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

BRUNA ARRUDA ARAUJO

ANALYSIS AND OPTIMIZATION OF CACHE-RELATED PARAMETERS FOR REAL-TIME SYSTEMS

DISSERTATION OF DEGREE OF MASTER

Joinville
2023

Bruna Arruda Araujo

ANALYSIS AND OPTIMIZATION OF CACHE-RELATED PARAMETERS FOR REAL-TIME SYSTEMS

Dissertation presented to Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos of Universidade Federal de Santa Catarina Santa Catarina in partial fulfillment of the requirements for the degree of Master in Eletronic Systems Engineering.

Supervisor:: Dr. Giovani Gracioli
Co-supervisor:: Dr. Tomasz Kloda

Joinville
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Araujo, Bruna
Analysis and Optimization of Cache-Related Parameters for Real-time Systems / Bruna Araujo ; orientador, Giovani Gracioli, coorientador, Tomasz Kloda, 2023.
114 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Campus Joinville, Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos, Joinville, 2023.

Inclui referências.

1. Engenharia de Sistemas Eletrônicos. 2. Cache optimization. 3. Real-time systems. 4. Cache replacement policies. I. Gracioli, Giovani. II. Kloda, Tomasz. III. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Engenharia de Sistemas Eletrônicos. IV. Título.

Bruna Arruda Araujo

Analysis and Optimization of Cache-Related Parameters for Real-Time Systems

The present work in degree of Master was evaluated and approved by the committee composed by the following members:

Dr. Anderson Spengler
PPGESE - Universidade Federal de Santa Catarina

Dr. Alex Sandro Roschildt Pinto
PPGCC - Universidade Federal de Santa Catarina

We certify that this is the **original and final version** of the conclusion work that was deemed suitable for obtaining the title of Master in Eletronic Systems Engineering.

Dr. Lucas Weihmann
Program Coordinator

Dr. Giovani Gracioli
Advisor

Joinville, 25 de Julho de 2023.

To those who are, in some way, my family.

ACKNOWLEDGEMENTS

My heartfelt acknowledgments begin with my family. I want to express my gratitude to my parents, Adriane and Sandro, for their tireless dedication to me and my education. To my stepfather, Reginaldo, who has always treated me as a daughter and supported all my choices, encouraging me to go further. And to my stepmother, Laudeci, who has consistently aided my time away from home. I'd also like to extend my thanks to the Schnorr family, especially Jane, Paulo and Ezequiel, who welcomed me since my undergraduate years and became my pillar of support in various places.

A special thanks goes to my advisor, Giovani Gracioli, for providing me with such an engaging topic and for their unwavering patience during the completion of this work. I'd also like to express my gratitude to Tomasz Kloda, Dennis Hoornaert, Sergio Arribas and Lucas Santos for their immense contributions throughout the research. The academic and administrative support and the scientific environment at LISHA were pivotal in achieving the results of this thesis. Many thanks also to all my colleagues at LISHA.

I must also acknowledge some friends who played a crucial role during this period. Firstly, Vanessa Prediger, who was always ready to listen, share experiences, and assist me on countless occasions. I'd also like to extend my gratitude to individuals who were instrumental in helping me regain my self-confidence in the last year: Isaac Vianna, Renato Junk, Marcio Montibeler, Carmen Gaulke, Denis Dalpiaz, Éder Guber, Thais Musika, João Paulo Frotscher, Vivian Oliveira, Bianca Acuna, Johannes Sell and Manuela Rothbarth.

Finally, I'd like to thank the Federal University of Santa Catarina for enabling my academic journey. Last but not least, I'm deeply grateful to CAPES for their financial support during my years in the graduate program, which was essential for my life in Joinville.

" We ourselves feel that what we are doing is just a drop in the ocean.
But the ocean would be less because of that missing drop. "
(Mother Teresa of Calcutta, 1910 - 1997)

RESUMO

As memórias cache são fundamentais nos sistemas ciberfísicos contemporâneos devido às penalidades temporais incorridas por falhas da cache. Pesquisas anteriores se concentraram principalmente em políticas de substituição de linha e particionamento da cache em sistemas em tempo real para mitigar o impacto das falhas no Pior Tempo de Execução (*Worst Case Execution Time* - WCET).

Esta dissertação explora extensivamente várias políticas de substituição de linhas da cache em sistemas de tempo real, com o objetivo central de avaliar de forma abrangente o seu desempenho em termos de falhas e escalonabilidade – duas métricas cruciais na otimização de sistemas de tempo real. Os resultados da pesquisa confirmam que o tamanho da partição da cache e o número *ways* da cache exercem uma influência profunda no desempenho e na escalonabilidade da aplicação.

O impacto do tamanho da partição da cache nas políticas de substituição de linha é particularmente significativo. Os resultados experimentais ilustram que a modificação do tamanho da partição da cache pode levar a melhorias significativas no tempo de execução em aplicações específicas. Por exemplo, a aplicação `pca-small` obteve uma melhoria impressionante de 40% no tempo de execução com uma partição de 128 KB simplesmente alterando a política de substituição, destacando o papel crítico de otimizar políticas de substituição de linhas da cache e tamanhos de partição adaptados a tarefas específicas.

O número de formas da cache também desempenha um papel crucial nas políticas de substituição da cache. A transição de um número menor para um número maior de *ways* pode resultar em aumentos substanciais de desempenho em determinados cenários. Por exemplo, a aplicação `svd3-large` obteve uma melhoria de desempenho de aproximadamente 12% ao passar de quatro *ways* para 16 *ways*, atribuída principalmente à adoção da política LIP e de uma partição da cache de 64 KB. Essas observações enfatizam a influência significativa do número de *ways* da cache no comportamento e no desempenho da política de substituição de cache.

A escalonabilidade, uma consideração importante em sistemas de tempo real, é profundamente influenciada por essas otimizações da cache. Mesmo em cenários moderados de aceleração da cache, melhorias de escalonamento de até 5% podem ser alcançadas para casos não preemptivos. Em cenários com as maiores acelerações da cache, as melhorias na capacidade de escalonamento chegam a mais de 40% para casos não preemptivos em conjuntos de tarefas de alta utilização.

A contribuição central da dissertação reside no desenvolvimento de um algoritmo projetado para otimizar a utilização e escalonabilidade da cache em sistemas de tempo real, alocando dinamicamente partições da cache. Este algoritmo supera consistentemente o algoritimo de particionamento da cache ideal (*Optimal Cache Partitioning Algorithm* - OCPA) alcançando ganhos de até 2%.
Concluindo, esta pesquisa ressalta a importância dos parâmetros da cache no aumento da previsibilidade, eficiência e desempenho geral em sistemas de tempo real, ao mesmo tempo em que destaca o potencial para melhorias substanciais na escalonabilidade. Essas descobertas estabelecem as bases para futuras explorações e inovações no campo da otimização da cache em sistemas de tempo real.

Palavras-chave: Otimização da cache, sistemas de tempo real, políticas de substituição de linhas da cache, particionamento da cache, previsibilidade, escalonabilidade.

ABSTRACT

Cache memories are pivotal in contemporary cyber-physical systems due to the temporal penalties incurred from cache failures. Previous research has primarily focused on cache line replacement policies and cache partitioning in real-time systems to mitigate cache failures' impact on Worst Case Execution Time (WCET). However, the broader implications of cache parameters on these policies, especially in the context of cache partitions, have received limited attention.

This dissertation extensively explores various cache line replacement policies in real-time systems, with a central objective to comprehensively assess their performance in terms of cache misses and schedulability—two crucial metrics in optimizing real-time systems. The research findings confirm that cache partition size and the number of cache ways exert a profound influence on both application performance and schedulability.

Cache partition size's impact on cache policies is particularly significant. Experimental results illustrate that modifying the cache partition size can lead to significant runtime improvements in specific applications. For instance, the `pca-small` application achieved an impressive 40% runtime improvement with a 128 KB cache partition simply by changing the replacement policy, highlighting the critical role of optimizing cache replacement policies and partition sizes tailored to specific tasks.

The number of cache ways also plays a crucial role in cache replacement policies. Transitioning from a lower to a higher number of cache ways can result in substantial performance boosts in certain scenarios. For example, the `svd3-large` application saw a performance improvement of approximately 12% when moving from four cache ways to 16 cache ways, primarily attributed to adopting the LIP policy and a 64 KB cache partition. These observations emphasize the significant influence of the number of cache ways on cache replacement policy behavior and performance.

Schedulability, a key consideration in real-time systems, is profoundly influenced by these cache optimizations. Even under moderate cache acceleration scenarios, schedulability improvements of up to 5% can be achieved for non-preemptive cases. In scenarios with the highest cache accelerations, schedulability improvements soar to over 40% for non-preemptive cases in high-utilization task sets.

The dissertation's central contribution lies in the development of an algorithm designed to optimize cache utilization and schedulability in real-time systems by dynamically allocating cache partitions. This algorithm consistently outperforms the Optimal Cache Partitioning Algorithm (OCPA) achieving gains of up to 2%.

In conclusion, this research underscores the significance of cache parameters in enhancing predictability, efficiency, and overall performance in real-time systems while highlighting the potential for substantial schedulability improvements. These findings lay the foundation for future exploration and innovation in the field of cache optimization within real-time systems.

Keywords: Cache optimization, real-time systems, cache line replacement policies, cache partitioning, predictability, schedulability.

RESUMO EXPANDIDO

Introdução

Na era da Indústria 4.0 e da proliferação da Internet das Coisas (IoT), os Sistemas Ciberfísicos (CPS) tornaram-se a espinha dorsal das tecnologias modernas. Esses sistemas, também conhecidos como CPS críticos, são essenciais em setores cruciais, como aviação, energia nuclear, satélites, estações espaciais, veículos autônomos e cirurgia robótica. A evolução tecnológica trouxe um desafio significativo para esses sistemas: a crescente inteligência exigida, especialmente no contexto de Inteligência Artificial (IA). Contudo, essa complexidade crescente traz consigo um problema crítico: a previsibilidade limitada do comportamento dos CPS.

Nesse cenário, a memória cache, um componente central dos sistemas computacionais, desempenha um papel fundamental. As políticas de substituição de linha da cache têm um impacto direto na previsibilidade da cache, influenciando diretamente o Tempo de Execução do Pior Caso (WCET) e a capacidade de escalonamento dos sistemas em tempo real. Falhas de cache e as penalidades associadas podem causar sérios problemas, especialmente em sistemas críticos, onde a previsibilidade é essencial.

Objetivos

O principal objetivo desta pesquisa é analisar e otimizar os parâmetros relacionados à cache, como tamanho da partição, número de *ways*, tamanho da linha da cache e a política de substituição para cada partição, a fim de aprimorar a capacidade de escalonamento de sistemas críticos em tempo real. Para alcançar esse objetivo abrangente, os seguintes objetivos específicos foram definidos: conduzir uma revisão bibliográfica abrangente das pesquisas relacionadas para reunir conhecimento no campo das políticas de substituição de linha da cache e seu impacto nos sistemas em tempo real. Além disso, avaliar o desempenho das políticas de substituição populares, incluindo LRU, FIFO, RANDOM, LIP e BIP, em termos de falhas da cache e escalonabilidade de tarefas em tempo real, usando um conjunto diversificado de *benchmarks*. A pesquisa também visa implementar e avaliar o desempenho da Política de Inserção Dinâmica (DIP) em um simulador de cache, especificamente cachegrind, para determinar sua eficácia na redução de falhas de cache. Além disso, investigar e analisar os efeitos de diferentes configurações da cache no desempenho das políticas de substituição de linha da cache, com foco em seu impacto nos sistemas críticos em tempo real. Por fim, desenvolver um algoritmo que otimize os parâmetros da cache e as políticas de substituição, considerando a abordagem de particionamento da

cache para tarefas em um sistema em tempo real. Ao abordar esses objetivos específicos, esta pesquisa visa contribuir para o avanço das técnicas de otimização da cache para sistemas críticos em tempo real, visando melhorar a escalonabilidade do sistema e o desempenho geral.

Metodologia

Esta pesquisa adota uma abordagem sistemática para atingir seus objetivos. Inicialmente, é realizada uma revisão bibliográfica abrangente para estabelecer uma base teórica sólida. Em seguida, uma série de experimentos é conduzida, implementando e avaliando diversas políticas de substituição de linha da cache, como LRU, FIFO, RANDOM, LIP e BIP, usando benchmarks variados para representar cenários do mundo real.

Além disso, a eficácia da Política de Inserção Dinâmica (DIP) é analisada em um simulador de mémoria cache. Investigações detalhadas sobre diferentes configurações da cache são realizadas, com foco especial em seu impacto nos sistemas críticos de tempo real. Por fim, um algoritmo inovador é desenvolvido e testado em simulações controladas e cenários do mundo real para validar sua aplicabilidade.

Resultados e Discussões

Os resultados experimentais destacaram a influência significativa do tamanho da partição de cache nas políticas de substituição de linha. Modificar o tamanho da partição da cache teve um impacto notável no tempo de execução de aplicações específicas. Por exemplo, a aplicação `pca-small` alcançou uma impressionante melhoria de 40% no tempo de execução com uma partição de cache de 128 KB, simplesmente alterando a política de substituição, evidenciando o papel crítico da otimização das políticas de substituição de linha de cache e dos tamanhos de partição adaptados a tarefas específicas.

Além disso, o número de *ways* da cache também desempenhou um papel crucial nas políticas de substituição de linha. A transição de um número menor para um número maior de *ways* da cache resultou em aumentos substanciais de desempenho em cenários específicos. Por exemplo, a aplicação `svd3-large` obteve uma melhoria de desempenho de aproximadamente 12% ao passar de quatro para 16 vias da cache, atribuída principalmente à adoção da política LIP e a uma partição da cache de 64 KB. Essas observações enfatizam a influência significativa do número de *ways* da cache no comportamento e no desempenho da política de substituição de linhas da cache.

A escalonabilidade, uma consideração crucial em sistemas em tempo real, foi profundamente influenciada por essas otimizações. Mesmo em cenários

moderados de aceleração da cache, melhorias de escalonabilidade de até 5% foram alcançadas para casos não preemptivos. Em cenários com as maiores acelerações, as melhorias na capacidade de escalonamento chegaram a mais de 40% para casos não preemptivos em conjuntos de tarefas de alta utilização.
A contribuição central da dissertação reside no desenvolvimento de um algoritmo projetado para otimizar a utilização e escalonabilidade da cache em sistemas em tempo real, alocando dinamicamente partições da cache. Este algoritmo supera consistentemente o algoritmo de particionamento da cache ideal (OCPA), alcançando ganhos de até 2%. Esses resultados reais validam a eficácia do algoritmo proposto, demonstrando melhorias tangíveis na previsibilidade e na capacidade de escalonamento dos sistemas em tempo real, estabelecendo assim um novo padrão na otimização de cache para esses ambientes críticos.

Considerações Finais
Em resumo, esta pesquisa ressalta a influência fundamental dos parâmetros da cache na previsibilidade, eficiência e desempenho geral de sistemas de tempo real. Além disso, demonstra o potencial para melhorias substanciais na escalonabilidade. As percepções obtidas a partir dos estudos sobre partição da cache e minimização do uso da cache em sistemas em tempo real enriquecem ainda mais nossa compreensão das estratégias de gerenciamento da cache. Essas descobertas estabelecem as bases para futuras explorações e inovações no campo da otimização da cache em sistemas de tempo real.

Palavras-Chaves: Otimização da cache, sistemas de tempo real, políticas de substituição de linhas da cache, particionamento da cache, previsibilidade, escalonabilidade.

LIST OF FIGURES

Figure 1.1 – Number of cache misses from different benchmarks varying cache override policy. Normalized according to LRU. 128 KB cache size with 8 ways. 31
Figure 2.1 – Memory hierarchy structure. 37
Figure 2.2 – Multicore system architecture with dedicated L1 and L2 caches and shared LLC (L3). 40
Figure 2.3 – Direct mapping of a memory with 32 blocks into an eight-word cache. 42
Figure 2.4 – Associative mapping of a block of main memory 43
Figure 2.5 – Associative mapping per set. 44
Figure 2.6 – FIFO replacement policy. 47
Figure 2.7 – LRU replacement policy. 48
Figure 2.8 – How the PLRU policy works when a hit occurs. 50
Figure 2.9 – How the PLRU policy works when a cache miss occurs. 50
Figure 2.10 – LIP adaptive insertion policy. 52
Figure 4.1 – Impact of cache misses on execution time per processor, varying the size of the cache partition in 8 ways. 78
Figure 4.2 – Impact of number of ways on execution time, varying cache partition size. 79
Figure 4.3 – Schedulability rate of non-preemptive fixed-priority algorithms. 81
Figure 4.4 – Schedulability rate of preemptive fixed-priority algorithms. . . 81
Figure 5.1 – Global Mechanism of DIP. 84
Figure 5.2 – Number of cache misses as a function of partition size for `kmeans` 88
Figure 5.3 – Number of cache misses as a function of partition size for `pca` 89
Figure 5.4 – Number of cache misses as a function of partition size for `spc` 89
Figure 5.5 – Number of cache misses as a function of partition size for `svd3` 90
Figure 5.6 – Number of cache misses as a function of the number of ways 91
Figure 5.7 – Variation of the value of *psel* during execution 92
Figure 5.8 – Number of cache misses as a function of the size of *psel* . . . 93
Figure 6.1 – Relation of WCET and cache partition size for `multi_ncut`. . 100
Figure 6.2 – Relation of WCET and cache partition size for `sift`. 100
Figure 6.3 – Relation of cache partition size and maximum WCET for `lda` 101
Figure 6.4 – Relation of cache partition size and maximum WCET for `lda`. 102
Figure 6.5 – Schedulability analysis for all benchmarks. 103

LIST OF TABLES

Table 2.1 – Summary of notations related to periodic real-time task model 56
Table 4.1 – Benchmark categorization considering the percentage of cache misses. 76
Table 4.2 – Parameters of the considered processors. 76
Table 4.3 – Parameters of schedulability experiments. 80
Table 5.1 – Benchmarks used in the experiments. 86
Table 5.2 – Cache parameters applied in the experiments. 87
Table 6.1 – Benchmarks used in the experiments. 98
Table 6.2 – Cache parameters applied in the experiments. 99

LIST OF ABBERVIATIONS

AI	Artificial Inteligence
BIP	Bimodal Insertion Policy
CPS	Cyber-Physical System
DIP	Dynamic Insertion Policy
DM	Deadline Monotonic
EDF	Earliest Deadline First
FIFO	First-In First-Out
HD	Hard Drive
HRT	Hard Real-Time
IoT	Internet of Things
LIP	LRU Insertion Policy
LLC	Last Level of Cache
LRU	Least Recently Used
MRU	Most Recntly Used
OCPA	Optimal Cache Partitioning Algorithm
PLRU	Pseudo Least Recently Used
RAM	Random Access Memory
RM	Rate Monotonic
RMS	Rate Monotinic Server
RTS	Real-Time System
SoC	System-on-Chip
SRT	Soft Real-Time
UAV	Unmanned Aerial Vehicles
WCET	Worst Case Execution Time

LIST OF SYMBOLS

τ	A task set
T_i	The ith task of τ
$J_{i,j}$	The jth job of the task T_i
e_i	The execution time of T_i
p_i	The period of T_i
d_i	The relative deadline of T_i
$r_{i,j}$	The relative deadline of T_i

CONTENTS

1 INTRODUCTION . . . 29
1.1 PROBLEM OVERVIEW . . . 31
1.2 GOALS . . . 33
1.3 DOCUMENT ORGANIZATION . . . 34
2 BACKGROUND . . . 35
2.1 MEMORY ARCHITECTURE . . . 35
2.1.1 Memory Hierarchy . . . 36
2.1.2 Temporal and Spatial Locality . . . 37
2.1.3 Cache Memory Organization . . . 38
2.1.3.1 Levels of Cache Memory . . . 39
2.1.3.2 Performance . . . 40
2.1.3.3 Mapping . . . 41
2.1.3.4 Cache Partitioning . . . 44
2.1.4 Cache Replacement Policies . . . 45
2.1.4.1 RANDOM . . . 45
2.1.4.2 First-In, First-Out (FIFO) . . . 46
2.1.4.3 Lest Recently Used (LRU) . . . 46
2.1.4.4 Pseudo-LRU (PLRU) . . . 49
2.1.5 Adaptive Insertion Policies . . . 49
2.1.5.1 LRU Insertion Policy (LIP) . . . 51
2.1.5.2 Bimodal Insertion Policy (BIP) . . . 51
2.1.5.3 Dynamic Insertion Policy (DIP) . . . 53
2.2 REAL-TIME SYSTEM . . . 53
2.2.1 Real-Time Systems Classification . . . 55
2.2.2 Notations Definitions and Task Models . . . 55
2.2.3 Real-Time Scheduling Algorithm . . . 56
2.2.3.1 Fixed-Priority Scheduling . . . 58
2.2.3.2 Dynamic-Priority Scheduling . . . 59
2.2.3.3 Resource Reservation-based Scheduling . . . 61
2.2.4 Real-Time Systems and Cache-Related Parameters . . . 61
2.3 SUMMARY . . . 62
3 RELATED WORK . . . 65
3.1 CACHE REPLACEMENT POLICIES FOR REAL TIME SYSTEMS . . . 65
3.2 CACHE PARTITIONING . . . 68
3.3 SCRATCHPAD MEMORIES IN HARD REAL-TIME SYSTEMS . . . 69
3.4 PARTIAL CONSIDERATIONS . . . 71
4 EVALUATION OF CACHE REPLACEMENT POLICIES . . . 73

4.1 CACHEGRIND: PROFILING CACHE BEHAVIOR IN DYNAMIC ANALYSIS . 73
4.2 CATEGORIZATION OF BENCHMARKS AND EVALUATION OF REPLACEMENT POLICIES . . . 74
4.3 SCHEDULABILITY IMPACT ASSESSMENT . . . 77
4.4 PARTIAL CONSIDERATIONS . . . 81
5 IMPLEMENTATION AND EVALUATION OF THE DIP POLICY IN CACHEGRIND 83
5.1 IMPLEMENTATION OF DIP . . . 83
5.2 PARAMETERS OF EXPERIMENTS . . . 86
5.3 ASSESSMENT OF THE DIP POLICY IMPLEMENTED IN CACHEGRIND . . . 86
5.4 PARTIAL CONSIDERATIONS . . . 91
6 CACHE OPTIMIZATION TECHNIQUES . . . 95
6.1 OVERVIEW OF OPTIMAL CACHE PARTITION ALGORITHM . . . 96
6.2 THE PROPOSED ALGORITHM . . . 96
6.3 EXPERIMENTAL COMPARISON . . . 98
6.3.1 First Experiment: WCET x Cache Partition Size . . . 99
6.3.2 Second Experiment: cache partition size x maximum WCET . . . 101
6.3.3 Third Experiment: Schedulability . . . 101
6.4 PARTIAL CONSIDERATIONS . . . 103
7 CONCLUSION . . . 105
7.1 SUMMARY OF CONTRIBUTIONS . . . 105
7.1.1 Evaluation of Cache Replacement Policies . . . 105
7.1.2 Implementation and Evaluation of the DIP Policy in Cachegrind . . 106
7.1.3 Cache Optimization Techniques . . . 106
7.2 CLOSING REMARKS . . . 107
7.3 FUTURE WORKS . . . 108
BIBLIOGRAPHY . . . 109

1 INTRODUCTION

In the era of Industry 4.0 and the widespread adoption of the Internet of Things (IoT), Cyber-Physical Systems (CPS) have become increasingly prevalent in modern society. These systems, known as hard CPS, are characterized by their strict constraints on predictability and confidentiality (AYDIN; JOHANSSON; SASTRY, 2018). They find extensive application in critical domains such as avionics systems, supervisory systems for nuclear power plants, satellite orbit controllers, life support devices in space stations, unmanned aerial vehicles (UAVs) , flight controllers, surgeon-machine interfaces in robotic surgery devices, and autonomous vehicles, among others.

