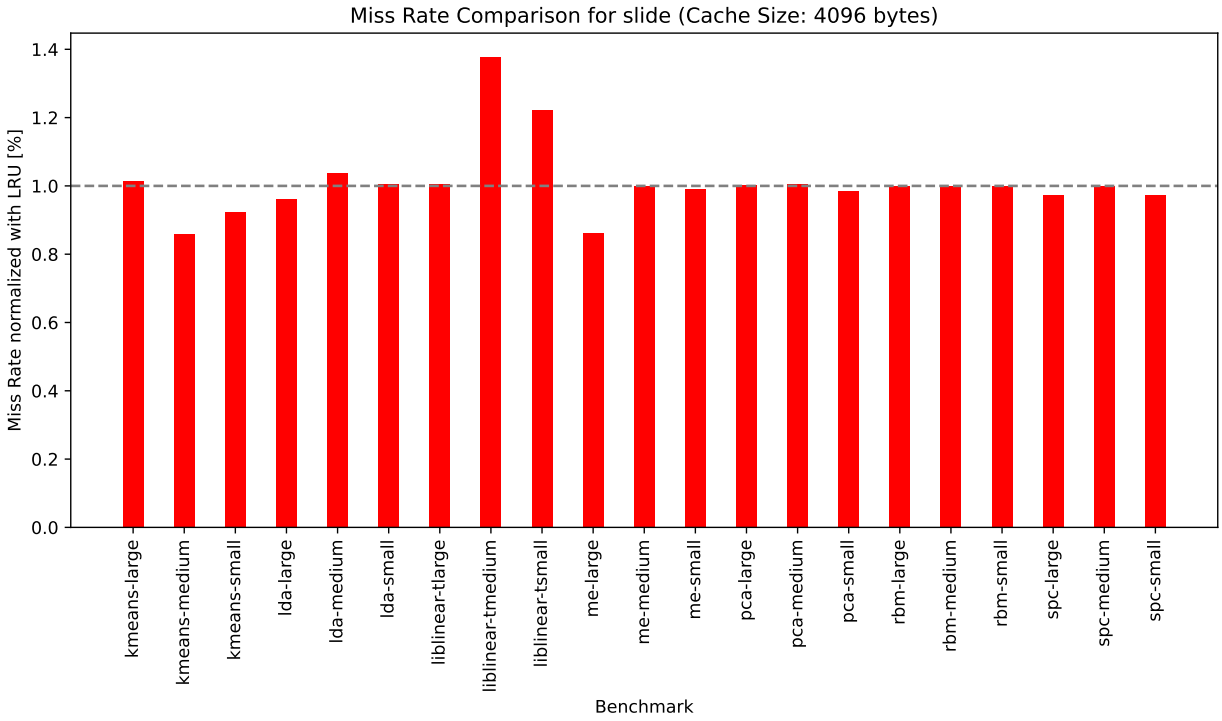
Source: Author (2024).

Figure A.19 – Miss rate evolution using Set Dueling with Cortex benchmarks (Cache size 32768 bytes).