As technology advances, CPS is entering a new phase with high expectations for their capabilities, particularly in the realm of artificial intelligence (AI). Modern CPS are becoming more intelligent, which demands greater computational power and increased complexity in both software and hardware. However, this complexity comes at a cost — a fundamental lack of predictability in CPS behavior has emerged as a significant challenge. (LIU, 2000; BUTTAZZO, 2011; GRACIOLI et al., 2019)

To improve system performance in typical scenarios, hardware designers have adopted various techniques such as caches, multilevel and superscalar pipelines, and prefetchers. While these techniques have demonstrated performance enhancements, they have also introduced variability in instruction execution times due to potential accidents and associated time penalties (REINEKE et al., 2007). Among these hardware components, cache memory architecture plays a pivotal role in determining medium and worst-case performance, primarily due to the substantial penalties associated with cache misses (GRACIOLI et al., 2015). In cases where a cache line needs to be replaced and returned to the main memory before being filled, cache misses can result in cycles exceeding 50 or more, presenting a hundredfold increase compared to cache hits (Liedtke; Hartig; Hohmuth, 1997). Several properties of cache memories, including associativity and replacement policies, significantly influence predictability, with the replacement policy emerging as one of the key factors impacting cache memory behavior predictability. (Heckmann et al., 2003)

Traditionally, the Least Recently Used (LRU) policy and its approximations have been the preferred choice for line replacement policies in processor chip caches for several decades (QURESHI et al., 2006; MANCUSO; YUN; PUAUT, 2019). While LRU performs well in workloads with high locality, it can exhibit pathological behavior when confronted with high memory demands that surpass the available cache size (QURESHI et al., 2007). In cases where a real-time application's

working set slightly exceeds the cache size, alternative policies such as RANDOM may experience fewer cache misses compared to LRU (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004; REINEKE et al., 2007; QURESHI; SRINIVASAN; RIVERS, 2007).

The diverse range of critical modern real-time systems mentioned earlier exhibits varying and often conflicting memory requirements (GRACIOLI et al., 2019). For instance, in autonomous vehicles, simple control tasks like airbags and ABS braking systems have low memory demands, while more complex tasks like pedestrian detection and sensory fusion necessitate substantial processing power and memory resources. In such scenarios, each task's performance can be maximized by selecting the most appropriate cache line replacement policy—one that minimizes cache misses—for each specific case. The integration of a cache architecture with a partitioning mechanism, isolating individual tasks at the cache level (GRACIOLI et al., 2015), and dynamically selecting the most suitable replacement policy within each partition holds the potential to reduce the system's worst-case execution time (WCET) and increase schedulability.

Several research studies have explored the benefits of varying cache line replacement policies. Quereshi et al. introduced the Bimodal Insertion Policy (BIP) as an alternative to LRU, demonstrating its superior performance. They further proposed the Dynamic Insertion Policy (DIP), which dynamically selects between BIP and LRU based on the lowest number of cache misses for a given task, resulting in a 21% reduction in cache misses compared to the second-best policy (QURESHI et al., 2007). Subramanian, Smaragdakis, and Loh (2006) proposed a hybrid cache line replacement scheme that dynamically selects between LRU and other policies in hardware without software interference (SUBRAMANIAN; SMARAGDAKIS; LOH, 2006). Mancuso, Yun, and Puaut (2019) introduced DM-LRU (Deterministic Memory - Least Recently Used), applying LRU to deterministic-classified memory regions evaluated through a simulated environment (MANCUSO; YUN; PUAUT, 2019). The DIP policy (QURESHI et al., 2007) opened the door to dynamic approaches for selecting the best cache line replacement policy at runtime (QURESHI et al., 2007).

Additionally, the papers by (ALTMEYER et al., 2014) and (SUN et al., 2023) have made significant contributions to the field. These studies have investigated cache partitioning techniques and cache usage optimization in the context of real-time systems. By applying fixed-priority scheduling and cache partitioning, these papers address the challenges of ensuring mixed-criticality tasks meet their timing requirements while efficiently utilizing cache resources. The outcomes of these studies provide valuable insights into cache management strategies for real-time systems, complementing the research efforts in cache line replacement policy optimization.

Building upon these foundations, the present research aims to address the challenge of optimizing cache line replacement policies and configuration parameters for different tasks in real-time CPS. We propose a novel algorithm that systematically explores the design space to determine the optimal cache settings for each task, taking into account their specific memory requirements and performance objectives. Leveraging the cache partitioning mechanism, our algorithm dynamically selects the most appropriate replacement policy within each partition, with the primary objective of reducing WCET and enhancing the system's schedulability.

In summary, this dissertation aims to bridge the existing gap in the literature by providing an algorithm that explores the design space to determine optimal cache settings for each task in critical CPS. By optimizing cache line replacement policies and configuration parameters, our research endeavors to improve predictability, reduce WCET, and enhance the schedulability of these critical systems. The insights gained from the studies on cache partitioning and cache usage minimization in real-time systems further enrich our understanding of cache management strategies.

1.1 PROBLEM OVERVIEW

Tasks and algorithms exhibit different memory access patterns depending on the organization of their data. However, modern caches typically have a single line replacement policy predetermined during the System-on-Chip (SoC) design phase. This one-size-fits-all policy is often determined based on average performance metrics. However, as depicted in Figure 1.1, independent tasks demonstrate varying behavior depending on the chosen replacement policy.

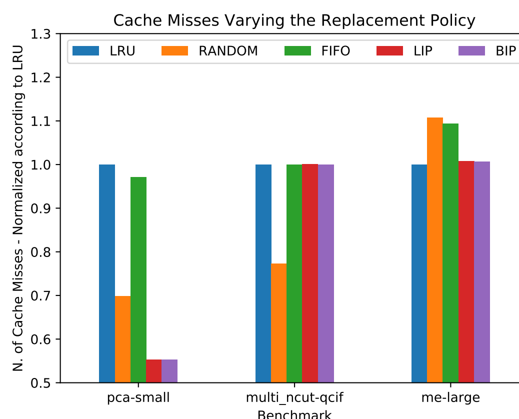


Figure 1.1 – Number of cache misses from different benchmarks varying cache override policy. Normalized according to LRU. 128 KB cache size with 8 ways.

Figure 1.1 showcases the cache miss counts for five different cache line replacement policies (LRU, RANDOM, FIFO - First-In First-Out, LIP - LRU Insertion Policy and BIP) obtained by running three benchmarks from the Cortex Benchmark Suite (THOMAS et al., 2014) on top of cachegrind, a cache profiling tool. The cache configuration used in this scenario includes a size of 128 KB, 8 ways, and a cache line of 64 bytes. The results are normalized with respect to LRU performance, meaning that a bar lower than the LRU bar indicates fewer cache misses for that policy.

For instance, the `pca` benchmark exhibits significantly fewer cache misses under the RANDOM, LIP, and BIP policies compared to LRU and FIFO. However, when running the `me` benchmark, these policies result in a higher number of cache misses compared to LRU. Figure 1.1 demonstrates that in this case, the LRU policy is preferable. Even the performance benefits offered by each policy may vary depending on the benchmark. This is evident in the `multi_ncut-qcif` benchmark, where RANDOM is the only policy with better performance than LRU, albeit with a smaller gain than in the `pca` benchmark.

This example, while simple, highlights the sub-optimality of selecting a single line replacement policy for all cases, as it can lead to increased task execution time. Therefore, it is believed that a unique and flexible architecture, allowing the selection of the replacement policy per task, as well as a mechanism for choosing the best cache parameters, can reduce the execution time of each task and improve the schedulability of the entire system.

In a recent collaboration between the Laboratory of Software/Hardware Integration (LISHA) at UFSC and the cyber-physical systems group at the Technical University of Munich, Germany, a modular and configurable cache architecture was proposed for a RISC-V processor (HOORNAERT et al., 2021). This architecture enables cache partitioning on a per-task basis and the application of different line replacement policies within each partition. Additionally, it allows the selection of cache parameters, such as line length and the number of ways, at compile time. However, existing literature lacks a mechanism that chooses the best cache configuration parameters for each real-time task.

The absence of a mechanism for selecting the optimal cache configuration parameters hinders the full potential of cache performance optimization in real-time systems. Addressing this gap is the objective of the present research, aiming to provide a solution that determines the most suitable cache configuration parameters for each real-time task, considering their unique memory requirements and performance goals. By integrating this mechanism with the modular and configurable cache architecture, we strive to enhance the overall efficiency and schedulability of real-time systems.

1.2 GOALS

Cache line replacement policies play a critical role in ensuring cache predictability and complying with time constraints, as cache misses and miss penalties significantly impact system performance. However, the effectiveness of each policy varies depending on the specific task it is applied to. Therefore, a cache architecture that allows the selection of the most suitable policy at run-time, resulting in fewer cache misses, can greatly improve worst-case execution time (WCET) and schedulability in critical real-time systems.

The main objective of this research is to analyze and optimize cache-related parameters, such as partition size, number of ways, cache line size, and the replacement policy for each partition, in order to enhance the schedulability of critical real-time systems. To achieve this overarching objective, the following specific objectives have been defined:

1. Conduct a comprehensive literature review of related works to gather insights and knowledge in the field of cache line replacement policies and their impact on real-time systems.

2. Evaluate the performance of popular cache line replacement policies, namely LRU, FIFO, RANDOM, LIP, and BIP, in terms of cache misses and the scalability of real-time tasks using a diverse set of benchmarks.

3. Implement and assess the performance of the Dynamic Insertion Policy (DIP) in a cache simulator, specifically cachegrind, to determine its effectiveness in reducing cache misses.

4. Investigate and analyze the effects of different cache settings on the performance of cache line replacement policies, with a focus on their impact on critical real-time systems.

5. Develop an algorithm that optimizes cache parameters and replacement policies, considering the cache partitioning approach for tasks in a real-time system.

By addressing these specific objectives, this research aims to contribute to the advancement of cache optimization techniques for critical real-time systems. The ultimate goal is to propose an algorithm that selects the most suitable cache parameters and replacement policies for each task, considering their individual requirements, in order to improve system schedulability and overall performance.

1.3 DOCUMENT ORGANIZATION

The remainder of this document is structured as follows:

Chapter 2 provides the necessary theoretical background for the development of this research. It covers key concepts and metrics related to cache memory organization and performance 2.1.3, as well as the definition, classification, and mechanisms of Real-Time Systems 2.2.

In **Chapter 3**, a comprehensive survey is conducted on relevant works in the field that align with the scope of this research. This chapter is divided into three main topics: Cache Partitioning Mechanisms 3.2, Cache Line Replacement Policies for Real-Time Systems 3.1, and Memories Scratchpads 3.3.

Chapter 4 focuses on analyzing the impact of different cache line replacement policies, namely LRU, FIFO, RANDOM, LIP, and BIP, in terms of cache misses and the schedulability of real-time tasks. This analysis is performed using a set of benchmarks on a cache architecture that supports per-task partitioning.

In **Chapter 5**, the Dynamic Insertion Policy (DIP) mechanism implemented in the cachegrind simulator is presented. This chapter discusses the simulations conducted, the benchmarks utilized, and provides a detailed discussion of the obtained results.

Chapter 6 introduces an algorithm inspired at (ALTMEYER et al., 2014). The proposed algorithm aims to select the optimal cache line replacement policy for task partitioned caches. The evaluation of this algorithm involves comparing the reduction of WCET, improvement of partition costs, and enhancement of schedulability.

Finally, **Chapter 7** concludes the research, summarizing the main findings, discussing their implications, and highlighting potential avenues for future work in this field.

2 BACKGROUND

This chapter provides a concise overview of the essential background knowledge required for this research. It begins by presenting the fundamental concepts of memory architecture, hierarchy and performance. This includes an exploration of cache memory and its role in improving system performance by reducing memory access latency. The chapter specifically focuses on cache line replacement policies and their behaviors in different scenarios. The aim is to understand how these policies impact cache performance and predictability.

The next section delves into the Real-Time Systems (RTS). It discusses the definition and classification of RTS and highlights their unique characteristics and requirements. The chapter examines the challenges and considerations involved in designing and analyzing RTS, such as meeting strict timing constraints and ensuring deterministic behavior.

By providing this foundational knowledge, the chapter sets the stage for the subsequent chapters, which explore cache partitioning mechanisms, cache line replacement policies for real-time systems, and other related topics. This background understanding is crucial for comprehending the research's objectives and methodology, and for grasping the significance of the findings and recommendations presented throughout the document.

2.1 MEMORY ARCHITECTURE

Over the years, computer systems have made significant advancements, particularly in the field of processors, leading to improved efficiency. However, the same level of progress has not been achieved in memory technology, especially concerning speed (PATTERSON; HENNESSY, 2013). This discrepancy has resulted in a performance gap, where high-speed processors encounter delays when accessing and transferring data to or from main memory.

As computerized systems are applied to increasingly complex projects, the demands for memory performance and size have also grown. This has led to the emergence of different memory specifications with varying construction technologies, often associated with higher production costs (STALLINGS, 2009). These memory-related requirements pose complex challenges in the design of practical systems. While larger and faster memory may be desired, the cost implications must also be taken into account (TANENBAUM, 2006).

Consequently, the development of a memory hierarchy has become imperative. This hierarchy employs multiple components or technologies to meet the diverse performance needs of systems (HENNESSY; PATTERSON, 2011).

This section provides an overview of the memory hierarchy, with a spe-

cific focus on cache memory and cache replacement policies. Understanding these concepts is essential for the development of this research. The memory hierarchy plays a crucial role in bridging the speed gap between processors and main memory, improving overall system performance. Cache memory, as a key component of the memory hierarchy, operates as a high-speed buffer between the processor and main memory, storing frequently accessed data to minimize access latency. Cache replacement policies determine which cache lines are evicted when new data is fetched, and different policies exhibit varying behaviors in different scenarios. A comprehensive understanding of these concepts is necessary for investigating cache partitioning mechanisms and optimizing cache parameters to enhance the schedulability of real-time systems.

2.1.1 Memory Hierarchy

In 1946, Arthur Burks, Herman Goldstine and John von Neumann published preliminary discussions on the logical design of an electronic computing instrument. Even at that early stage, they recognized the critical role of memory capacity in the design of general-purpose systems and concluded that the use of different types of memories to meet varying usage demands was becoming increasingly necessary (BURKS; GOLDSTINE; NEUMANN, 1946).

The memory hierarchy was subsequently devised as a mechanism to address system resource requirements by incorporating memory levels, each of which is smaller, faster, and more expensive than the next level, moving farther away from the processor. Figure 2.1 illustrates the organization of the memory hierarchy. At the top of the hierarchy, within the triangle, are the registers and caches, which are implemented in closer proximity to the processor. Below the triangle, in the box, are external memory devices such as USB drives and hard disks (HD). As we move up the hierarchy, the devices take less time to access information (lower latency), but they come at a higher cost and offer a smaller storage capacity. Conversely, devices lower in the memory hierarchy are more cost-effective, provide larger storage capacity, but exhibit higher latency.

The memory hierarchy's goal is to optimize the overall performance of computer systems by strategically utilizing different memory levels. The registers, which are the fastest and most expensive memory units, are located directly within the processor and provide rapid access to data. Caches, which are smaller and faster than main memory, serve as intermediate storage between the registers and the main memory. Caches aim to store frequently accessed data to minimize the latency associated with accessing data from the larger and slower main memory. Main memory, also known as RAM (Random Access Memory), serves as a larger storage unit but has a higher latency compared

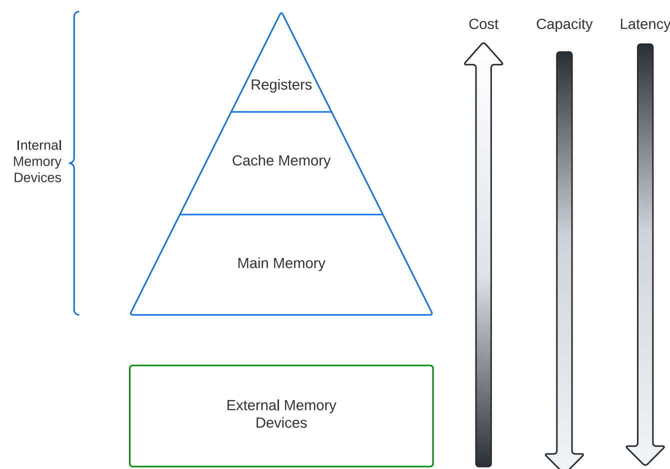


Figure 2.1 – Memory hierarchy structure.
Adapted from (STALLINGS, 2009)

to caches. External memory devices, such as pen drives and external HDs, are even larger in capacity but come with longer access times.

By employing the memory hierarchy, computer systems can optimize the use of resources, balancing speed, capacity, and cost, to deliver efficient and cost-effective performance. The hierarchical arrangement ensures that frequently accessed data is readily available in the faster memory levels, reducing the overall latency of memory operations.

2.1.2 Temporal and Spatial Locality

In the 1960s, IBM researchers made a significant discovery that revolutionized computer performance. They observed that a large portion of code exhibits repetitive patterns, which can be leveraged to enhance system efficiency. By storing frequently accessed sections of code in a small, high-speed memory, the impact of wait states can be minimized, relegating slower and less expensive memory to the less repetitive parts of the program (HANDY, 1998). This principle forms the basis of cache memory operation, which will be discussed in detail in Section 2.1.3. The concept of code repetition can be viewed from two perspectives: time locality and spatial locality.

Time locality, also known as temporal locality, refers to the tendency of instructions or data to be accessed in close temporal proximity. In other words, recently accessed memory locations are more likely to be accessed again in the near future, compared to those accessed a long time ago. For example, a processor is more likely to revisit a memory location accessed ten cycles ago rather than one accessed ten thousand cycles ago. Exploiting time locality allows for efficient caching and reduces the latency associated with memory

accesses (HANDY, 1998).

Spatial locality, on the other hand, relates to the observation that computer code frequently executes within a small area of memory. This area is not necessarily confined to a single range of main memory addresses; it can be distributed throughout the memory system. In simpler terms, if a particular memory location is accessed, it is highly likely that data stored in nearby memory locations will also be accessed in the near future. By capitalizing on spatial locality, cache systems can prefetch and store adjacent data, reducing the time needed to retrieve it when required (HANDY, 1998).