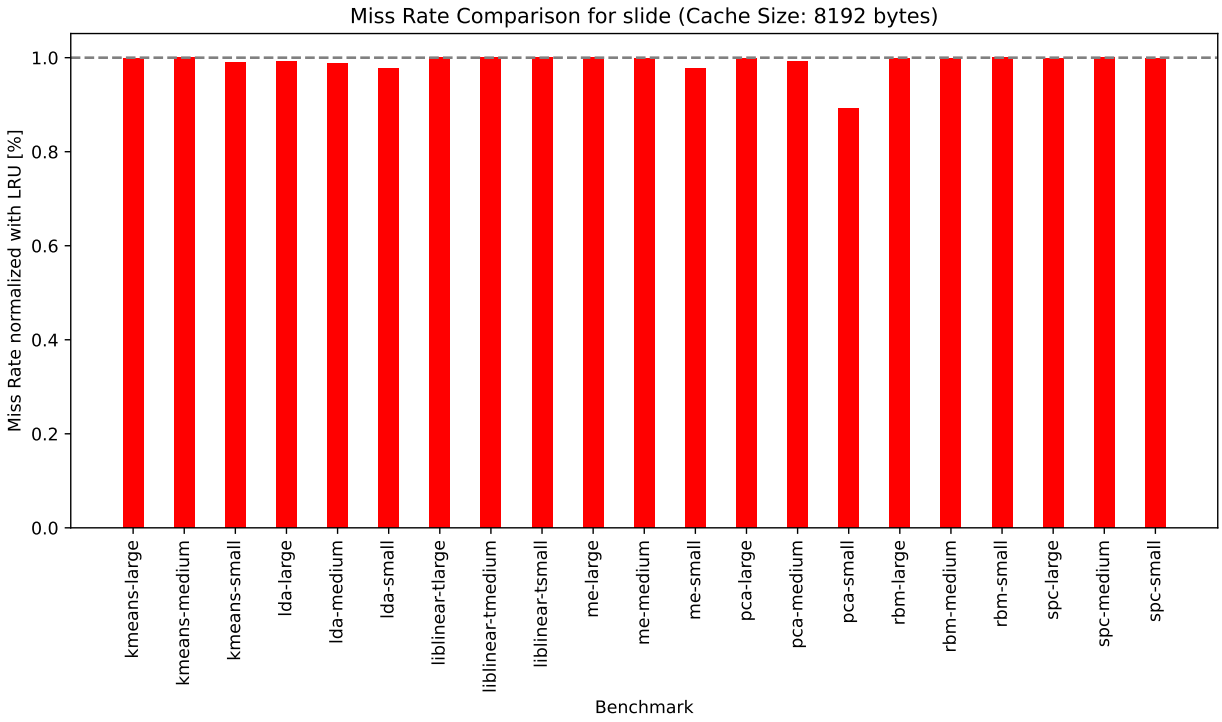
Source: Author (2024).

Figure A.20 – Miss rate evolution using Sliding Window with Cortex benchmarks (Cache size 4096 bytes).



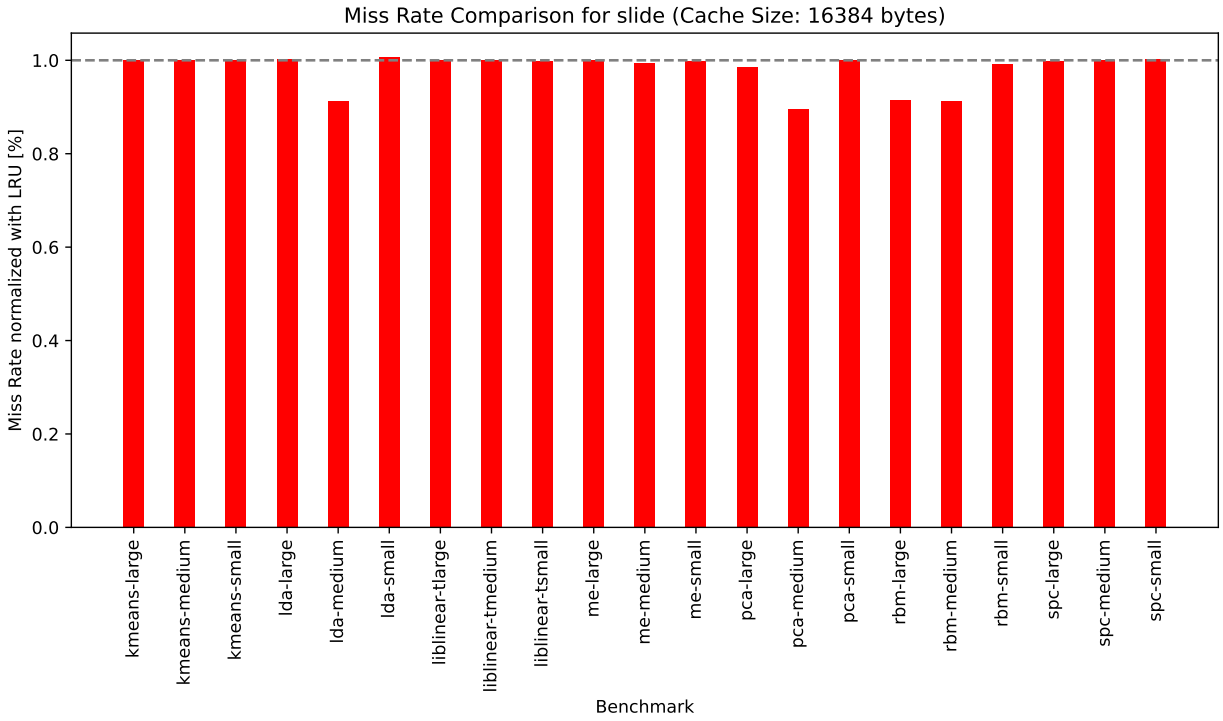
Source: Author (2024).