Both time and spatial locality are fundamental properties of program execution that can significantly impact memory access patterns. Caching mechanisms exploit these locality principles to deliver faster access times and improved overall system performance. By storing frequently used instructions and data closer to the processor, caches minimize the latency associated with retrieving information from slower, larger, and more distant memory levels.

2.1.3 Cache Memory Organization

Cache memories are specialized high-speed storage devices that play a crucial role in improving data transfer between the processor and main memory, as well as other cache levels (JACOB; WANG; NG, 2010). The primary objective of a cache is to enhance system performance by exploiting the principle of temporal locality. By storing recently accessed data, the cache enables quick retrieval when the processor requires it. As a result, the processor first checks the cache for the requested data, and only if it is not found there, does it search for the data in the main memory. By keeping a significant portion of frequently used data in the cache, the average access time can be substantially reduced (TANENBAUM, 2006).

The cache operates based on the principle of a memory hierarchy, as discussed earlier. It consists of a hierarchy of cache levels, with each level being closer to the processor and faster but smaller compared to the levels below it. The cache works by dividing memory into fixed-size blocks, known as cache lines or cache blocks, which are used as the basic unit of data transfer between the cache and main memory. When the processor requests data, the cache checks if the desired data is present in one of its cache lines. If a cache hit occurs, indicating that the data is already in the cache, it is quickly accessed and provided to the processor. On the other hand, if a cache miss occurs, meaning the data is not present in the cache, the cache needs to retrieve the data from the main memory, resulting in higher latency (JACOB; WANG; NG, 2010).

The efficiency of a cache memory is determined by its hit rate, which represents the percentage of data accesses that result in a cache hit. A higher hit rate indicates better cache utilization and overall system performance. Caches employ various techniques, such as associative mapping and replacement policies, to manage the cache lines and optimize the hit rate. These techniques ensure that the most frequently accessed data is retained in the cache, while less frequently used data is replaced to make room for new data (TANENBAUM, 2006).

In summary, cache memories act as high-speed intermediaries between the processor and main memory, leveraging temporal locality to store recently accessed data and minimize the latency associated with accessing the slower main memory. By keeping frequently used data closer to the processor, caches enhance overall system performance and reduce average access times.

2.1.3.1 Levels of Cache Memory

Multilevel caches have become commonplace in modern computer systems to optimize data transfer and improve performance. The cache hierarchy typically consists of three levels: L1, L2, and LLC (Last Level of Cache), each with different capacities and characteristics (STALLINGS, 2009).

The L1 cache, also known as the primary cache, is the fastest but smallest cache level. It is usually integrated into the processor chip and divided into separate data and instruction caches. The L1 cache stores frequently accessed data and instructions to provide quick access for the CPU. Different processors have different sizes of L1 cache, ranging from 8 KB in early Intel processors like the i486 to larger sizes like 32 KB in Intel i7 and even 64 KB in ARM Cortex-A53.

To compensate for the limited capacity of the L1 cache, the L2 cache, or secondary cache, was introduced. The L2 cache has a larger storage capacity compared to the L1 cache. It acts as an intermediary between the L1 cache and the main memory, storing additional data to reduce the need for accessing the slower main memory. The size of the L2 cache varies depending on the processor architecture. For example, Intel i7 processors provide an L2 cache of 256 KB, while ARM processors offer configurable L2 cache sizes of up to 2 MB.

In some systems, an additional cache level, LLC, is implemented. The LLC is a specialized memory designed to improve the performance of the previous cache levels. It serves as a buffer between the processor cores and the main memory, helping to mask the latency associated with main memory access. The LLC is typically larger than the L2 cache and has a significant impact on overall system runtime performance. Its size and configuration vary depending on the specific system design and requirements.

In multicore processors, each core may have its dedicated L1 cache integrated into the processor chip. However, the higher-level caches, such as L2 and LLC, are often shared among multiple cores. To ensure efficient data transfer, these shared caches are connected to the CPU through high-speed alternate system buses, separate from the main system bus. This design allows the shared caches to handle cache traffic without being affected by congestion on the main bus.

Figure 2.2 provides an illustration of a possible memory configuration in a multicore system with four cores and a three-level cache hierarchy. Each core has its dedicated L1 cache. The L2 cache levels are shared, with cores 1 and 2 sharing one L2 cache and cores 3 and 4 sharing the other. Finally, there is an external LLC cache that is shared among all the cores, in this example the LLC is shown as a third level L3. This configuration demonstrates the hierarchical structure of cache memory in a multicore system.

Overall, the multilevel cache architecture with dedicated and shared caches improves system performance by reducing the latency associated with main memory access and providing fast data retrieval for the processor cores.

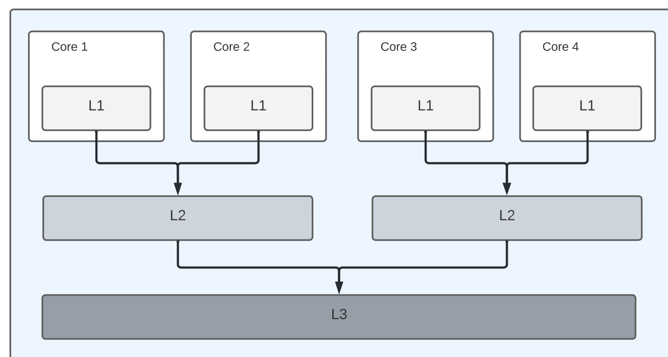


Figure 2.2 – Multicore system architecture with dedicated L1 and L2 caches and shared LLC (L3).

Source: Adapted from (STALLINGS, 2009)

2.1.3.2 Performance

The performance of cache memory can be evaluated based on two key metrics: cache hit ratio and cache miss ratio. Additionally, the miss penalty is another important factor to consider.

A cache hit occurs when the processor successfully finds the required data in the cache. It indicates that the cache memory is functioning effectively, allowing for fast access to frequently accessed data. On the other hand, a cache miss happens when the required data is not found in the cache. In such

cases, the cache needs to fetch the required block from the main memory, resulting in increased latency and performance degradation.

To assess cache performance, two ratios are commonly used:

1. Hit ratio: This is the ratio between the number of cache hits and the total number of cache accesses. It provides a measure of how often the cache is able to satisfy data requests without accessing the main memory. A higher hit ratio indicates a more efficient cache, as a larger proportion of data requests are fulfilled from the cache itself (HENNESSY; PATTERSON, 2011).
2. Miss ratio: This is the ratio between the number of cache misses and the total number of cache accesses. It represents the frequency at which the cache fails to find the required data and needs to fetch it from the main memory. A lower miss ratio indicates better cache performance, as fewer cache accesses result in cache misses (HENNESSY; PATTERSON, 2011).

In addition to hit and miss ratios, the miss penalty is a critical factor in cache performance evaluation. The miss penalty refers to the cost incurred when a cache miss occurs and a cache line needs to be replaced. It includes the time taken to retrieve the required block from the main memory and the subsequent cache line replacement. Minimizing the miss penalty is crucial for improving cache efficiency and reducing overall system latency (PATTERSON; HENNESSY, 2013).

By analyzing the hit ratio, miss ratio, and miss penalty, the performance of a cache can be evaluated, and appropriate optimizations can be applied to enhance cache efficiency and reduce access latency.

2.1.3.3 Mapping

Cache mapping determines the mapping scheme used to allocate memory blocks from the main memory to cache lines. It defines how the cache index is computed from the memory address and how cache lines are selected for replacement in case of a cache miss.

There are three commonly used cache mapping techniques, according to (PATTERSON; HENNESSY, 2013). They are:

1. **Direct Mapping:** In this mapping scheme, each memory block is mapped to a specific cache line based on its memory address. The cache index is calculated using a subset of the address bits. For example, if the cache has 2^n lines, the lower n bits of the memory address are used to determine the cache index. Direct mapping provides a simple and efficient mapping

strategy, but it can lead to a high collision rate when multiple memory blocks are mapped to the same cache line, resulting in frequent cache misses.

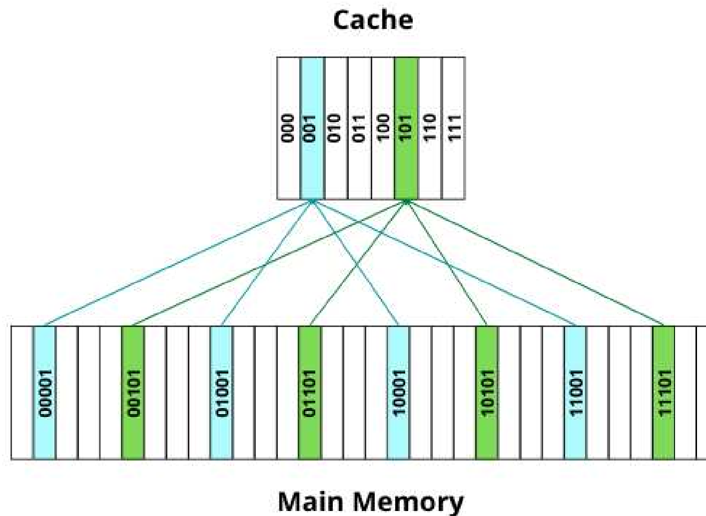


Figure 2.3 – Direct mapping of a memory with 32 blocks into an eight-word cache.

Source: Adapted from (PATTERSON; HENNESSY, 2013)

In Figure 2.3, the least significant 3 bits of the main memory blocks are used as the cache index. Thus, addresses 00001_{bin}, 01001_{bin}, 10001_{bin} and 11001_{bin} are all mapped to cache entry 001_{bin}, while addresses 00101_{bin}, 01101_{bin}, 10101_{bin} and 11101_{bin} are all mapped to the cache entry 101_{bin}.

2. **Set-Associative Mapping:** Set-associative mapping combines the advantages of direct mapping and fully associative mapping. It divides the cache into a set of smaller groups or sets, where each set contains a fixed number of cache lines. Memory blocks can be mapped to any cache line within a specific set. The cache index is calculated using a subset of the address bits that identifies the set. Set-associative mapping reduces the collision rate compared to direct mapping by allowing more flexibility in cache line selection. Common examples include 2-way, 4-way, and 8-way set-associative mapping, where each set contains two, four, or eight cache lines, respectively.

   In Figure 2.4, any memory block can be allocated on any line in the cache memory without any restrictions. The cache index is not derived from the

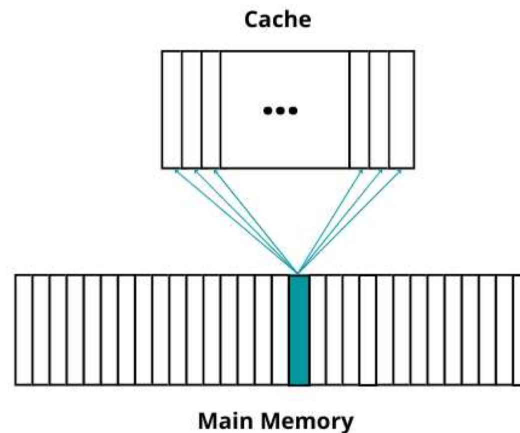


Figure 2.4 – Associative mapping of a block of main memory

Source: Adapted from (PATTERSON; HENNESSY, 2013)

memory address, and the cache controller searches the entire cache to find the requested block.

3. **Fully Associative Mapping:** In fully associative mapping, any memory block can be mapped to any cache line, without any restrictions. This means that the cache index is not derived from the memory address. Instead, the cache controller searches the entire cache to find the requested block. Fully associative mapping provides the highest flexibility and lowest collision rate among all mapping techniques. However, it requires more complex hardware and incurs higher access latency due to the need for a full cache search. Consequently, fully associative mapping is typically used in smaller cache configurations or as an LLC in multilevel cache systems.

   In Figure 2.5, the cache memory is divided into v sets. The main memory block B_0 must be mapped to set 0 and can be allocated to any line in that set. The block B_{v-1} of main memory must be mapped in any line of the set $v-1$ of the cache.

The choice of cache mapping technique greatly influences cache performance, hit rate, and access latency. Different applications and memory access patterns may benefit from different mapping schemes. The selection of an appropriate cache mapping technique involves considering factors such as cache size, associativity, access patterns, and the trade-off between cache hit rate and hardware complexity. Optimizing cache mapping is an important

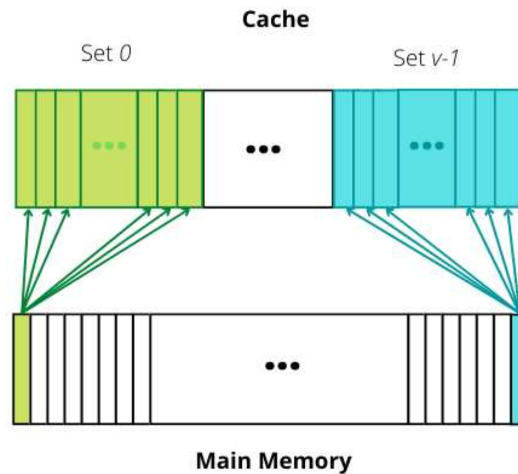


Figure 2.5 – Associative mapping per set.

Source: Adapted from (STALLINGS, 2009)

aspect of cache design and can significantly impact the overall system performance.

2.1.3.4 Cache Partitioning

Cache partitioning is a technique used in real-time systems to allocate a specific portion of the cache to a particular task or core. This helps in isolating task workloads, reducing interference between cores (in multicore systems), improving system predictability, and facilitating Worst-Case Execution Time (WCET) estimation (SANCHEZ; KOZYRAKIS, 2011).

Cache partitioning involves dividing the cache into private and shared spaces. Each task or core is allocated a private space in the cache where it can store its most recently accessed memory blocks. This private space is virtual, meaning that it consists of the N most recently used blocks accessed by the task in each cache set, regardless of their physical location. The actual storage location of these blocks may vary while the task is running. Once the task is completed, the associated private space is released (LESAGE; PUAUT; SEZNEC, 2012).

The shared space in the cache holds blocks that are accessed by all cache-accessing tasks, regardless of whether they have a dedicated private space. It contains the least accessed memory blocks of tasks. The shared space dynamically consists of cache lines that do not belong to any private space. This includes cache lines occupied by task blocks that have no private cache space and cache lines used by a task beyond its allocated capacity of private

space (LESAGE; PUAUT; SEZNEC, 2012).

There are two forms of cache partitioning: index-based and way-based partitioning. In index-based partitioning, partitions are formed by aggregating associative sets in the cache. Each partition consists of multiple cache sets. In way-based partitioning, partitions are formed by aggregating individual cache ways. Each partition consists of multiple cache ways (GRACIOLI et al., 2015).

Different cache partitioning mechanisms have been proposed in the literature to address both types of partitioning. These mechanisms aim to efficiently allocate cache space to tasks, optimize cache utilization, and manage the sharing of cache resources among tasks. The details of cache partitioning mechanisms will be discussed in Section 3.2.

2.1.4 Cache Replacement Policies

Cache replacement policies play a critical role in managing cache utilization and performance. When a cache miss occurs, requiring the retrieval of a block of memory into the cache, a cache line needs to be selected for replacement. This decision becomes particularly important in caches that utilize associative or set associative mapping techniques. The cache line replacement policy dictates which block will be evicted from the cache and where the incoming block will be placed.

Choosing an effective cache replacement policy is essential to maximize cache hit rates, reduce cache thrashing, and improve overall system performance. Various cache replacement policies have been developed, each with its own advantages and trade-offs. In the following subsection, we explore some commonly used cache replacement policies, delve into their characteristics, and examine their impact on cache performance

2.1.4.1 RANDOM

The RANDOM cache replacement policy is the simplest and most straightforward approach among cache line replacement policies. It operates by randomly selecting a cache line for eviction whenever a new block needs to be inserted into the cache. The existing block occupying the randomly chosen position is replaced with the new block.

The RANDOM policy is easy to implement since it does not require any complex tracking or decision-making mechanisms. However, its simplicity comes at the cost of performance. Since the replacement decision is purely random, it disregards the principle of temporal locality, which states that recently accessed blocks are more likely to be accessed again in the near future. As a result, the RANDOM policy tends to lead to suboptimal cache performance.

Empirical studies have demonstrated that the RANDOM policy performs, on average, approximately 22% worse than more advanced policies like LRU in the context of data caches (AL-ZOUBI; MILENKOVIC; MILENKOVIC, 2004). Despite its inferior performance, the RANDOM policy is still employed in certain processors, such as the Cortex A-53. In cases where simplicity and low implementation overhead outweigh the potential performance gains of more sophisticated policies, the RANDOM policy may be a viable choice.

2.1.4.2 First-In, First-Out (FIFO)

The FIFO cache replacement policy operates based on the order of entry of blocks into the cache. Each block is assigned a position in the cache based on the time it was brought in, with the oldest block occupying the LRU position. When a cache miss occurs and a new block needs to be inserted, it is placed in the first position of the cache, pushing all existing blocks toward the LRU position. As a result, the oldest block, which is located in the LRU position, is evicted from the cache.

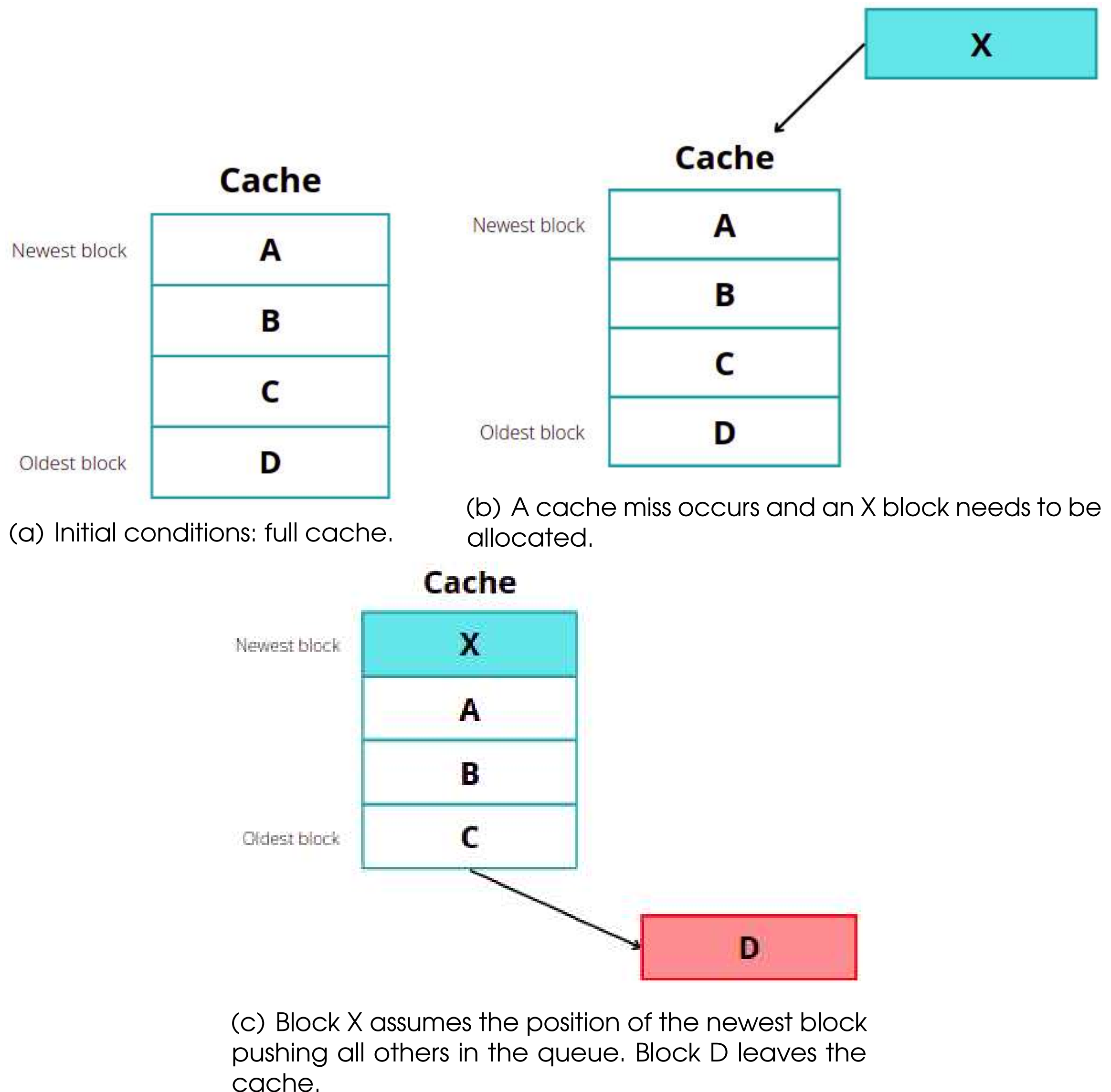
In Figure 2.6, an example of a cache employing the FIFO policy is illustrated. The cache initially contains blocks A, B, C, and D (Figure 2.6(a)). When a cache miss occurs and a new block X is brought in, it is allocated in the most recently used (MRU) position, displacing the existing blocks towards the LRU position. Consequently, the oldest block D, which occupied the LRU position, is evicted from the cache (Figure 2.6(b)).