Figure A.21 – Miss rate evolution using Sliding Window with Cortex benchmarks (Cache size 8192 bytes).



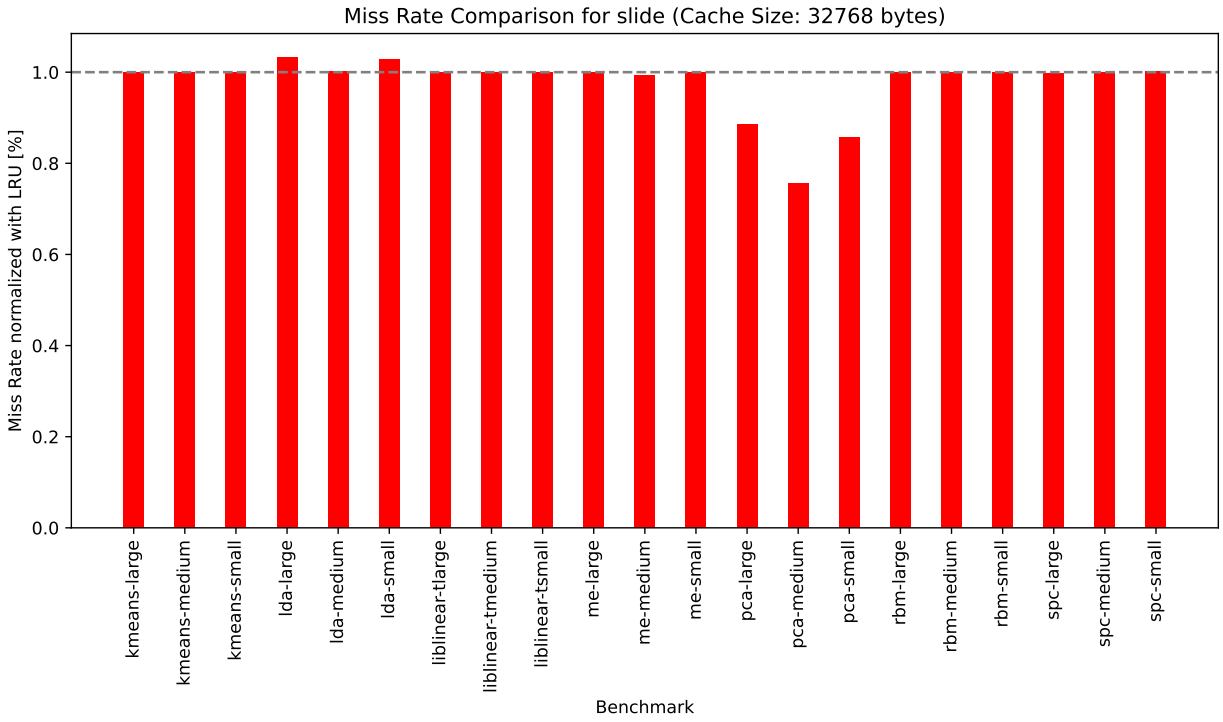
Source: Author (2024).

Figure A.22 – Miss rate evolution using Sliding Window with Cortex benchmarks (Cache size 16384 bytes).