The FIFO policy ensures a fair and deterministic replacement strategy, as blocks are evicted in the order in which they entered the cache. However, it suffers from the drawback of not considering the access frequency or recency of the blocks. This can lead to poor cache performance in scenarios where certain blocks are repeatedly accessed, while others are rarely used. Despite its simplicity and fairness, the FIFO policy is generally outperformed by more advanced replacement policies, such as LRU, in terms of cache hit rate and overall system performance.

2.1.4.3 Lest Recently Used (LRU)

The Least Recently Used cache replacement policy is a popular and effective strategy for managing cache memory. It is based on the principle of temporal locality, which states that recently accessed data is more likely to be accessed again in the near future. The LRU policy aims to maximize cache hit rates by prioritizing cache lines that have been accessed most recently.

In the LRU policy, each cache line is associated with a timestamp or counter that indicates the order of its access. Whenever a cache hit occurs,



(a) Initial conditions: full cache.

(b) A cache miss occurs and an X block needs to be allocated.

(c) Block X assumes the position of the newest block pushing all others in the queue. Block D leaves the cache.

Figure 2.6 – FIFO replacement policy.

Source: Adapted from (HANDY, 1998)

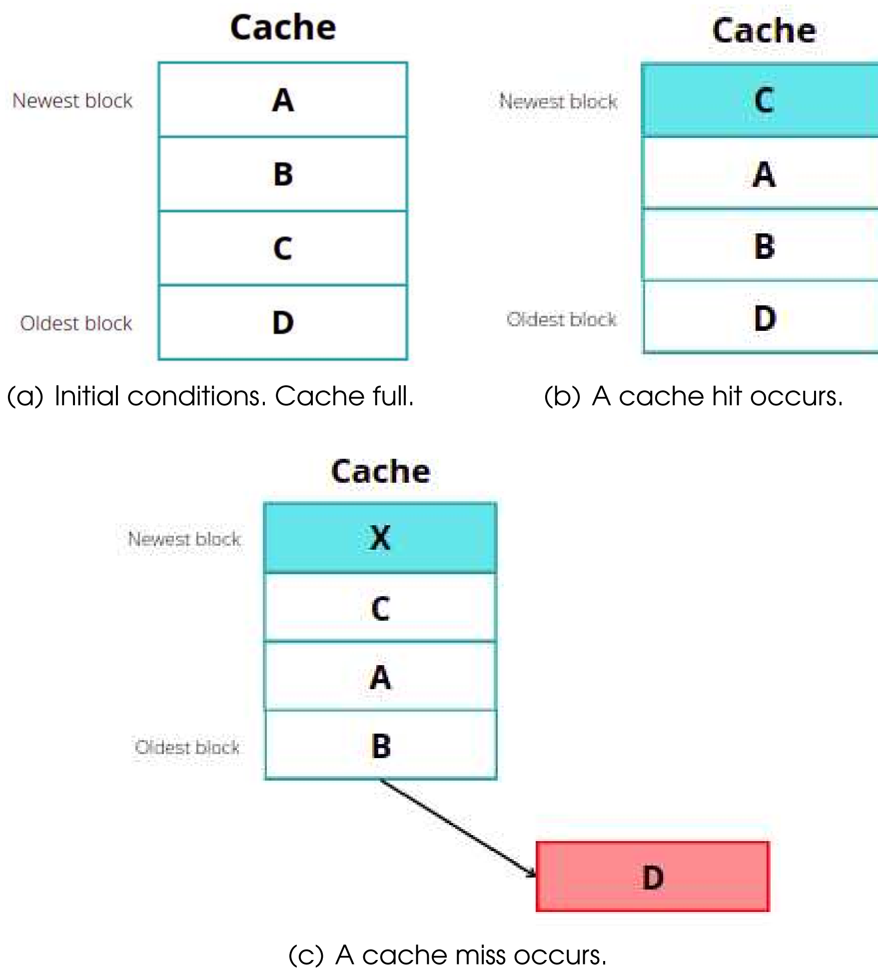
indicating that a requested block is found in the cache, the corresponding cache line's timestamp is updated to the current time, marking it as the most recently used. This ensures that frequently accessed data remains in the cache and can be quickly retrieved when needed.

When a cache miss occurs and a new block needs to be brought into the cache, the cache line with the oldest timestamp, representing the least recently used block, is selected for replacement. The new block is then inserted into the cache at the position of the evicted block.

The significance of the LRU policy lies in its ability to exploit the temporal locality exhibited by many programs. By giving priority to recently accessed data, it maximizes the chances of keeping frequently used data in the cache,

reducing the frequency of cache misses and improving overall system performance. (SMITH, 1982)

Figure 2.7 provides a visual representation of the LRU policy. In Figure 2.7(a), the cache initially contains blocks A, B, C and D, with their corresponding timestamps indicating their order of access, from the most recent to the least recent. In Figure 2.7(b), a cache hit occurs when block C is accessed. As a result, block C's timestamp is updated, moving it to the most recent position. Blocks A and B retain their relative positions. In Figure 2.7(c), a cache miss occurs when block X is requested and not found in the cache. Block X is brought in from main memory and replaces the block with the oldest timestamp, which is block D in this case. The timestamps of blocks A, B and C are updated accordingly.



(a) Initial conditions. Cache full.

(b) A cache hit occurs.

(c) A cache miss occurs.

Figure 2.7 – LRU replacement policy.

Source: Adapted from (HANDY, 1998)

The LRU policy has proven to be highly effective in improving cache performance and reducing cache thrashing. Its ability to prioritize frequently accessed data aligns well with the behavior of many programs, resulting in higher

cache hit rates and overall system efficiency. As a result, the LRU policy is widely used in various computer architectures and systems.

2.1.4.4 Pseudo-LRU (PLRU)

The Pseudo Least Recently Used (Pseudo-LRU or PLRU), also known as tree-LRU, is a widely adopted cache line replacement policy in modern cache memory designs. It has gained popularity and is extensively utilized in commercial products by leading companies like AMD and Intel (Abel; Reineke, 2013).

The Pseudo-LRU algorithm employs a binary tree data structure to track the cache memory state. In this tree, each node represents a cache line and is associated with a bit that can be interpreted as an arrow. The tree's structure allows for efficient determination of the least recently used cache line.

When a cache hit occurs, indicating that the requested data is already present in the cache, the corresponding bits pointing to that memory location are inverted. This inversion ensures that the recently accessed cache line moves closer to the root of the binary tree, indicating its updated usage.

On the other hand, when a cache miss happens and new data needs to be brought into the cache, the line pointed to by the bits is selected for replacement. The new data replaces this line, and the corresponding bits are then inverted, indicating its recent usage.

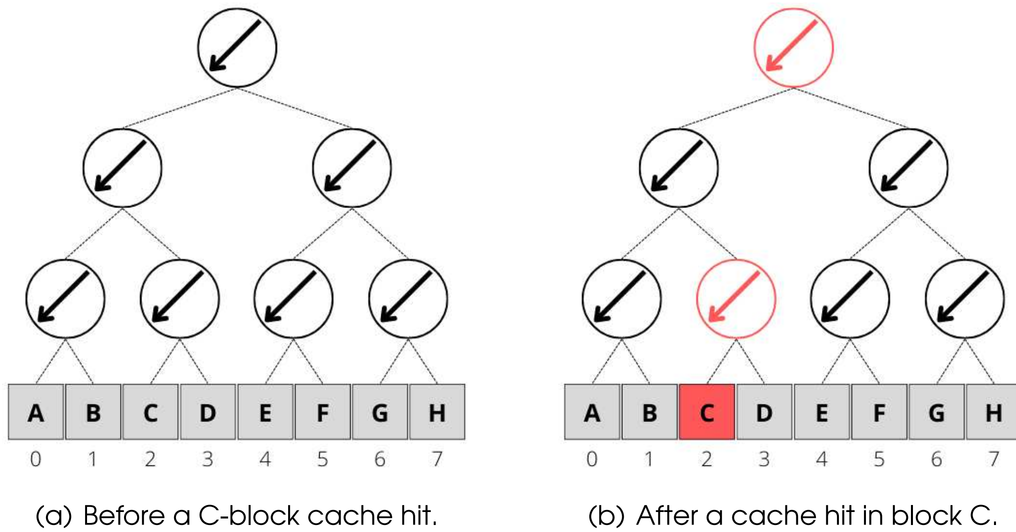
The Pseudo LRU policy leverages the tree-like structure and the bit inversions to approximate the least recently used cache line efficiently. By keeping track of the cache lines' relative recency of use, the Pseudo LRU policy aims to minimize cache misses and maximize cache hit rates, leading to improved performance.

Figure 2.9 provides an illustration of the Pseudo LRU policy in action. In this example, a cache miss occurs when a requested block is not found in the cache. The new data replaces the cache line pointed to by the bits, and the corresponding bits are inverted, indicating the updated cache line usage.

The Pseudo LRU policy has become prevalent due to its effectiveness in managing cache memory and its utilization by major industry players. Its ability to efficiently approximate the least recently used cache line makes it a valuable tool for improving cache performance in a variety of computing systems.

2.1.5 Adaptive Insertion Policies

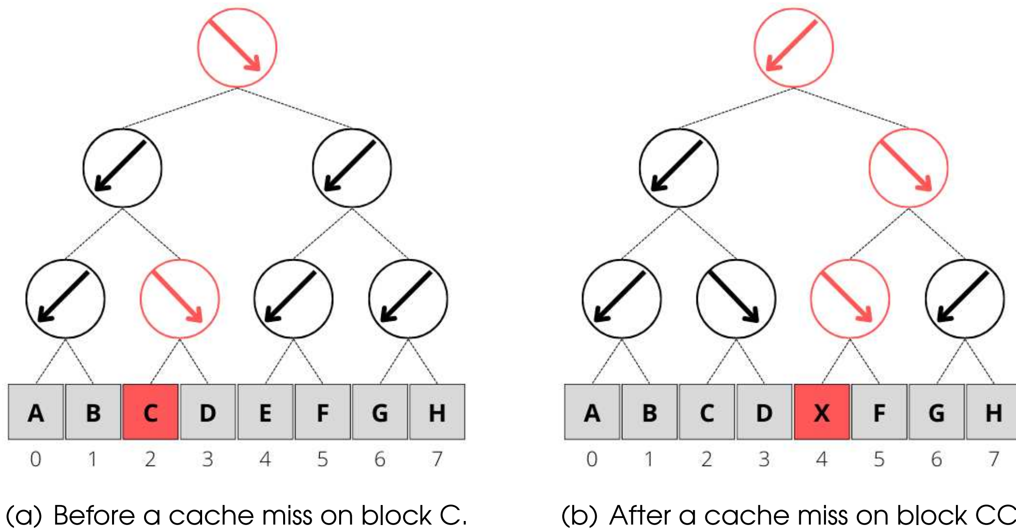
This subsection introduces three adaptive insertion policies, namely LIP (LRU Insertion Policy), BIP (Bimodal Insertion Policy), and DIP (Dynamic Insertion Policy), proposed by Qureshi et al. (QURESHI et al., 2007). These policies are



(a) Before a C-block cache hit. (b) After a cache hit in block C.

Figure 2.8 – How the PLRU policy works when a hit occurs.

Source: Adapted from (HANDY, 1998)



(a) Before a cache miss on block C. (b) After a cache miss on block CC.

Figure 2.9 – How the PLRU policy works when a cache miss occurs.

Source: Adapted from (HANDY, 1998)

based on LRU policy, but aim to reduce cache misses for memory-intensive workloads.

The LRU policy, while widely used, may not be optimal for all workload patterns. The adaptive insertion policies offer greater flexibility and adaptability to improve cache performance in such scenarios.

The LIP, BIP and DIP policies proposed by Qureshi et al. demonstrate their potential in enhancing cache performance for memory-intensive workloads. By deviating from strict LRU ordering and considering factors like insertion time and age ranges, these adaptive insertion policies improve cache hit rates and

effectively reduce the number of cache misses.

2.1.5.1 LRU Insertion Policy (LIP)

The LIP is an adaptive insertion policy that enhances cache performance by allocating all input lines at the LRU position in the cache. Unlike traditional LRU, where cache lines are reordered only based on their access history, LIP introduces the concept of promoting lines from the LRU position to the MRU position when new lines are brought in from the main memory. This adaptive approach aims to address the limitations of LRU and optimize cache performance for memory-intensive workloads.

The key advantage of LIP is its ability to avoid thrashing for workloads with a working set larger than the cache size. By initially placing all input lines at the LRU position, LIP ensures that frequently accessed lines remain in the cache while infrequently accessed lines are more likely to be evicted. This strategy helps maintain a balanced cache content and mitigates the negative impact of excessive cache misses. (QURESHI; SRINIVASAN; RIVERS, 2007)

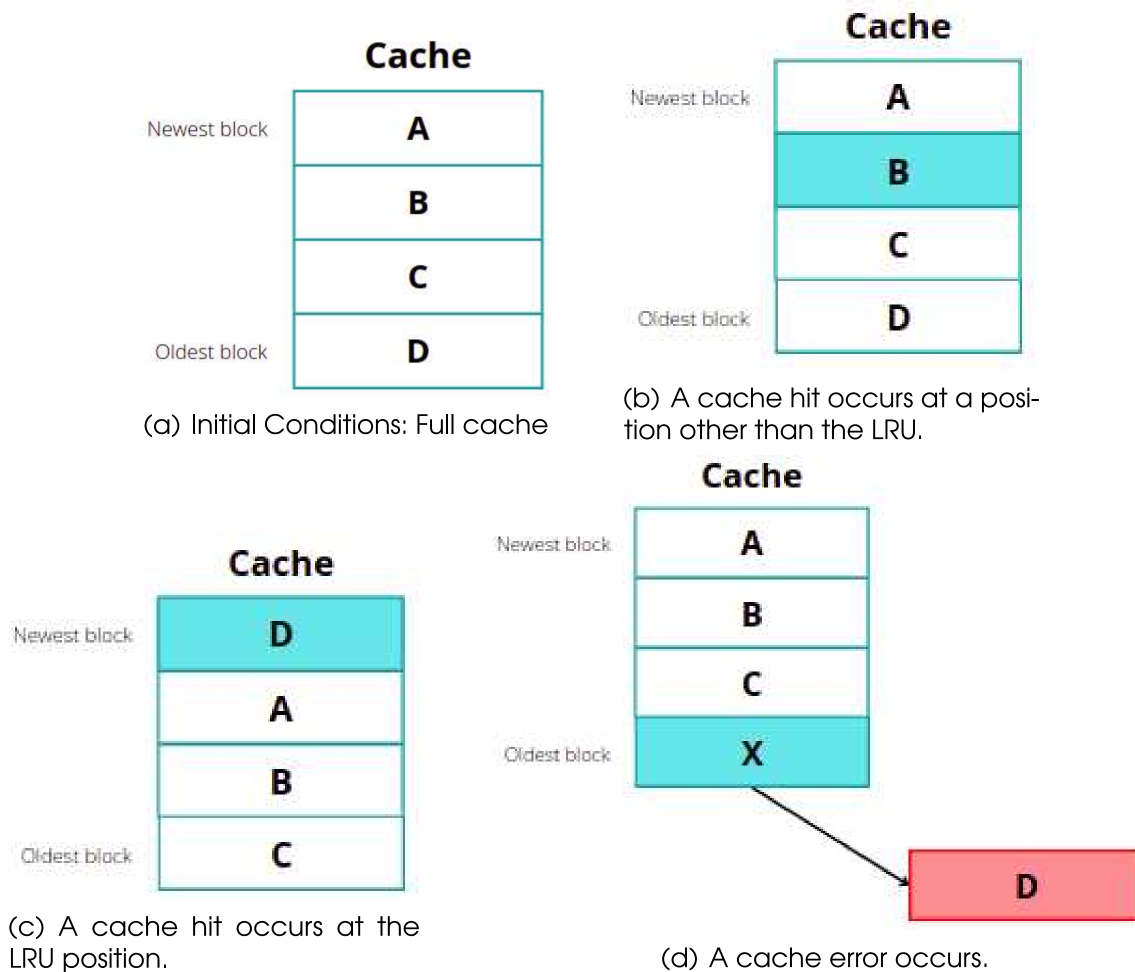
Furthermore, LIP exhibits excellent performance for workloads that exhibit a cyclic access pattern. By promoting recently accessed lines from the LRU position to the MRU position, LIP adapts to the cyclic nature of these workloads and achieves near-optimal hit rates. This adaptive behavior allows LIP to effectively exploit temporal locality and improve cache efficiency.

Figure 2.10 visually demonstrates the behavior of the LIP policy. In the example depicted, the cache initially contains blocks A, B, C and D. When a cache hit occurs in block B, which does not occupy the LRU position, no changes are made to the cache configuration. However, when a cache hit occurs in block C, located at the LRU position, it gets promoted to the MRU position, reflecting its recent access. On the other hand, when a cache miss happens, as shown by the insertion of block X in the LRU position, the block D, which previously occupied that position, is evicted from the cache.

The LIP policy demonstrates its technical importance by providing an adaptive insertion strategy that goes beyond traditional LRU. By considering both the LRU position and the promotion of recently accessed lines, LIP effectively manages cache contents, improves hit rates and reduces cache misses. As a result, it has become a valuable tool in optimizing cache performance for memory-intensive workloads.

2.1.5.2 Bimodal Insertion Policy (BIP)

The BIP is an adaptive insertion policy that builds upon the concepts introduced by the LIP. BIP incorporates a bimodal parameter, denoted as ϵ



(a) Initial Conditions: Full cache

(b) A cache hit occurs at a position other than the LRU.

(c) A cache hit occurs at the LRU position.

(d) A cache error occurs.

Figure 2.10 – LIP adaptive insertion policy.

or the "bimodal throttle parameter," which ranges from 0 to 1. This parameter determines the position at which a new line will be inserted in relation to the LRU position.

The unique feature of BIP is its ability to dynamically adjust the insertion position based on the value of ϵ. When ϵ is set to 0, BIP behaves identically to LIP, allocating all input lines at the LRU position. This ensures the thrashing protection provided by LIP, where frequently accessed lines are retained in the cache while infrequently accessed lines are evicted.

However, as the value of ϵ increases towards 1, BIP gradually transitions towards a behavior resembling the LRU policy. In other words, when ϵ approaches 1, the cache line insertion position becomes closer to the MRU end of the cache. This shift aligns with the LRU principle, where lines that have been accessed more recently are favored over those accessed less recently.

By incorporating this bimodal parameter, BIP introduces a higher degree of flexibility and adaptability compared to LIP. It can dynamically respond to changes in the working set and adjust the cache line insertion position accord-

ingly. This adaptability is crucial in optimizing cache performance for varying workload characteristics, as it allows BIP to strike a balance between thrashing protection and the exploitation of temporal locality.

To summarize, BIP enhances the LIP policy by introducing the bimodal throttle parameter ϵ, which governs the insertion position of new cache lines. It offers a spectrum of behaviors ranging from LIP to LRU, depending on the value of ϵ. This adaptability enables BIP to effectively manage the cache content, protect against thrashing and adapt to changing working set patterns.

2.1.5.3 Dynamic Insertion Policy (DIP)

The DIP is an adaptive insertion policy that dynamically selects between the traditional LRU policy and the BIP based on their respective performance. The key idea behind DIP is to choose the policy that minimizes cache losses, which are estimated during runtime.

DIP maintains runtime estimates of the losses incurred by each of the competing policies, LRU and BIP. These loss estimates provide insights into the effectiveness of each policy in the current workload scenario. The policy with the lower estimated losses is selected for cache line insertions, ensuring that the cache operates with the most suitable policy given the workload characteristics.

By continually monitoring and comparing the loss estimates of LRU and BIP, DIP can dynamically adapt to changes in the workload and select the policy that offers better cache performance. This adaptive behavior allows DIP to leverage the strengths of both policies and optimize cache hit rates.

For more detailed information on the functioning mechanism of the DIP implemented in this work, please refer to Section 5.1 for a comprehensive explanation.

2.2 REAL-TIME SYSTEM

Real-time systems play a critical role in a wide range of domains, including safety-critical applications, embedded systems, telecommunications, industrial automation, medical devices and flight control systems. These systems are specifically designed to respond and control external events with precise timing requirements, where meeting strict timing constraints is as important as producing accurate results (STANKOVIC; RAMAMRITHAM, 1998; BUTTAZZO, 2011; OLIVEIRA, 2020).

Timing is of utmost importance in real-time systems. Unlike traditional computer systems, where the correctness of the computation is the primary concern,

real-time systems prioritize the timely execution of tasks. They are subject to stringent timing requirements that must be met to ensure proper functionality. Failure to meet these timing constraints can result in severe consequences, including financial losses, safety hazards and even loss of human life. (LIU, 2000)

The design and analysis of real-time systems require specialized techniques and methodologies. Researchers and practitioners have developed various scheduling algorithms, resource management strategies and analysis methods to ensure the predictability and reliability of real-time systems in meeting their timing requirements.

Scheduling algorithms play a crucial role in real-time systems, as they determine the order and timing of task executions. These algorithms consider factors such as task priorities, deadlines and resource availability to allocate system resources efficiently. Commonly used scheduling algorithms include Rate-Monotonic (RM), Earliest Deadline First (EDF) and Deadline Monotonic (DM) (LIU; LAYLAND, 1973; BAKER, 2001; SRINIVASAN; BARUAH; GOOSSENS, 2011). Each algorithm has its advantages and limitations, and the choice of scheduling algorithm depends on the specific characteristics and requirements of the real-time system.