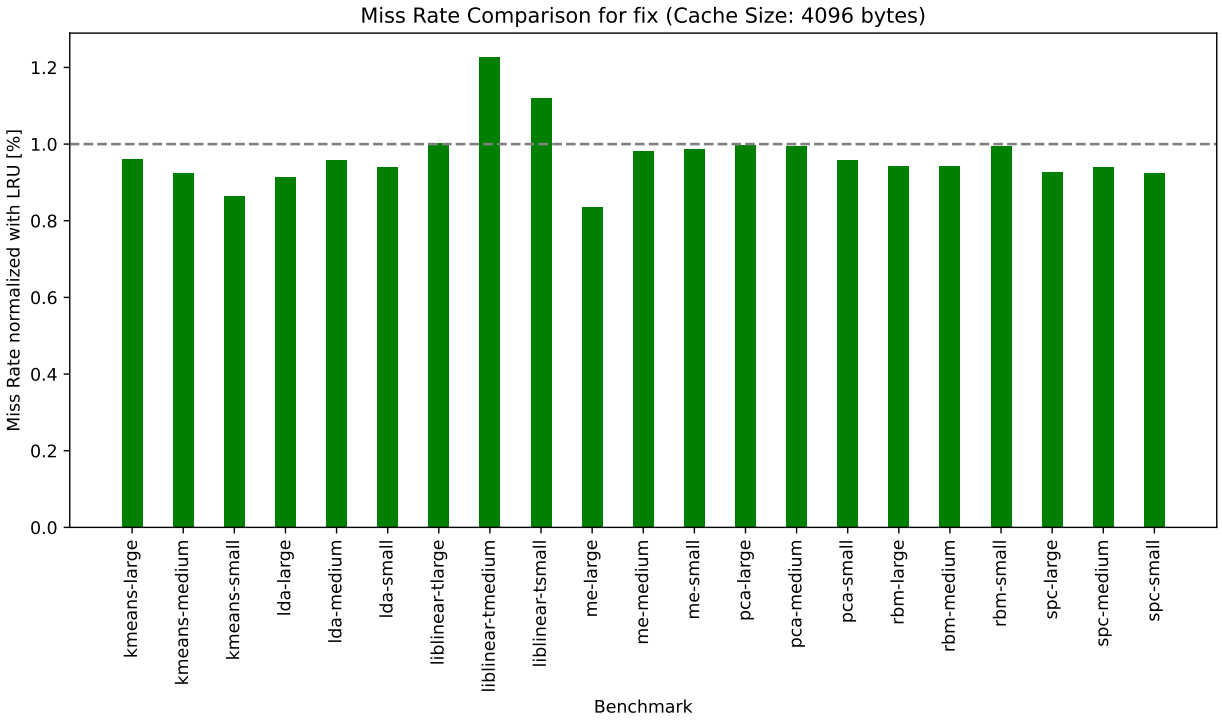
Source: Author (2024).

Figure A.23 – Miss rate evolution using Sliding Window with Cortex benchmarks (Cache size 32768 bytes).



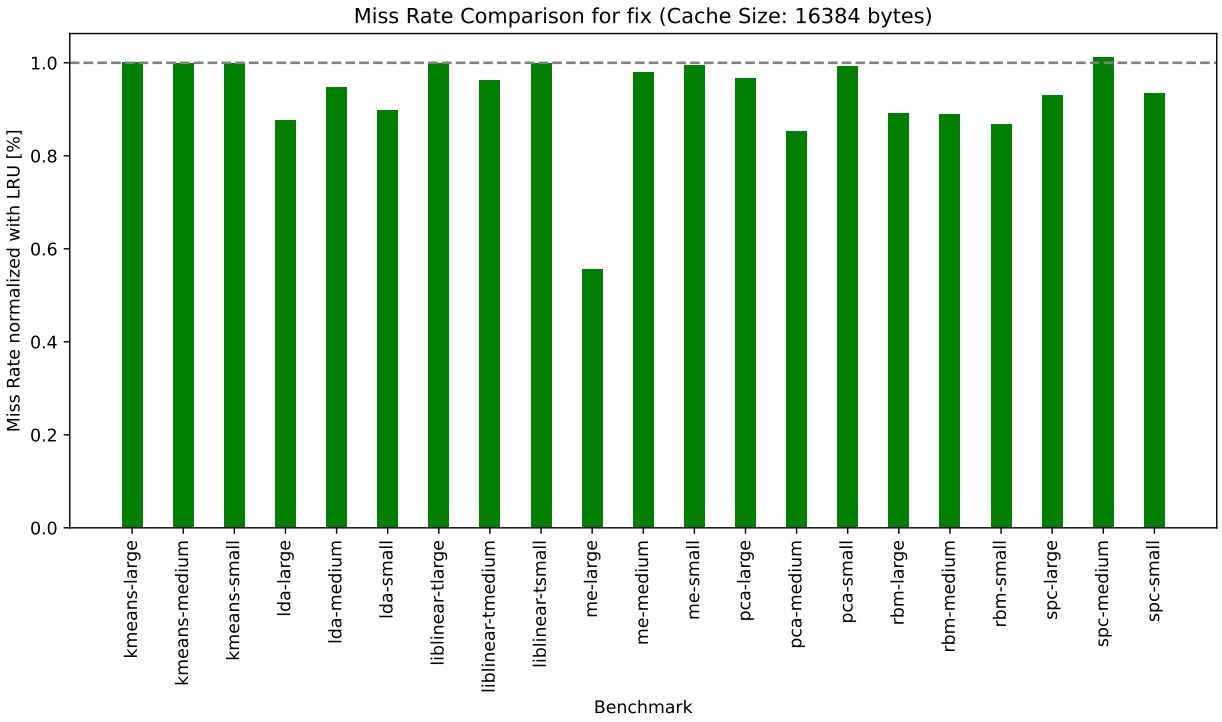
Source: Author (2024).