Resource management is another vital aspect of real-time systems. Efficient utilization of system resources, such as CPU, memory and I/O devices, is crucial to meet timing constraints. Techniques such as resource reservation, priority inheritance and priority ceiling protocol help manage and allocate resources effectively, avoiding resource contention and ensuring that tasks with higher priority receive the necessary resources to meet their deadlines (SPRUNT; BURNS; BERNAT, 2002).

To assess the timing behavior of real-time systems, various analysis methods and tools have been developed. These methods, such as worst-case execution time analysis and schedulability analysis, provide insights into the system's timing properties and help determine if the system can meet its timing requirements under different scenarios. Furthermore, simulation and testing techniques are employed to validate the real-time system's performance and identify potential timing issues.

In the upcoming sections, we will delve into these techniques and methodologies in greater detail, shedding light on their significance in the development and deployment of real-time systems. By understanding these principles and utilizing appropriate design and analysis techniques, engineers and developers can effectively design, optimize and validate real-time systems to operate with precision, reliability, and safety.

2.2.1 Real-Time Systems Classification

Real-time systems can be classified into two categories based on the consequences of missing timing requirements (OLIVEIRA, 2020):

1. **Hard Real-Time (HRT) Systems:** In HRT systems, missing a timing deadline can lead to catastrophic consequences. These systems are typically found in safety-critical domains such as medical systems, aerospace and nuclear controls, where precise and timely actions are essential for system integrity and human safety. HRT systems require strict adherence to timing constraints and any violation can result in severe financial losses, safety hazards, or even loss of human life.

2. **Soft Real-Time (SRT) Systems:** SRT systems can tolerate occasional timing violations without causing catastrophic failures. While meeting timing deadlines is still important in SRT systems, missing occasional deadlines may result in degraded performance or reduced quality of service. These systems are often found in multimedia applications, audiovisual systems and non-critical control systems. SRT systems focus on providing acceptable results within specific time limits, but small timing deviations can be tolerated without significant consequences.

2.2.2 Notations Definitions and Task Models

In this section, we will define some important notations and models used in the context of real-time systems. These notations will be used throughout the entire work.

The three most popular models of tasks in real-time systems are periodic, sporadic and aperiodic (ABENI; BUTTAZZO; PALOPOLI, 2000; WILHELM; ENGBLOM, 2008; CERVIN; HENRIKSSON; LINCOLN, 2018).

1. **Periodic Task Model:** In the periodic task model, a set of tasks, denoted as τ, is composed of n tasks, $T_1, T_2, ..., T_n$. Each task T_i releases a job every unit of time, referred to as the period p_i. The release time of the j-th job of T_i is denoted as $r_{i,j}$ and is named $J_{i,j}$. Task releases can be triggered by external events such as device interrupts or expiration timers.

2. **Sporadic Task Model:** In the sporadic task model, the period of a task (p_i) represents a lower bound on the separation of tasks, rather than an exact time interval between jobs as in the periodic task model. If we consider the minimum time interval between tasks as the task period, the sporadic task model can be analyzed as the periodic task model.

3. **Aperiodic Task Model:** Unlike periodic and sporadic tasks, aperiodic tasks do not have a period or a minimum interval. They can be released at any time and may have critical deadlines that require special handling through aperiodic task servers.

For the purpose of this work, we will primarily focus on the periodic task model since it is widely used in the literature, well-studied and offers flexibility in real-time systems. Table 1 summarizes the definitions and notations related to the periodic task model used in the subsequent analyses. These definitions and notations will serve as a foundation for our analysis of real-time systems.

Table 2.1 – Summary of notations related to periodic real-time task model

Notation	Description	Definition
τ	A task set.	$\tau = T_1, ..., T_n$
T_i	The ith task of τ	$1 \leq i \leq n$
$J_{i,j}$	The jth job of the task T_i	$j > 1$
e_i	The execution time of T_i	$e_i > 0$
p_i	The period of T_i	$p_i \geq e_i$
d_i	The relative deadline of T_i	$d_i \geq_i$
$r_{i,j}$	The release time of $J_{i,j}$	$r_{i,j} = r_{i,j-1} + p_i$

2.2.3 Real-Time Scheduling Algorithm

In real-time systems, when a single processor needs to handle a set of concurrent tasks that may overlap in time, a scheduling algorithm is required to determine the order in which tasks are executed. This scheduling algorithm follows a predefined criteria, known as the scheduling policy. It defines the rules that govern the assignment of the processor to multiple tasks at any given time (LIU; LAYLAND, 1973).

In the context of scheduling algorithms, several important terms and concepts are used (BARUAH; GOOSSENS, 2005):

- Active Task: A task that can potentially run on the processor regardless of its actual availability.
- Task Ready: A task waiting for the processor to be allocated.
- Task Running: A task that is currently being executed by the processor.
- Ready Queue: A queue that holds all the ready tasks waiting for the processor. Operating systems may have multiple ready queues to handle different categories of tasks.

Preemption is another important concept in scheduling algorithms. It refers to the interruption of a running task when a higher-priority task arrives. The running task is suspended and placed back in the ready queue, allowing the higher-priority task to gain immediate access to the processor (BUTTAZZO, 2011).

Scheduling algorithms can be classified based on their characteristics and activation mechanisms (LIU, 2000; BARUAH; FISHER, 2006; OLIVEIRA, 2020). Two common classifications are:

1. **Time-Driven Schedulers vs. Event-Driven Schedulers:**

    - Time-Driven Schedulers: These algorithms make scheduling decisions at specific time points that are predetermined before the system starts executing. The decisions are based on a fixed schedule.
    - Event-Driven Schedulers: These algorithms make scheduling decisions when specific events occur, such as task releases. Tasks are placed in scheduling queues ordered by their priorities.

2. **Static Schedulers vs. Dynamic Schedulers:**

    - Static Schedulers: These schedulers use a calculated priority assignment during the system design phase. The priorities are assigned offline based on knowledge of release and execution times of all tasks. The priority assigned to a task remains fixed throughout the execution. An example of a static scheduling policy is the Rate Monotonic algorithm, which assigns priorities based on the task's period.
    - Dynamic Schedulers: These schedulers calculate task priorities at runtime using their parameters. The priority is recalculated after each job release. The priority assignment is dynamic and may change during the execution. An example of a dynamic scheduling algorithm is the Earliest Deadline First algorithm, which assigns higher priority to tasks with closer deadlines.

Understanding real-time scheduling algorithms is crucial for analyzing the performance and predictability of real-time systems. These algorithms play a vital role in determining how tasks are scheduled and executed, impacting factors such as efficiency, responsiveness and the ability to meet timing constraints. Over the years, researchers and practitioners have dedicated significant efforts to develop a wide range of scheduling algorithms and techniques tailored to the unique requirements of real-time systems. The following topics will explore

some of the most popular and widely used scheduling algorithms that have emerged from this body of work.

2.2.3.1 Fixed-Priority Scheduling

Fixed-Priority Scheduling is a widely used scheduling approach in real-time systems, where each task is assigned a static priority that does not change during runtime. Therefore, in the context of real-time systems, fixed priority and static scheduling typically refer to the same concept of assigning static priorities to tasks and the terms can be used interchangeably.

In fixed-priority scheduling, tasks are prioritized based on their criticality or timing requirements. The task with the highest priority is executed first, followed by tasks with lower priorities. If multiple tasks have the same priority, the scheduling policy may employ additional rules, such as first-come-first-served or round-robin, to determine the order of execution.

One of the main advantages of fixed-priority scheduling is its simplicity and efficiency. The scheduling decisions can be made statically during the design phase of the system, which eliminates the need for runtime priority calculations. This simplicity enables predictable behavior and facilitates the analysis of the system's timing properties.

Rate Monotonic (RM) scheduling is one of the most commonly used static priority scheduling algorithms in real-time systems. It assigns priorities to tasks based on their periods, where tasks with shorter periods are assigned higher priorities. The fundamental principle of the RM algorithm is that tasks with shorter periods have higher rates of execution and therefore require more frequent access to system resources. By assigning priorities inversely proportional to task periods, the RM algorithm ensures that tasks with tighter timing constraints are scheduled with higher priority.

One of the key advantages of the RM algorithm is its simplicity, as it only requires knowledge of task periods during the design phase. This simplicity allows for efficient implementation and low computational overhead. Moreover, the RM algorithm provides a schedulability guarantee, stating that a set of tasks scheduled using RM will meet all their timing constraints if the total utilization of the system does not exceed a certain threshold, known as the utilization bound (LIU, 2000; BUTTAZZO, 2011).

Several studies have further extended the analysis and understanding of the RM algorithm. For example, researchers have investigated its optimality in different scenarios and explored its limitations in handling task dependencies, sporadic tasks and other practical considerations (BAKER, 2001; SPRUNT; BURNS; BERNAT, 2002). Additionally, variations of the RM algorithm, such as the Deadline

Monotonic (DM) algorithm, have been proposed to accommodate tasks with varying deadlines and provide enhanced scheduling guarantees (LIU; LAYLAND, 1973; SRINIVASAN; BARUAH; GOOSSENS, 2011).

However, it is important to note that the RM algorithm assumes that tasks are independent and have deterministic execution times. Additionally, it assumes that preemption and context switching incur negligible overhead. These assumptions may not hold in all real-time systems and the limitations of the RM algorithm should be carefully considered when applying it in practice.

Despite its limitations, the RM algorithm has been widely adopted and extensively studied due to its simplicity, predictability and efficiency. It serves as a fundamental benchmark for comparing and evaluating other scheduling algorithms in real-time systems (LIU, 2000; BUTTAZZO, 2011).

Fixed-priority scheduling has been extensively studied in the real-time systems' literature. Researchers have developed various analysis techniques and schedulability tests to determine the feasibility of task sets under fixed-priority scheduling (LIU; LAYLAND, 1973; BARUAH; GOOSSENS, 2005; CACCAMO; BUTTAZZO; LIPARI, 2002). Additionally, techniques such as priority inheritance and priority ceiling protocols have been proposed to address issues related to priority inversion and ensure the correctness of the scheduling order (SHA; RAJKUMAR, 2004; BURNS; WELLINGS, 2015).

While fixed-priority scheduling provides simplicity and analyzability, it may not always be suitable for systems with dynamic task sets or highly variable workload. In such cases, dynamic priority scheduling algorithms like EDF can offer more flexibility and responsiveness. The choice between fixed-priority and dynamic-priority scheduling depends on the specific requirements and characteristics of the real-time system.

2.2.3.2 Dynamic-Priority Scheduling

Dynamic-priority scheduling algorithms are widely used in real-time systems to manage task execution and meet stringent timing constraints. These algorithms assign priorities to tasks dynamically during runtime based on specific criteria such as deadlines, execution times, or urgency (ABENI; BUTTAZZO; PALOPOLI, 2000; BUTTAZZO, 2011)

Dynamic-priority scheduling offers flexibility and adaptability, allowing real-time systems to handle varying workloads and dynamic task arrivals. It enables tasks with higher priority or more urgent timing requirements to be scheduled and executed promptly, ensuring timely completion and meeting critical deadlines. The priority assignment in dynamic-priority scheduling algorithms can be based on factors like relative deadlines, execution times, importance, or crit-

icality of tasks (BARUAH; FISHER, 2006).

By dynamically adjusting task priorities, dynamic-priority scheduling algorithms make efficient use of system resources and ensure that tasks with the most urgent timing constraints receive the necessary attention. This approach optimizes the overall system performance, improves task schedulability and reduces the likelihood of deadline misses (LIU; LAYLAND, 1973; OLIVEIRA, 2020).

Dynamic-priority scheduling algorithms require runtime monitoring and adjustment of priorities, which can introduce additional computational overhead. However, the benefits of dynamic-priority scheduling, such as adaptability to changing conditions and the ability to handle dynamic task sets, often outweigh the associated overhead (SRINIVASAN; BARUAH; GOOSSENS, 2011; ANDERSON; BARUAH; KATCHER, 2017).

One prominent example of a dynamic-priority scheduling algorithm is Earliest Deadline First (EDF). It assigns priorities to tasks based on their relative deadlines, where tasks with closer deadlines are assigned higher priorities. The fundamental principle of the EDF algorithm is to prioritize tasks with earlier deadlines to ensure timely execution (ABENI; BUTTAZZO; LIPARI, 2000; BARUAH; GOOSSENS; SRINIVASAN, 2009).

One of the key advantages of the EDF algorithm is its optimality in scheduling uniprocessor systems. EDF is known as an optimal scheduling algorithm because it can schedule any set of feasible tasks if the system is schedulable. This optimality property makes EDF an attractive choice for applications with strict timing requirements (BARUAH; FISHER, 2006).

The EDF algorithm dynamically adjusts the task priorities at runtime, considering the remaining execution time and the relative deadlines of tasks. Whenever a scheduling decision is made, the task with the earliest deadline is selected for execution. This approach ensures that the task with the most urgent timing constraint is always scheduled first, increasing the chances of meeting all deadlines (ABENI; BUTTAZZO; LIPARI, 2003; SRINIVASAN; BARUAH; GOOSSENS, 2011; SPRUNT; BURNS; BERNAT, 2002).

However, it is important to note that the EDF algorithm requires accurate knowledge of task execution times and deadlines. In practice, estimation or measurement techniques are employed to obtain these parameters. Additionally, the EDF algorithm assumes that preemption and context switching can be performed with negligible overhead.

Several studies have explored the theoretical properties and practical aspects of the EDF algorithm. Researchers have analyzed its schedulability conditions, studied its performance in different scenarios and proposed enhancements to handle various system constraints (LEHOCZKY; SHA; DING, 1989; ABENI;

BUTTAZZO; LIPARI, 2003; BARUAH; GOOSSENS; SRINIVASAN, 2009). Furthermore, the EDF algorithm has been extended to handle multiprocessor systems and has served as a basis for developing more advanced scheduling algorithms, such as the earliest-deadline-off-first (EDF-ON) algorithm (ABENI; BUTTAZZO; LIPARI, 2000; ANDERSON; BARUAH; KATCHER, 2017).

In real-time systems, the choice between the EDF and RM algorithms depends on the specific requirements of the application. While EDF offers optimality and flexibility in handling dynamic tasks, RM provides simplicity and efficiency in managing static task sets. Understanding the characteristics and trade-offs of these scheduling algorithms is crucial for selecting the most suitable approach for a given real-time system.

2.2.3.3 Resource Reservation-based Scheduling

Resource reservation-based scheduling is a powerful paradigm for ensuring the timely execution of tasks in real-time systems by allocating dedicated resources to each task (LIU, 2000; BUTTAZZO, 2011; OLIVEIRA, 2020). This approach aims to provide strong guarantees and meet strict timing constraints.

In resource reservation-based scheduling, tasks are assigned fixed and exclusive resources based on their timing requirements (AYAV; BARUAH, 2004). These resources can include CPU time, memory, communication bandwidth, or other system resources. The system reserves and allocates these resources to tasks during their execution, ensuring their availability when needed.

One well-known technique in resource reservation-based scheduling is the Rate-Monotonic Server (RMS) algorithm (BUTTAZZO, 2011). The RMS algorithm combines the principles of fixed-priority scheduling with resource reservation. It assigns priorities to tasks based on their periods, similar to the RM algorithm (LIU, 2000). Additionally, it employs server-based scheduling, where specific resources are dedicated to each task throughout its execution.

By employing resource reservation-based scheduling, real-time systems can achieve enhanced predictability and ensure the timely execution of critical tasks. This approach has been widely applied in safety-critical domains such as aerospace, automotive and industrial control systems.

2.2.4 Real-Time Systems and Cache-Related Parameters

Caches play a vital role in computer systems, offering faster access to frequently accessed data and reducing memory access latency. However, in real-time systems, the presence of caches introduces additional complexities and challenges due to their impact on timing predictability (BURNS; WELLINGS,

2015; CACCAMO; BUTTAZZO; LIPARI, 2002; WILHELM; ENGBLOM, 2008; BLETSAS; SOUDRIS, 2009).

Real-time systems have stringent timing requirements, where meeting deadlines is critical for the system's correct operation. Caches introduce uncertainty in the timing behavior of programs due to their caching and replacement policies. Cache replacement policies determine which cache lines are evicted when the cache is full and a new entry needs to be brought in. The choice of cache replacement policy significantly affects the predictability of cache behavior and, consequently, the timing predictability of real-time systems (BURNS; WELLINGS, 2015).

Traditional cache replacement policies, such as Least Recently Used (LRU), are designed to optimize overall system performance by evicting the least recently used cache lines. However, LRU and similar policies may not be suitable for real-time systems, as they do not prioritize timing predictability. These policies can lead to cache thrashing, where cache lines critical for meeting timing constraints are frequently evicted, resulting in increased cache misses and potentially missed deadlines (BURNS; WELLINGS, 2015).

In addition to cache replacement policies, cache partitioning techniques can be employed in real-time systems to allocate separate cache regions for critical tasks or data. By isolating critical components in dedicated cache partitions, interference from other tasks or processes can be minimized, enhancing timing predictability. Cache partitioning provides better control over cache access and eviction, ensuring that critical data is not displaced by less critical or non-real-time tasks (HOORNAERT et al., 2021).

To optimize cache parameters and replacement policies for real-time systems, thorough analysis of the application's timing requirements and criticality of data is essential. Performance evaluation techniques, such as worst-case execution time analysis and cache-related preemption delay analysis, can be employed to assess the impact of different cache configurations on timing predictability (ALTMEYER et al., 2014).

2.3 SUMMARY

Caches are essential components of computer systems, providing faster access to frequently accessed data and reducing memory access latency. However, their presence in real-time systems introduces complexities and challenges due to their impact on timing predictability. Real-time systems have strict timing requirements, where meeting deadlines is crucial for correct operation. Caches introduce uncertainty in program timing due to caching and replacement policies.

Traditional cache replacement policies, like Least Recently Used (LRU), optimize overall system performance but may not prioritize timing predictability. These policies can lead to cache thrashing, where critical cache lines are frequently evicted, resulting in increased cache misses and potentially missed deadlines. To address this, cache partitioning techniques allocate separate cache regions for critical tasks or data, minimizing interference and enhancing timing predictability.

Optimizing cache parameters and replacement policies for real-time systems requires a thorough analysis of timing requirements and data criticality. Performance evaluation techniques, such as worst-case execution time analysis and cache-related preemption delay analysis, help assess the impact of different cache configurations on timing predictability.

Understanding the relationship between cache replacement policies and real-time system predictability is crucial for reliable operation and meeting timing constraints. By selecting suitable cache replacement policies, employing cache partitioning techniques, and performing thorough analysis, real-time systems can achieve improved timing predictability and effectively support critical real-time applications.

3 RELATED WORK

This chapter provides an overview of relevant research papers that have contributed to the advancement of the subjects explored in this work. The chapter is structured as follows:

Section 3.1 reviews previous studies focusing on cache line replacement policies that have been specifically designed to optimize real-time systems. These papers delve into various aspects of cache management, addressing challenges related to cache line replacement strategies and proposing novel approaches to enhance the predictability and performance of real-time systems.

Section 3.2 presents a compilation of research work pertaining to memory partitioning mechanisms. These papers investigate different techniques and algorithms for partitioning the memory space to optimize resource allocation and utilization in real-time systems. They explore the trade-offs between different memory partitioning schemes and their impact on system performance, determinism and timing guarantees.

Section 3.3 provides an overview of relevant literature focused on scratchpad memories (SPMs). These papers highlight the significance of SPMs in real-time systems and explore their benefits, such as deterministic behavior, reduced cache-related delays, control over data placement and lower power consumption. They discuss various aspects of SPM design, optimization techniques and integration strategies within cache hierarchies to improve the performance and predictability of real-time systems.

By reviewing the findings and insights presented in these papers, this chapter aims to provide a comprehensive understanding of the existing body of knowledge related to cache line replacement policies, memory partitioning mechanisms and the role of scratchpad memories in optimizing real-time systems. This knowledge serves as a foundation for the development of the subsequent chapters and contributes to addressing the specific objectives of this work.

3.1 CACHE REPLACEMENT POLICIES FOR REAL TIME SYSTEMS

Cache replacement policies play a crucial role in determining cache performance. Several studies have focused on evaluating and improving cache replacement policies specifically for real-time systems.

Hardy et al. (2005) investigated the impact of cache-related preemption delays on real-time systems and proposed techniques to mitigate their effects (HARDY; PUAUT; PAUTET, 2005). They analyzed how preemption caused

by cache-related delays affects the schedulability of real-time tasks. The study identified two main factors contributing to cache-related preemption delays: inter-task cache interference and self-interference. They introduced techniques such as static cache partitioning, dynamic cache partitioning and cache locking to reduce these delays and improve the schedulability of real-time systems.

Pautet et al. (2004) explored cache optimization techniques to enhance the performance of real-time systems by minimizing cache-related preemption delays (PAUTET; PUAUT; PELLETIER, 2004). They proposed a technique called Task-Related Cache Partitioning (TRCP), which dynamically partitions the cache based on the behavior of individual tasks. The TRCP approach allocates cache space proportionally to the memory requirements of tasks, reducing cache conflicts and improving cache hit rates. The study demonstrated the effectiveness of TRCP in reducing cache-related preemption delays and improving the predictability of real-time systems.

Zhao et al. (1997) evaluated different cache replacement policies and their impact on real-time performance (ZHAO; MOSSE; KNAG, 1997). They considered policies such as Least Recently Used (LRU), Random and Pseudo-Least Recently Used (PLRU). The study used a set of benchmarks and real-time task sets to analyze the behavior of these policies in terms of cache hit rates and preemption delays. Their results showed that the choice of cache replacement policy significantly affects the performance of real-time systems. While LRU is a widely used policy, it may not always provide the best performance for real-time workloads. The study highlighted the need for tailored cache replacement policies that consider the unique characteristics of real-time systems.

Dobbing et al. (2007) proposed a method to predict the worst-case number of cache misses to improve WCET analysis (DOBBING; DYSTER; GERNDT, 2007). They introduced a cache modeling approach that considers the cache behavior under worst-case conditions. By accurately estimating the number of cache misses, the WCET analysis can provide tighter bounds on the execution time of real-time tasks. The study presented an algorithm to compute the worst-case number of cache misses and evaluated its effectiveness using a set of benchmarks. The results demonstrated the potential of cache modeling in improving the accuracy of WCET analysis.

Reineke et al. (2007) focused on the predictability of cache replacement policies and provided insights into the characteristics and limitations of different policies (REINEKE et al., 2007). They compared policies such as LRU, FIFO and Random in terms of their predictability and the worst-case cache behavior they can cause. The study showed that LRU, while widely used, can lead to unpredictable cache behavior, making it challenging to analyze the worst-case exe-

cution time of tasks. The authors discussed the trade-offs between predictability and performance and highlighted the need for cache replacement policies that offer better predictability without compromising performance.

Reineke et al. (2014) extended their work by introducing the Selfish-LRU policy, which considers preemption-aware caching for improved predictability and performance (REINEKE et al., 2014). The Selfish-LRU policy aims to reduce the interference caused by cache preemption by prioritizing the eviction of cache blocks belonging to preempted tasks. The study evaluated the Selfish-LRU policy against traditional LRU and demonstrated its effectiveness in improving the predictability of cache behavior and reducing the number of preemption-induced cache misses. The results highlighted the importance of preemption-aware caching techniques in real-time systems.

Qureshi et al. (2006) presented a case for MLP (Memory Level Parallelism)-Aware Cache Replacement (QURESHI et al., 2006). They proposed a policy that takes into account the memory access pattern and exploits the MLP characteristics to improve cache performance. The study showed that MLP-aware replacement policies can significantly reduce the number of cache misses and improve the overall system performance. The authors emphasized the importance of considering memory access patterns in cache replacement policies, especially in real-time systems where minimizing cache misses is critical for meeting timing constraints.

Nelissen et al. (2017) conducted a survey and empirical evaluation of cache replacement policies for real-time systems (NELISSEN et al., 2017). They discussed various policies, including LRU variants, pseudo-LRU, Random and First-In-First-Out (FIFO), and evaluated their performance using a set of benchmarks. The study analyzed different aspects such as cache hit rates, cache conflicts, and the predictability of these policies. The results provided valuable insights into the strengths and weaknesses of each policy, aiding the selection and design of cache replacement policies for real-time systems.

The studies reviewed in this section highlight the importance of cache replacement policies in real-time systems. While traditional policies like LRU are widely used, they may not always be the best choice for real-time workloads. Techniques such as dynamic cache partitioning, preemption-aware caching and MLP-aware replacement policies have shown promising results in improving cache utilization, predictability and performance. These findings serve as a foundation for further research and the development of novel cache replacement policies tailored to the requirements of real-time systems.

3.2 CACHE PARTITIONING

Cache partitioning is a technique employed to allocate cache resources among different tasks or task groups in a multi-tasking real-time system. The goal is to ensure each task receives a dedicated portion of the cache, reducing interference and improving the predictability of cache behavior.

Patel et al. (2015) characterized the impact of cache partitioning on real-time performance by considering factors such as cache hit rates, cache conflicts and the schedulability of tasks (PATEL; PERUMALLA, 2015). The study proposed two cache partitioning techniques: Fixed-Size Partitioning (FSP) and Variable-Size Partitioning (VSP). FSP divides the cache into fixed-sized partitions assigned to different tasks, while VSP dynamically adjusts the partition sizes based on task demands. The experiments conducted using a set of real-time benchmarks demonstrated the benefits of cache partitioning in improving task isolation and reducing cache conflicts, leading to better real-time performance.

Sun et al. (2018) investigated cache partitioning for hard real-time systems with mixed-criticality workloads, where tasks with different criticality levels coexist (SUN; FAN; DENG, 2018). They proposed a mixed-criticality cache partitioning technique that guarantees cache space to high-criticality tasks while allowing low-criticality tasks to use the remaining cache space opportunistically. The study analyzed the impact of different partitioning strategies on the cache hit rates and execution times of tasks with varying criticality levels. The results showed that mixed-criticality cache partitioning can effectively utilize cache resources while maintaining the required level of isolation and predictability for high-criticality tasks.

Lesage et al. (2012) introduced PRETI, a partitioned real-time shared cache technique for mixed-criticality real-time systems (LESAGE; PUAUT; SEZNEC, 2012). PRETI aims to provide strong isolation between tasks with different criticality levels by allocating dedicated cache partitions to each criticality level. The study presented an analysis of the worst-case cache behavior of mixed-criticality tasks and demonstrated the effectiveness of PRETI in ensuring predictable cache performance. The experiments conducted on synthetic task sets validated the benefits of PRETI in terms of improved isolation and reduced interference between tasks.

Altmeyer et al. (2014) evaluated the performance of cache partitioning for hard real-time systems, focusing on the impact of partitioning granularity and the allocation strategy on cache utilization and schedulability (ALTMEYER et al., 2014). The study compared different partitioning approaches, including page-level partitioning and block-level partitioning. The results showed that finer-

grained partitioning can lead to better cache utilization but may introduce higher overhead due to cache management. The study provided valuable insights into the trade-offs associated with cache partitioning in real-time systems and highlighted the need for careful consideration of partitioning strategies based on the system requirements.

Brandenburg et al. (2010) proposed a dynamic cache partitioning technique for hard real-time systems based on the system's runtime behavior and the demands of individual tasks (BRANDENBURG; ANDERSON; BARUAH, 2010). The approach dynamically adjusts cache partitions according to the changing requirements of tasks, aiming to maximize the overall system performance. The study presented an analysis of the worst-case cache behavior and the impact of dynamic partitioning on task schedulability. The experiments conducted using synthetic and real-world benchmarks demonstrated the effectiveness of dynamic cache partitioning in improving cache utilization and overall system performance.

Gracioli and Fröhlich proposed a page coloring mechanism able to partition the data for both the application and RTOS (GRACIOLI; FRöHLICH, 2013). The mechanism separates the memory request in two heaps, one that serves application requests and another one that serves OS requests. The work evaluated the impact of cache interference caused by the RTOS into the schedulability of critical tasks.

The studies discussed in this section shed light on cache partitioning techniques for real-time systems. Cache partitioning provides a means to ensure task isolation and predictable cache behavior, especially in multi-tasking environments. Techniques such as fixed-size partitioning, variable-size partitioning, mixed-criticality partitioning and dynamic partitioning offer different trade-offs in terms of cache utilization, overhead and isolation guarantees. These techniques serve as valuable tools for system designers to achieve better cache utilization and improved real-time performance.

3.3 SCRATCHPAD MEMORIES IN HARD REAL-TIME SYSTEMS

Scratchpad memories (SPMs) have gained attention as an alternative to caches in real-time systems. SPMs provide predictable and deterministic memory access times, eliminating the variability and interference associated with caches. This section reviews studies focusing on the utilization and management of scratchpad memories in hard real-time systems.

Venkatasubramanian et al. (2006) presented a dynamic management technique for scratchpad memories in real-time systems (VENKATASUBRAMANIAN et al., 2006). The study proposed a compiler-driven approach to allocate

data objects to the scratchpad memory, aiming to optimize cache utilization and improve the predictability of memory accesses. The technique leverages static analysis and runtime profiling to identify frequently accessed data objects and allocate them to the scratchpad memory. The experiments conducted using real-time benchmarks demonstrated the benefits of the dynamic management of scratchpad memories in reducing memory access latency and improving the overall system performance.

Palem et al. (2003) focused on modeling and optimizing embedded memory architectures, including scratchpad memories (PALEM; KUNDU, 2003). The study provided an in-depth analysis of the benefits and challenges associated with scratchpad memories in real-time systems. It discussed different optimization techniques, such as data placement and scheduling, to exploit the advantages of scratchpad memories and reduce memory access latencies. The authors emphasized the importance of considering both the software and hardware aspects in the design and utilization of scratchpad memories for real-time systems.

Puaut and Pais (2007) conducted a quantitative comparison between scratchpad memories and locked caches in hard real-time systems (PUAUT; PAIS, 2007). The study presented an analytical framework to evaluate the performance and predictability of these memory structures. It considered factors such as memory access latency, cache conflicts and worst-case execution times. The results showed that scratchpad memories can provide better predictability and reduce memory access latencies compared to locked caches. However, the study also highlighted the need for careful consideration of the memory size and data placement techniques to fully exploit the benefits of scratchpad memories.

Chen et al. (2004) proposed a compiler-directed approach for scratchpad memory management in real-time systems with mixed-criticality workloads (CHEN; HAN; BURNS, 2004). The study introduced techniques to allocate scratchpad memory resources based on the criticality levels of tasks, aiming to ensure the isolation and predictability of high-criticality tasks. The experiments conducted using synthetic and real-world benchmarks demonstrated the effectiveness of the proposed approach in guaranteeing the required level of isolation for high-criticality tasks while maximizing the overall system performance.

The studies presented in this section highlight the advantages of scratchpad memories in real-time systems. Scratchpad memories offer deterministic and predictable memory access times, making them suitable for hard real-time applications. Dynamic management techniques, data placement strategies and compiler-directed approaches contribute to optimizing the utilization of

scratchpad memories and improving the overall system performance. These studies provide valuable insights for system designers considering scratchpad memories as an alternative to caches in real-time systems.

3.4 PARTIAL CONSIDERATIONS

In this chapter, we have provided a comprehensive review of the related work on cache-related issues in real-time systems. The discussed papers have contributed to our understanding of cache replacement policies, cache partitioning techniques, and the utilization of scratchpad memories. By examining the different approaches and experimental evaluations presented in these papers, we have gained valuable insights into optimizing cache utilization, improving predictability and enhancing the performance of real-time systems.

The analysis of cache replacement policies has emphasized the significance of selecting appropriate policies tailored to the requirements of real-time systems. The studies have highlighted the trade-offs between cache hit rates, worst-case execution time (WCET) analysis, and predictability. Various policies, such as LRU-based policies, MLP-aware policies and preemption-aware policies, have been proposed to address these trade-offs and improve cache performance. The empirical evaluations and experimental results conducted in these studies have provided guidance for selecting cache replacement policies that suit the characteristics of real-time systems.

Cache partitioning techniques have been investigated to ensure task isolation and predictable cache behavior in multi-tasking real-time systems. The research in this area has focused on achieving performance isolation between tasks with different criticality levels and minimizing interference caused by shared caches. Hardware-based partitioning, software-based partitioning and mixed-criticality cache partitioning approaches have been proposed and evaluated. These techniques offer mechanisms to effectively partition cache resources and optimize cache allocation for individual tasks, thereby improving system predictability and meeting real-time requirements.

Additionally, the utilization of scratchpad memories has been explored as an alternative to traditional caches in hard real-time systems. Scratchpad memories provide deterministic and predictable memory access times, making them suitable for time-critical applications. The studies discussed in this chapter have highlighted dynamic management, data placement, and compiler-directed approaches for utilizing scratchpad memories. These techniques offer insights into efficient allocation of data to scratchpad memories, reducing cache misses, and enhancing overall system performance.

The literature reviewed in this chapter contributes to our understanding of

cache-related challenges and opportunities in the context of real-time systems. The findings provide valuable guidance for researchers and system designers in making informed decisions regarding cache design and optimization strategies. By considering the specific requirements and constraints of real-time systems, further research can build upon the foundations established by these studies, leading to the development of advanced techniques and approaches to address cache-related issues. Ultimately, these advancements will contribute to improving the predictability, performance, and efficiency of real-time systems.

With the completion of this chapter, we have achieved specific objective 1 proposed in Chapter 1, fulfilling the goal of providing a comprehensive review of the related work on cache-related issues in real-time systems.

4 EVALUATION OF CACHE REPLACEMENT POLICIES

In this chapter, we delve into the concepts presented in the motivation section (Section 1.1). Our objective is to evaluate the behavior of different cache line replacement policies across various applications. This evaluation aims to ascertain the impact of a cache architecture that offers task partitioning and customizable cache parameters on the schedulability rate of critical tasks within a real-time system.

The subsequent sections provide detailed explanations of the following topics. First, we introduce the Cachegrind simulation tool (Section 4.1), which enables us to assess the effects of substitution policies on benchmarks. By examining the number of cache misses and execution time, we can analyze the performance implications of different policies. The outcomes of this analysis are presented in Section 4.2. Finally, we delve into Section 4.3, where we conduct a study to determine the improvements in system schedulability resulting from the employed cache architecture.

4.1 CACHEGRIND: PROFILING CACHE BEHAVIOR IN DYNAMIC ANALYSIS

Cachegrind, an instrumental tool provided by Valgrind (SEWARD et al., 2021), plays a pivotal role in dynamic analysis by facilitating cache simulations and generating detailed profiles. This section explores the significance of Cachegrind within the context of cache behavior analysis and its implications for the evaluation of cache substitution policies in a real-world scenario.

Cachegrind, an open-source tool widely used for dynamic analysis, focuses on providing insights into cache performance by simulating cache hierarchies and collecting valuable profiling data (SEWARD et al., 2021). By utilizing Cachegrind, researchers gain access to crucial information, including the number of executed instructions (Irefs), accesses and misses in the L1 instruction cache (I1 refs and I1 misses, respectively). Additionally, it provides data-related metrics such as references to data (Drefs), accesses and misses in the L1 data cache (D1 refs and D1 misses, respectively), as well as the number of misses in the last level cache (LL misses).

The standard version of Cachegrind, developed by its creators (SEWARD et al., 2021), employs the widely-used LRU policy for evicting and writing cache lines following a cache miss. However, in this dissertation, an extension to the standard Cachegrind version has been implemented to incorporate alternative cache substitution policies such as FIFO, RANDOM, LIP, BIP and DIP. This extension enhances the flexibility and scope of analysis when evaluating the impact of different cache substitution policies on real-time system performance.

The integration of Cachegrind into the research methodology ensures accurate and detailed cache behavior profiling. By employing Cachegrind's extended version, it becomes possible to assess the influence of various cache substitution policies on cache hit rates, cache miss rates and overall cache performance. The obtained insights provide researchers with a deeper understanding of cache behavior in different scenarios and aid in making informed decisions regarding cache parameter configurations for real-time systems.

In summary, Cachegrind, a powerful and customizable tool, enables dynamic analysis and profiling of cache behavior. Its integration into this dissertation facilitates the evaluation of cache replacement policies, enabling researchers to make informed decisions based on comprehensive cache performance analysis.

4.2 CATEGORIZATION OF BENCHMARKS AND EVALUATION OF REPLACEMENT POLICIES

In this section, we discuss the categorization of benchmarks used to evaluate the performance of different cache replacement policies. Additionally, we analyze the impact of these policies on the execution time of various applications, aiming to understand the influence of cache architecture on the schedulability rate of critical tasks in a real-time system.

To assess the performance variation among different applications when cache replacement policies are modified and to evaluate the effect of these policies on the application execution time, four sets of benchmarks were employed, listed below. These benchmarks represent both memory-intensive workloads, such as those encountered in autonomous vehicles (Cortex (THOMAS et al., 2014) and PARSEC (BIENIA, 2011)) and low-memory applications, such as simple controllers (Mibench (Guthaus et al., 2001) and Mälardalen (GUSTAFSSON et al., 2010)).

Mälardalen[1] **-** The Mälardalen benchmarks, made freely available by the Mälardalen University of Sweden, serve as a means to evaluate and compare different Worst-Case Execution Time (WCET) analysis tools (GUSTAFSSON et al., 2010).

MiBench[2] **-** Developed by researchers at the University of Michigan, the MiBench benchmarks aim to provide a comprehensive set of programs that are representative of commercially embedded systems. The benchmarks are divided into six categories based on their commercial applications: Automotive

[1] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[2] http://vhosts.eecs.umich.edu/mibench/

and Industrial Control, Network, Security, Consumer Devices, Office Automation and Telecommunications (Guthaus et al., 2001).

Cortex Suite[3] **-** The Cortex Suite, developed at the University of San Diego, consists of eight applications focusing on natural language processing, computer vision and machine learning (THOMAS et al., 2014).

PARSEC[4] **-** PARSEC (Princeton Application Repository for Shared-Memory Computers) is a collection of multithreaded benchmarks specifically designed to represent emerging workloads and serve as a benchmark suite for shared-memory multiprocessor systems (BIENIA, 2011).

To evaluate the impact of cache replacement policies, experiments were conducted using the extended version of Cachegrind. The benchmarks were executed with varying cache partition sizes (from 4 KB to 1 MB), different numbers of cache ways (2, 4, 8 and 16) and different cache replacement policies (LRU, LIP, RANDOM, FIFO and BIP). For the BIP policy, three variations of the bimodal parameter (ϵ - 1/64, 1/32 and 1/16) were considered, following the approach in (QURESHI et al., 2007). The cache line size was fixed at 64 bytes.

From these experiments, the number of cache misses was recorded for each application and replacement policy, resulting in over 18,000 combinations of experiments. The number of cache misses for each replacement policy was then compared to the performance of the LRU policy, which is commonly considered the baseline due to its favorable performance characteristics (PATTERSON; HENNESSY, 2013). Based on these comparisons, the applications were categorized into four groups:

(i) Applications exhibiting significant gains (over 15% improvement) in the number of cache misses when using at least one replacement policy other than LRU and under at least one cache configuration setting (considering partition size and number of ways).

(ii) Applications showing considerable gains (between 10% and 15%) but not fitting into the previous category.

(iii) Applications with moderate gains (between 5% and 10%) that do not fall into the first two categories.

(iv) Applications demonstrating low or no gains (up to 5%) that do not fit into any of the previous categories.

Table 4.1 provides an overview of the benchmark categorization based on the described methodology.

In order to examine the impact of cache misses on the execution time of benchmarks, a collection of experiments was conducted using the cache-

[3] http://cseweb.ucsd.edu/groups/bsg/
[4] https://parsec.cs.princeton.edu/

Table 4.1 – Benchmark categorization considering the percentage of cache misses.

Category	Benchmark
No gain to small (0-5%)	Cortex: disparity-qcif, me-medium, sphinx-large, stitch-fullhd, liblinear-tlarge, liblinear-tmedium, spc-large, spc-medium, spc-small, mser-fullhd, mser-qcif, me-large, me-small, mser-cif, multi_ncut-fullhd, rbm-small Parsec: canneal-large, ferret-large, raytrace-large, raytrace-medium, streamcluster-large, swaptions-large, vips-large mibench: adpcm_large_en, adpcm_small_dec, bf_large_dec, bf_small_enc, bitcount_large, bitcount_small, jpeg_large_ppm, jpeg_small_ppm, jpeg_small_progressive, qsort_small, stringsearch_large, susan_large, susan_large_corner, susan_large_edge, susan_large_smooth, susan_small, susan_small_corner, susan_small_edge, susan_small_smooth, gsm_small_enc Malardalen: all
Moderate (5-10%)	Cortex: disparity-fullhd, liblinear-tsmall, texture_synthesis-cif, sphinx-small, svd3-small, kmeans-large, sift-vga, rbm-large, sphinx-medium Parsec: vips-medium, vips-small mibench: gsm_small_dec, stringsearch_small, patricia_small, qsort_large, rijndael_small_dec
Large (10-15%)	Cortex: kmeans-medium, multi_ncut-cif, tracking-fullhd, tracking-vga, disparity-cif, lda-large, lda-small, rbm-medium, srr-large, tracking-cif, sift-qcif Parsec: ferret-large mibench: patricia_large, basicmath_small, fft_small_inv, gsm_large_enc, jpeg_large_progressive, rijndael_large_enc, rijndael_small_enc
Very large (>15%)	Cortex: kmeans-small, lda-medium, multi_ncut-qcif, pca-large, pca-medium, pca-small, sift-cif, sift-fullhd, stitch-vga, stitch-cif, svd3-large, svd3-medium, texture_synthesis-fullhd, disparity-vga, srr-medium,srr-small mibench: basicmath_large, crc32, fft_large, fft_small, gsm_large_dec, rijndael_large_dec, sha_large, sha_small, dijkstra_large, dijkstra_small

related parameters of five different processors (as shown in Table 4.2). The execution time for each benchmark was calculated using Equation 4.1, which takes into account the number of instructions (I), the number of cache misses (D_MISSES), the number of data references (D_REFS), the cycles per instruction (CPI), cache miss penalty (MISS_PENALTY) and cache hit penalty (HIT_PENALTY) specific to each processor parameter (as defined in Table 4.2).