Figure A.24 – Miss rate evolution using Fix Window with Cortex benchmarks (Cache size 4096 bytes).



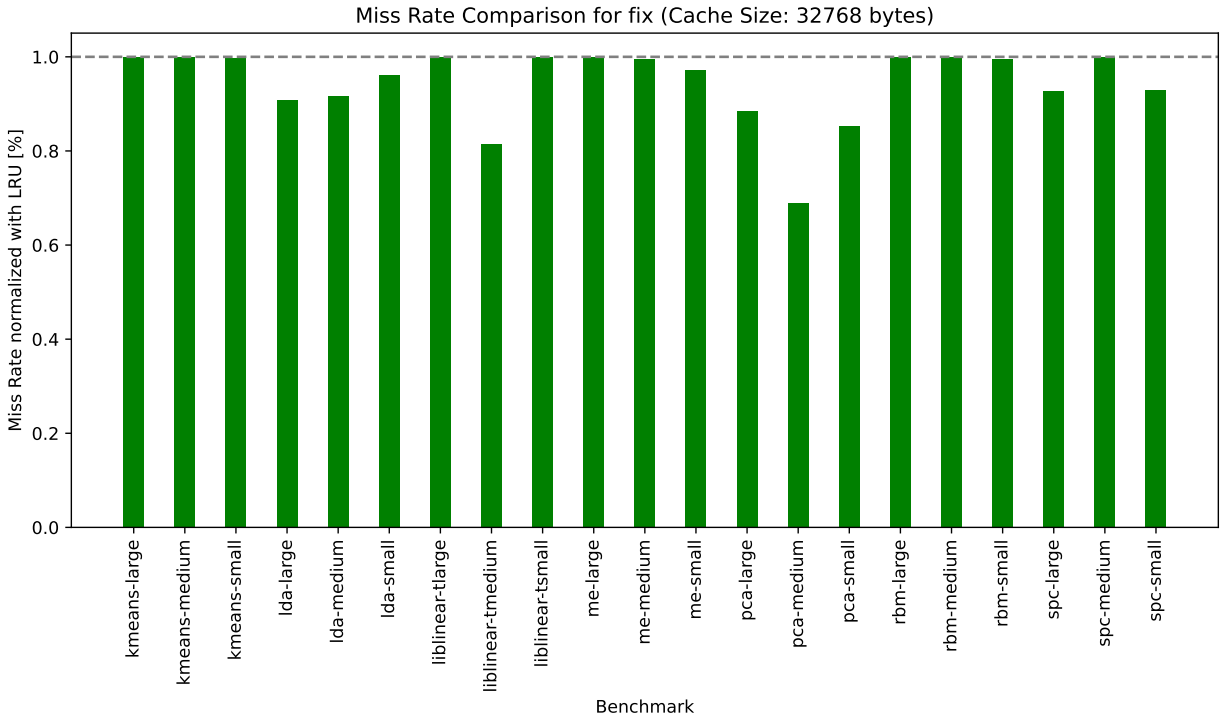
Source: Author (2024).

Figure A.25 – Miss rate evolution using Fix Window with Cortex benchmarks (Cache size 16384 bytes).



Source: Author (2024).

Figure A.26 – Miss rate evolution using Fix Window with Cortex benchmarks (Cache size 32768 bytes).



Source: Author (2024).