Table 4.2 – Parameters of the considered processors.

Processors	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 instructions 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

$$
\begin{aligned}
exec_time = (I \times CPI) + \\
(D_MISSES \times MISS_PENALTY) + \\
((D_REFS - D_MISSES) \times HIT_PENALTY)
\end{aligned}
\tag{4.1}
$$

Figure 4.1 presents the results obtained for the application "pca-small" with eight cache ways, considering three different processors: (a) Intel i7, (b) ARM A53 and (c) the processor used in (QURESHI et al., 2007). The x-axis represents the size of the cache partition, while the y-axis represents the normalized

execution time compared to the LRU policy. The slashes on the graph indicate different cache line replacement policies. A slash lower than that of LRU denotes a reduction in cache misses and, consequently, an improvement in execution time. The figure highlights the benefits of utilizing various cache replacement policies and different cache partition sizes for a specific task. For instance, in Figure 4.1(c), "pca-small" achieved a nearly 40% improvement in runtime with a 128 KB cache partition simply by changing the replacement policy. Furthermore, the figure illustrates the influence of cache partition size on cache line replacement policies, revealing that certain policies are more sensitive to cache size than others (e.g., FIFO exhibits less variance compared to RANDOM).

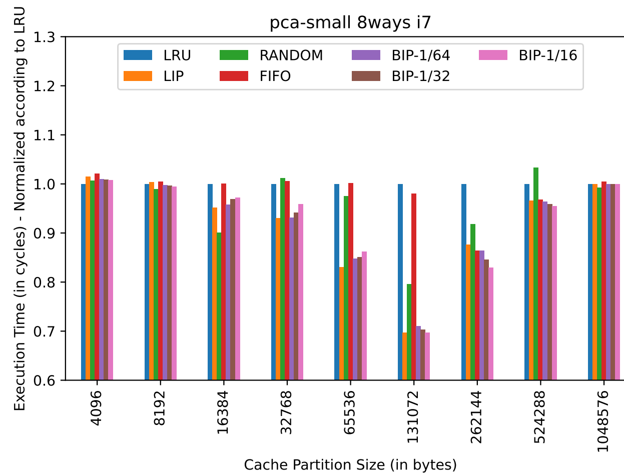
Figure 4.2 presents the results obtained for "svd3-large" on the Intel i7 processor, this time varying the number of cache ways (four ways in Figure 4.2(a) and 16 ways in Figure 4.2(b)). In this scenario, the most significant optimization, compared to the LRU policy, was achieved with the LIP policy and a 64 KB cache partition, resulting in approximately a 12% improvement. This figure emphasizes that the number of cache ways is another parameter influencing the performance and behavior of cache replacement policies.

Section 3.2 delves into a comprehensive examination of studies and research findings concerning cache replacement policies within real-time systems. This section underscores the critical significance of selecting cache replacement policies that meticulously account for the distinctive attributes of real-time workloads. The results presented in this chapter substantiate these findings with tangible evidence, aligning seamlessly with the overarching discourse on cache replacement policies in the realm of real-time systems. Specifically, these results showcase how diverse cache replacement policies and cache configurations wield a substantial influence on application performance, bolstering the conclusions drawn from the research outlined in section 3.2.

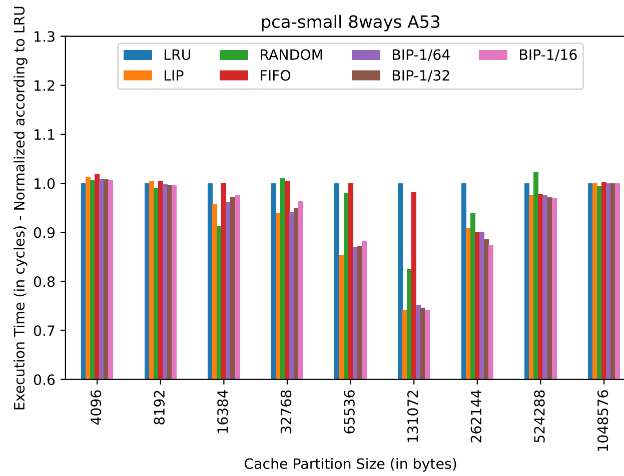
Section 4.3 further utilizes the benchmark categories to evaluate the benefits of employing a per-partition cache line replacement policy in terms of schedulability.

4.3 SCHEDULABILITY IMPACT ASSESSMENT

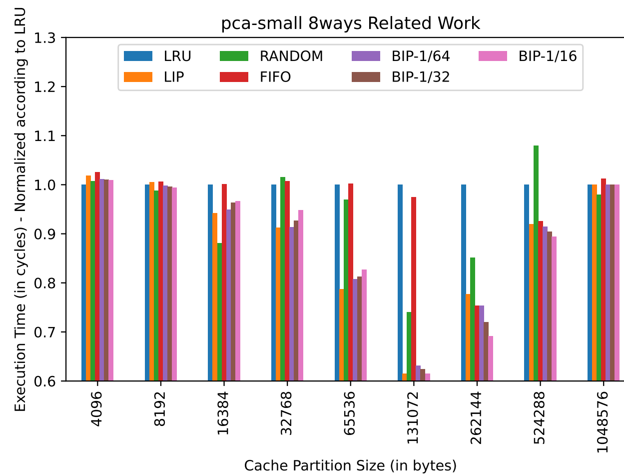
The evaluation of the proposed approach involved comparing the schedulability rates of different cache replacement policies for a set of periodic tasks running on a single processor. The experiments were conducted using fixed-priority scheduling algorithms with and without preemption. Two sets of tasks were generated for each experiment: a base set running under the LRU policy and an identical set running under the flexible approach that performed the best.



(a) Intel i7

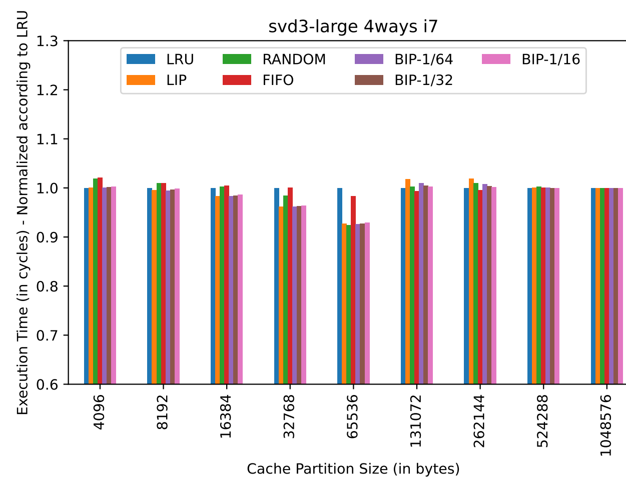


(b) ARM A53

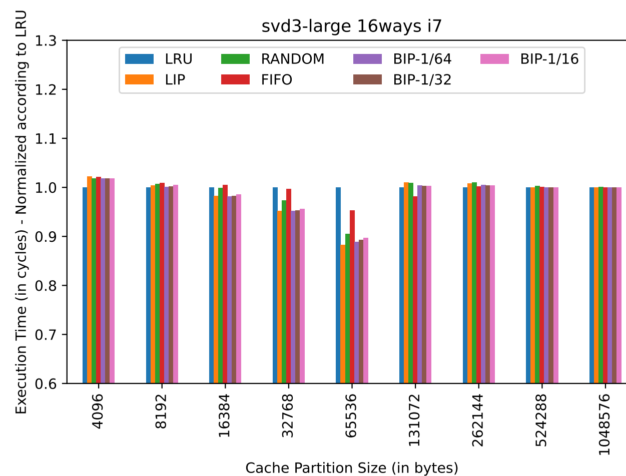


(c) Processador usado em (QURESHI et al., 2007).

Figure 4.1 – Impact of cache misses on execution time per processor, varying the size of the cache partition in 8 ways.



(a) 4 ways



(b) 16 ways

Figure 4.2 – Impact of number of ways on execution time, varying cache partition size.

The base set of tasks was generated with varying numbers of tasks (n) ranging from 4 to 32 and the utilization (U) of the tasks was varied from 0 to 1 with a step of 0.05. Each experiment involved generating 10,000 sets of tasks for each U and n combination. The utilization values of each task were randomly generated using the impartial method proposed by Emberson et al. (EMBERSON; STAFFORD; DAVIS, 2010) to ensure a uniformly distributed random distribution of utilization values that sum up to U. The periods (t_i) of the tasks were randomly generated within the range of 10,000 to 1,000,000 microseconds.

To create the identical set, the WCET of the tasks from the base set was scaled down by a speedup factor. The speedup factor was determined based on the cache acceleration category. Table 4.3 presents the cache acceleration categories derived from the benchmark results obtained in Section 4.2.

The acceleration factors for each category were randomly generated within specific ranges.

Table 4.3 – Parameters of schedulability experiments.

Category	**Parameters**
cat I	90%: 0.00 - 0.05, 10%: 0.06 - 0.10
cat II	10%: 0.00 - 0.05, 20%: 0.06 - 0.10 30%: 0.11 - 0.15, 40%: 0.16 - 0.25
cat III	100%: 0.16 - 0.40

For example, in category I, 90% of the tasks had acceleration factors randomly generated between 0.00 and 0.05, while the remaining 10% had factors between 0.06 and 0.10. The acceleration factors were then used to decrease the WCET of the tasks in the identical set according to the formula $c_i \cdot (1 - s_i)$, where c_i is the WCET of the original task from the base set, and s_i is the acceleration factor.

Both the base set and the identical set of tasks had implicit deadlines and were assigned priorities based on the Rate-Monotonic scheduling policy, where tasks with shorter periods received higher priorities.

The schedulability rates of the base task sets and their corresponding identical sets were compared using non-preemptive and preemptive fixed-priority scheduling algorithms. For the non-preemptive case, the response time analysis proposed by Davis et al.(DAVIS et al., 2007) was applied, while for the preemptive case, the response time analysis by Audsley et al.(AUDSLEY et al., 1993) was used. In the preemptive case, it was assumed that each task was assigned a separate cache partition.

Figure 4.3 shows the schedulability rate for non-preemptive fixed-priority algorithms, while Figure 4.4 presents the schedulability rate for preemptive fixed-priority algorithms. The plots (a), (b) and (c) in both figures correspond to cache acceleration categories I, II and III, respectively. The results indicate that even in category II with moderate acceleration, the schedulability gain can reach up to 5% for non-preemptive cases and can maintain system schedulability in preemptive cases for utilization greater than 0.8. In category III, with the highest accelerations, the schedulability improvement can reach over 40% for non-preemptive cases with high-utilization task sets.

These findings highlight the effectiveness of the proposed flexible approach in improving schedulability rates for real-time systems by utilizing cache replacement policies that take into account the cache behavior and the characteristics of the tasks. These empirical findings underscore the effectiveness of the flexible approach proposed in this chapter. This approach capitalizes on cache replacement policies that account for both cache behavior and

task characteristics, as elaborated in the section 3.2 on cache replacement policies. By aligning the research findings from both sections, it becomes evident that cache replacement policies play a pivotal role in shaping real-time system performance and schedulability. The selection of cache replacement policies tailored to real-time workloads is essential for optimizing cache utilization, improving predictability, and enhancing overall system performance. These combined insights serve as a foundation for further research and the development of novel cache replacement policies specifically designed to meet the unique demands of real-time systems.

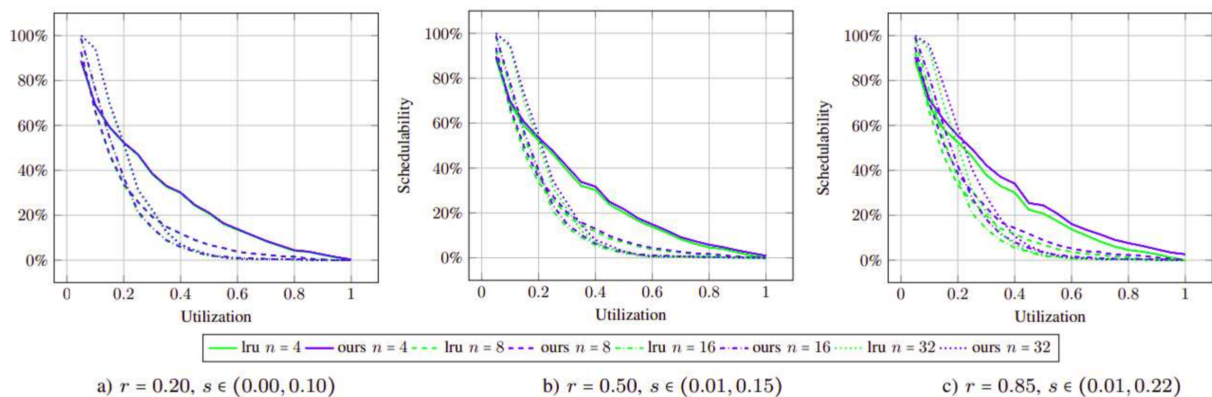


Figure 4.3 – Schedulability rate of non-preemptive fixed-priority algorithms.

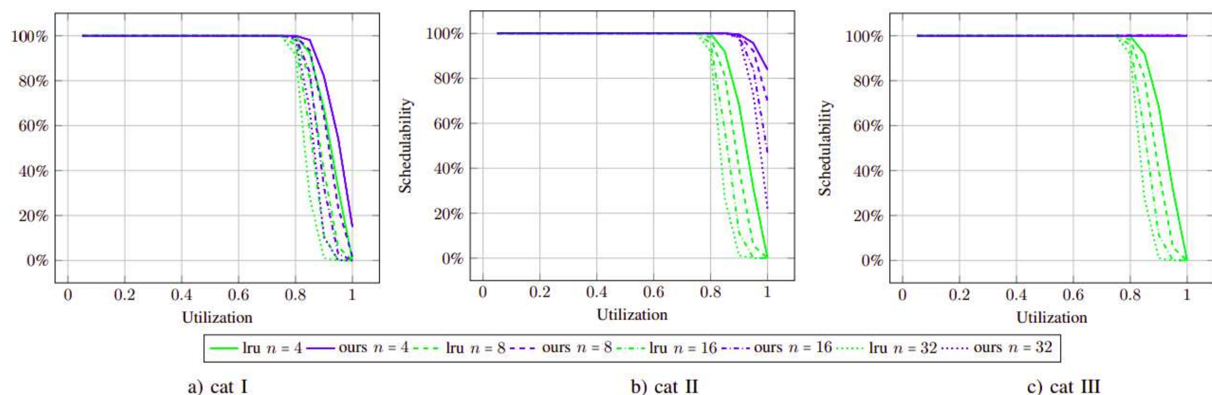


Figure 4.4 – Schedulability rate of preemptive fixed-priority algorithms.

4.4 PARTIAL CONSIDERATIONS

This chapter presented a comprehensive evaluation and analysis of cache replacement policies and their impact on system schedulability. The chapter starts by conducting an extensive evaluation using a diverse set of benchmarks. These benchmarks are designed to simulate applications with

varying memory usage profiles. By applying different cache replacement policies, the chapter measures and analyzes their performance in reducing cache misses. The results obtained from this evaluation not only validate the motivation behind the research but also provide valuable insights into the effectiveness of different policies in improving cache performance.

Moving forward, the chapter delves into the schedulability analysis of a cache architecture that supports task partitioning and allows for the selection of replacement policies per partition. The implemented cache replacement policies include RANDOM, LRU, FIFO, LIP and BIP. By comparing systems utilizing this architecture to those relying solely on the LRU policy, the chapter examines the impact on system schedulability. The analysis demonstrates a clear improvement in schedulability for systems that embrace the proposed architecture, emphasizing the importance of flexible cache replacement policies in real-time systems.

By achieving the objectives set for this chapter, a significant contribution is made to the overall research. The findings not only highlight the effectiveness of various cache replacement policies in reducing cache misses but also underline the benefits of incorporating task partitioning and flexible replacement policies for enhanced schedulability. The findings of this chapter is one of the results that have been published at 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC) (ARAUJO et al., 2021). These insights pave the way for future research and advancements in cache management techniques for real-time systems.

5 IMPLEMENTATION AND EVALUATION OF THE DIP POLICY IN CACHEGRIND

In this chapter, we present the implementation and evaluation of the DIP (Dynamic Insertion Policy) in Cachegrind. The DIP policy is designed to improve cache performance by dynamically determining the optimal time to insert cache lines. The implementation involves integrating the DIP policy into the existing cache simulation framework provided by Cachegrind, while the evaluation focuses on assessing its effectiveness in improving cache performance and system schedulability.

The implementation of the DIP policy in Cachegrind involves modifying the cache management algorithm to incorporate the DIP policy logic. This includes monitoring the cache state, analyzing access patterns, and dynamically adjusting the insertion time of cache lines based on the observed patterns. By implementing the DIP policy within Cachegrind, researchers and developers can leverage the enhanced cache simulation capabilities to evaluate its performance in various scenarios.

To evaluate the effectiveness of the DIP policy, a series of experiments are conducted using Cachegrind with the integrated DIP extension. These experiments involve running benchmark applications that represent a range of real-world scenarios and workload characteristics. During the evaluation, metrics such as cache hit rates, cache miss rates and execution times are measured and compared against other commonly used cache replacement policies. Additionally, the impact of the DIP policy on system schedulability is assessed to determine any potential improvements.

The evaluation of the DIP policy provides valuable insights into its performance and suitability for various cache management scenarios. By comparing its performance against other cache replacement policies, researchers and practitioners can make informed decisions regarding the selection of cache policies based on their specific system requirements and workload characteristics.

In summary, this chapter presents the implementation of the DIP policy within Cachegrind and evaluates its effectiveness in improving cache performance and system schedulability. The integrated DIP policy allows for enhanced cache simulation capabilities, providing researchers and developers with valuable insights into the impact of cache policies on overall system performance.

5.1 IMPLEMENTATION OF DIP

Cachegrind operates by instrumenting the program's binary code, tracking each memory access and simulating the behavior of the cache system. It

maintains a set of cache data structures, including the main cache and auxiliary caches, to keep track of cached data and determine cache hits and misses.

The global mechanism proposed in (QURESHI et al., 2007) offers a strategy for integrating the DIP policy into cachegrind, as depicted in Figure 5.1. This mechanism involves the use of two auxiliary directories: ATD-LRU (Auxiliary Tag Directory with LRU policy) and ATD-BIP (Auxiliary Tag Directory with BIP policy). These directories contain copies of the main cache blocks stored in the Main Tag Directory (MTD). Each auxiliary directory exclusively applies either the LRU policy or the BIP policy. The selection of the policy to be used in the main cache is dynamically determined based on the number of cache misses in each auxiliary directory. This decision is facilitated by a binary counter, *psel*, which tracks and controls the cache replacement policy selection process. Specifically, *psel* is incremented whenever a cache miss occurs in ATD-BIP and decremented when a cache miss occurs in ATD-LRU. If the most significant bit of *psel* is 0, indicating that *psel* is less than a predefined threshold (*psel_msb*), the LRU policy is chosen for the MTD. Otherwise, if the most significant bit is 1, indicating that *psel* is greater than or equal to *psel_msb*, the BIP policy is selected.

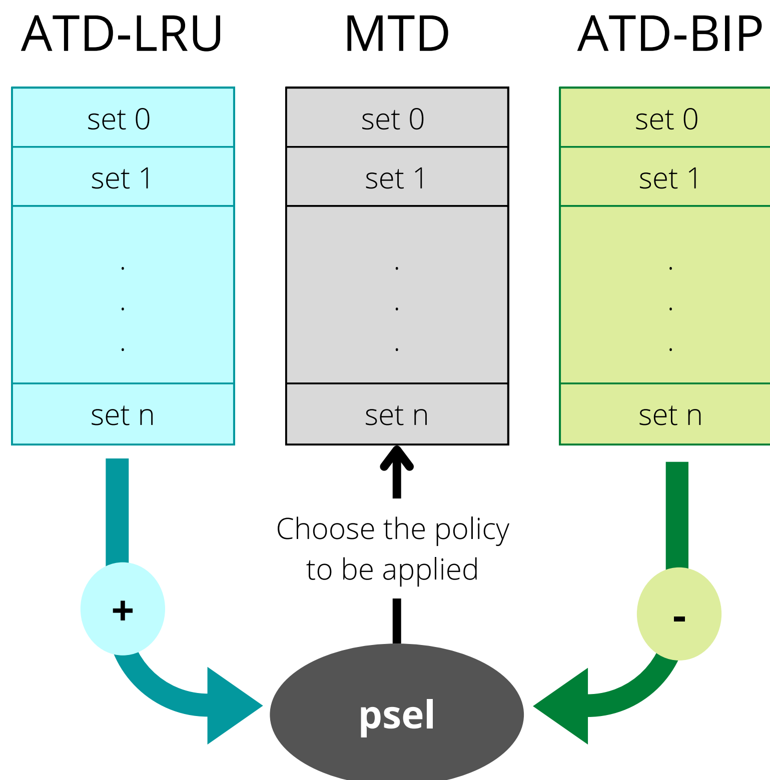


Figure 5.1 – Global Mechanism of DIP.

To implement the DIP policy in cachegrind, two cache copies are cre-

ated: one operating under the BIP policy and the other under the LRU policy. This allows for a direct comparison of the performance between the two policies. The *psel* counter is implemented with a configurable number of bits (e.g., 4, 8, 10, 11, 20, or 40 bits). It can be decremented to 0 and incremented to its maximum binary value (*psel_max*). Additionally, *psel* is initialized to the middle value of its range (*psel_msb*). At the end of each evaluation, if *psel* is greater than or equal to *psel_msb*, the BIP policy is applied to the main cache. Otherwise, if *psel* is less than *psel_msb*, the LRU policy is selected. For example, with a 4-bit *psel*, the range is from 0 to 15, corresponding to the binary values 0000 and 1111, respectively. In this case, *psel_msb* would be set to 8, which is 1000 in binary, representing the first value with a 1 in the most significant bit. The initialization of *psel* to *psel_msb* ensures it starts in the middle of the (0, 15) range.

Algorithm 5.1 outlines the implementation of the DIP policy in cachegrind. It initializes the main cache, as well as the auxiliary caches for the LRU and BIP policies. The static *psel* variable is declared with a size specified by *psel_size*. The LRU policy is applied to cache_lru, while the BIP policy is applied to cache_bip.

During cache access, if a cache miss occurs in cache_lru, *psel* is incremented if it is less than *psel_max*. Similarly, if a cache miss occurs in cache_bip, *psel* is decremented if it is greater than 0. Based on the value of the most significant bit of *psel*, the corresponding policy (LRU or BIP) is applied to the main cache, cache_main.

```
1  define cache_main; /* initialize the main tag */
2  define cache_lru; /* inicialize the auxiliary tag for LRU  */
3  define cache_bip; /* inicializa the auxiliary for BIP */
4
5  static psel:[psel_size]; /* declare variable psel */
6
7  apply LRU at cache_lru;
8  apply BIP at cache_bip;
9
10
11 if miss at cache_lru
12     if psel < psel_max
13         psel++;
14 if miss at cache_bip
15     if psel > 0
16         psel--;
17
18 if msb_psel=0
19     apply LRU at cache_main;
```

```
20 else
21     apply BIP at cache main;
```

Algorithm 5.1 – Algorithm for DIP policy.

5.2 PARAMETERS OF EXPERIMENTS

To evaluate the implementation of the DIP policy in Cachegrind, several experiments were conducted using different sets of benchmarks. The benchmarks were categorized into three sets: Mälardalen, MiBench and CortexSuite. The specific benchmarks used in each set are listed in Table 5.1.

The experiments were conducted using different cache parameters, as shown in Table 5.2. The cache partition size was varied from 2KB to 1MB and the number of cache ways ranged from 2 to 32. The cache line size was kept constant at 64 bytes throughout the experiments. Additionally, the size of the *psel* controller, which determines the behavior of the DIP policy, was also varied. The *psel* sizes considered were 4 bits, 8 bits, 10 bits, 11 bits, 20 bits and 40 bits. This variation in *psel* size aimed to understand the impact it has on the behavior of the DIP policy, as the original work by (QURESHI et al., 2007) only defined *psel* sizes of 10 and 11 bits.

Table 5.1 – Benchmarks used in the experiments.

Set	***Benchmark***
Mälardalen	compress, edn, insertsort, prime
MiBench	adpcm, basicmath, bitcount, bf, crc32, dijkstra, fft, gsm, jpeg, patricia, qsort, sha, stringsearch, susan
Cortex Suite	kmeans, pca, spc, svd3

By conducting experiments with different cache parameters and *psel* sizes, it becomes possible to analyze the behavior of the DIP policy under various configurations. The goal is to observe the number of failures in the L1 cache caused by the LRU, BIP and DIP policies and compare their performances across different benchmarks and cache settings. These experiments provide insights into the effectiveness of the DIP policy and its potential advantages over traditional cache replacement policies.

5.3 ASSESSMENT OF THE DIP POLICY IMPLEMENTED IN CACHEGRIND

The experiments to evaluate the DIP used the following methodology: the Cachegrind ran with each benchmark matching each of the cache parameters from Table 5.2 for the LRU, BIP and DIP policies. In all, more than 16,000 results

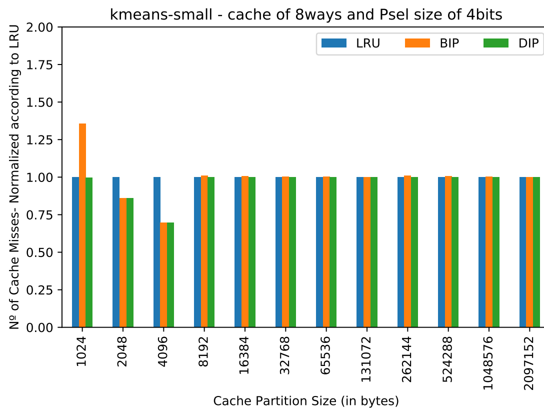
Table 5.2 – Cache parameters applied in the experiments.

Parameter	Set up
Partition Size	2 kB, 4 kB, 8 kB, 16 kB, 32 kB, 64 kB, 128 kB, 256 kB, 512 kB, 1 MB
Ways	2, 4, 8, 16 e 32
Line size	64 B
psel size	4b, 8b, 10b, 11b, 20b, 40b

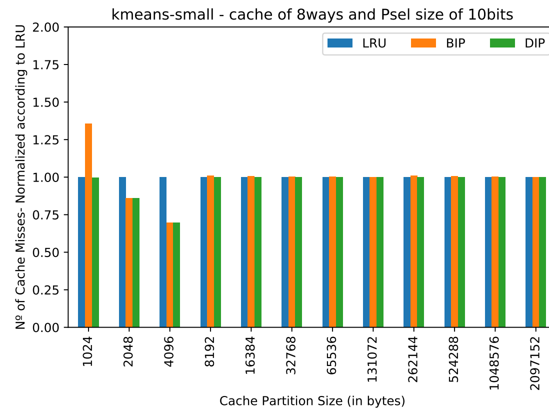
were generated with performance information from the L1 and LLC cache. With these results, graphs were generated of the number of cache misses as a function of the cache partition size (Figures 5.2, 5.3, 5.4 and 5.5), number of ways (Figure 5.6) and size of *psel* (Figure 5.8). The behavior of *psel* was also analyzed in Figure 5.7.

In the graphs of Figures 5.2, 5.3, 5.4 and 5.5, the y-axis represents the number of cache misses occurred in each application, respectively `kmeans`, `pca`, `spc` and `svd3`, when using the LRU, BIP and DIP policies; the x-axis contains the cache partition size values. In the figures the graphs are organized according to the number of ways and number of bits of *psel*, being (a) 8 ways and 4 bits, (b) 8 ways and 10 bits, (c) 16 ways and 4 bits and (d) 16 ways and 10 bits. The first conclusion to be drawn from these graphs concerns the functioning of the DIP, it is possible to observe that in all cases the DIP follows the policy that results in fewer cache misses (even having moments when the best policy is shown, as in Figures 5.3 (b) and (c) for cache partition of 131072 bytes). It is also important to note the variation that some policies present for certain partition sizes; for example, for `kmeans` the policies perform similarly for most partition sizes, with LRU being slightly better than BIP and DIP following this behavior; however, for 2048 and 4096 byte partitions, there is a drop in the number of cache misses causing not only the BIP policy to show better performance but also the two best cases for the *benchmark*. This behavior is repeated for `pca` and `svd3`, however the partition sizes that demonstrate the best performance are 131072 bytes and 262144 bytes for `pca` and 8192 bytes for `svd3`.

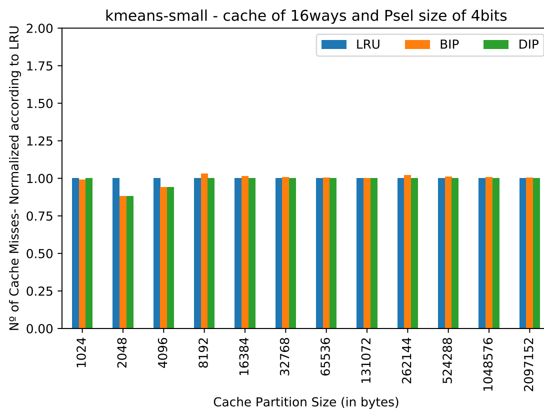
After verifying the behavior that cache partitions of different sizes cause in the DIP policy, it was decided to check if other cache parameters would also impact the performance of this policy. Figure 5.6, presents the results of the number of cache misses (y-axis) that occurred under each of the three policies as a function of the number of cache ways (x-axis). These graphs followed the parameters that performed best in the analysis of the cache partition size, (a) is the result of `kmeans` with a cache partition of 4096 bytes and PSEL of 4 bits, (b) is the result of `pca` with a cache partition of 131072 bytes and *psel* of 10 bits and



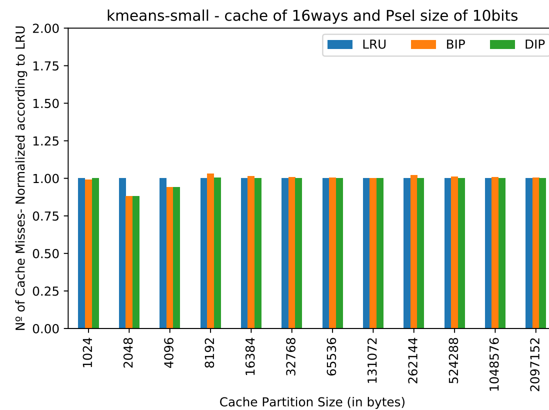
(a) 8-way cache partition and 4-bit *psel*



(b) 8-way cache partition and 10-bit *psel*



(c) 8-way cache partition and 4-bit *psel*

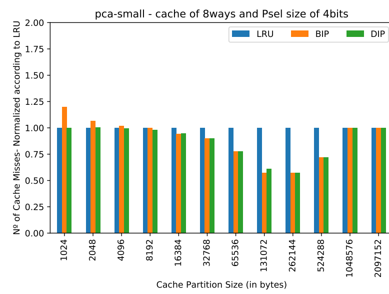


(d) 8-way cache partition and 10-bit *psel*

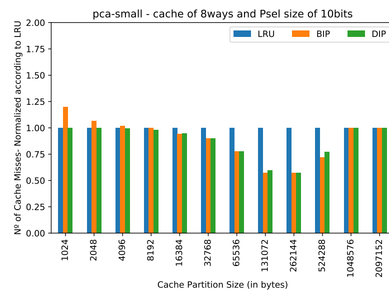
Figure 5.2 – Number of cache misses as a function of partition size for `kmeans`

(c) is the result of `svd3` with a cache partition of 4096 bytes and *psel* of 4 bits. from these results, it is possible to verify that the susceptibility of policies to the number of ways, being variable according to each application. For `kmean`, the number of ways that results in the least number of cache misses is 4; for `svd3` this value is 16 ways; for `pca` the minimum number is 16 ways, with the result of cache misses remaining constant after this value.

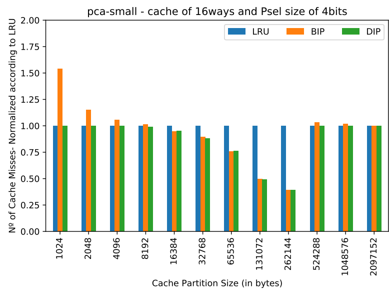
The Figures 5.2, 5.3, 5.4, 5.5 and 5.6 already demonstrate the dynamic nature of the DIP policy. By always selecting the best policy between LRU and BIP, DIP can adapt both to different applications and to the different phases of each application. Figure 5.7 shows the value of a 10-bit *psel* over the execution of four benchmarks, (a) `basicmath`, (b) `kmeans` and (c) `pca`. The horizontal axis represents the number of instructions, and the vertical axis corresponds to the value of *psel*. For a 10-bit *psel*, a value equal to or greater than 512 indicates that the BIP policy was selected, otherwise the LRU policy was used. For graphs (a) and (b), respectively, `basicmath` and `kmeans`, the value of *psel*, during the initial



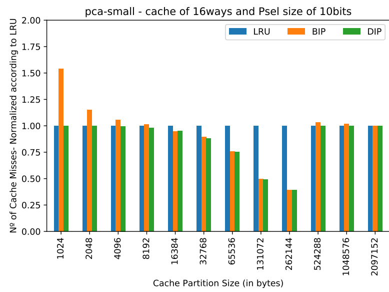
(a) 8-way cache partition and 4-bit *psel*



(b) 8-way cache partition and 10-bit *psel*

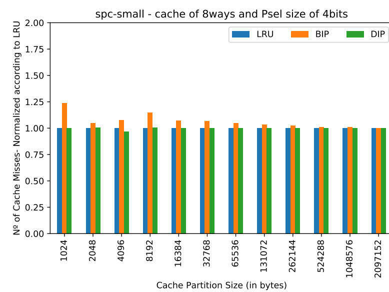


(c) 16-way cache partition and 4-bit *psel*

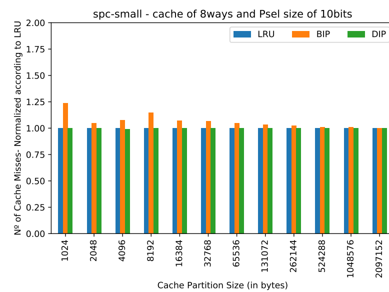


(d) 16-way cache partition and 10-bit *psel*

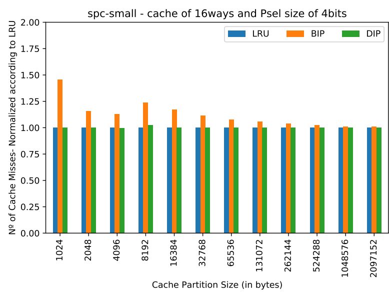
Figure 5.3 – Number of cache misses as a function of partition size for `pca`



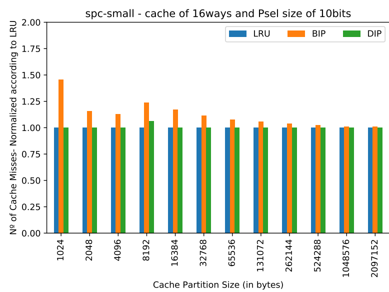
(a) 8-way cache partition and 4-bit *psel*



(b) 8-way cache partition and 10-bit *psel*

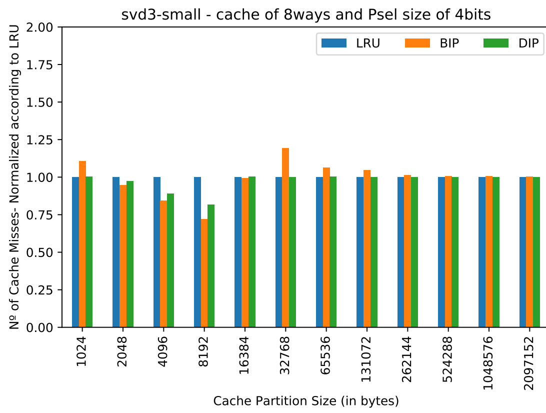


(c) 16-way cache partition and 4-bit *psel*

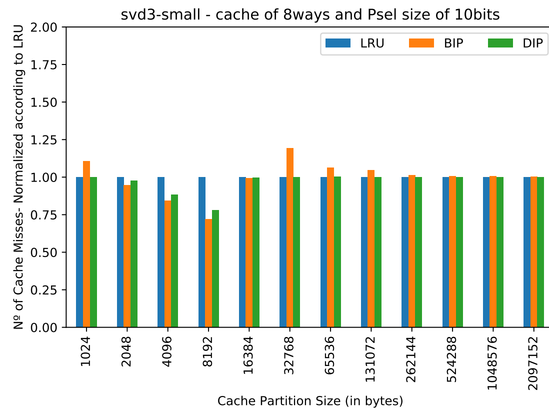


(d) 16-way cache partition and 10-bit *psel*

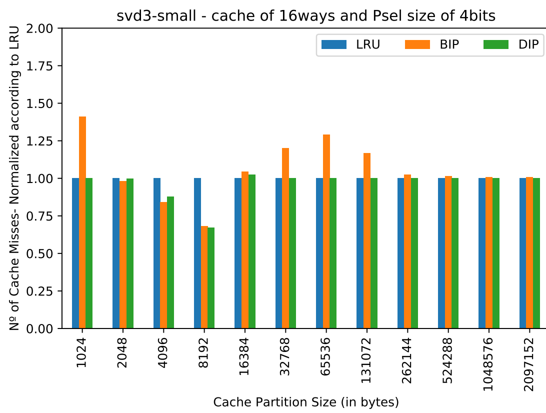
Figure 5.4 – Number of cache misses as a function of partition size for `spc`



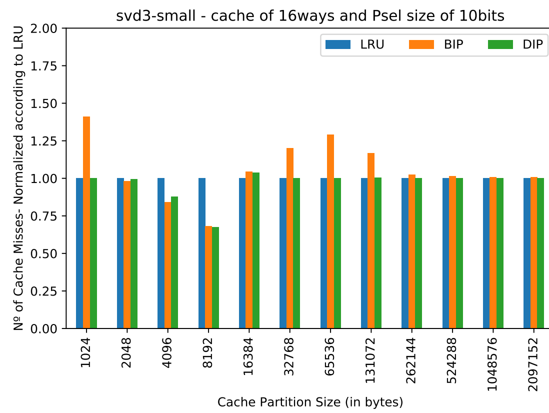
(a) 8-way cache partition and 4-bit *psel*



(b) 8-way cache partition and 10-bit *psel*



(c) 16-way cache partition and 4-bit *psel*

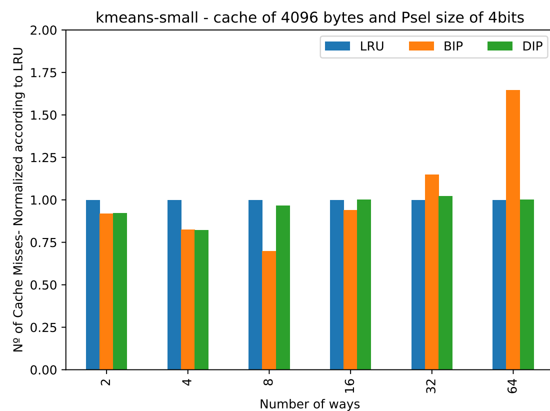


(d) 16-way cache partition and 10-bit *psel*

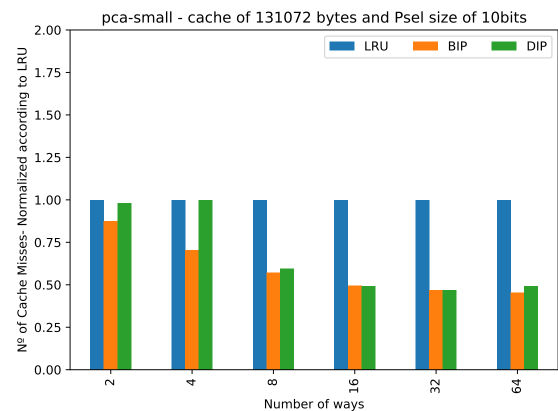
Figure 5.5 – Number of cache misses as a function of partition size for `svd3`

part of program execution, fits the baseline, 512, and any one of the policies works well. However, as the number of instructions increases during program execution, the value of *psel* decreases, so DIP selects LRU. For graph (c), of the `pca` application, the opposite occurs, at the beginning of the application execution, *psel* remains close to 0, indicating that BIP causes many more cache failures, however, with the increase in the number of instructions , this scenario is reversed, LRU causes more failures, the *psel* value increases and DIP selects the BIP policy.

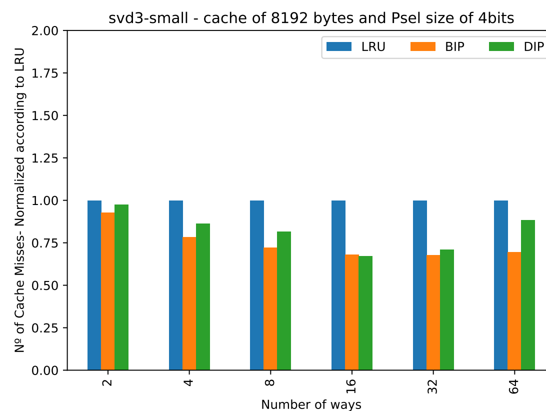
The behavior of the DIP policy in relation to the size of the implemented *psel* was also verified. Figure 5.8 shows these results. The x-axis represents the Variation of the value of *psel* during execution size in bits; the y-axis represents the number of cache misses suffered by each policy, normalized according to the LRU policy. Each graph displays the result for one policy, (a) for `kmeans`, (b) for `pca`, (c) for `svd3`. In (b) it is possible to observe that the DIP policy has a slight performance improvement from 10 bits, however in (a) and (c), the DIP



(a) `kmeans` with a cache partition of 4096 bytes and 4-bit *psel*



(b) `pca` with a cache partition of 131072 bytes and 10-bit *psel*



(c) `svd3` with a cache partition of 4096 bytes and 4-bit *psel*

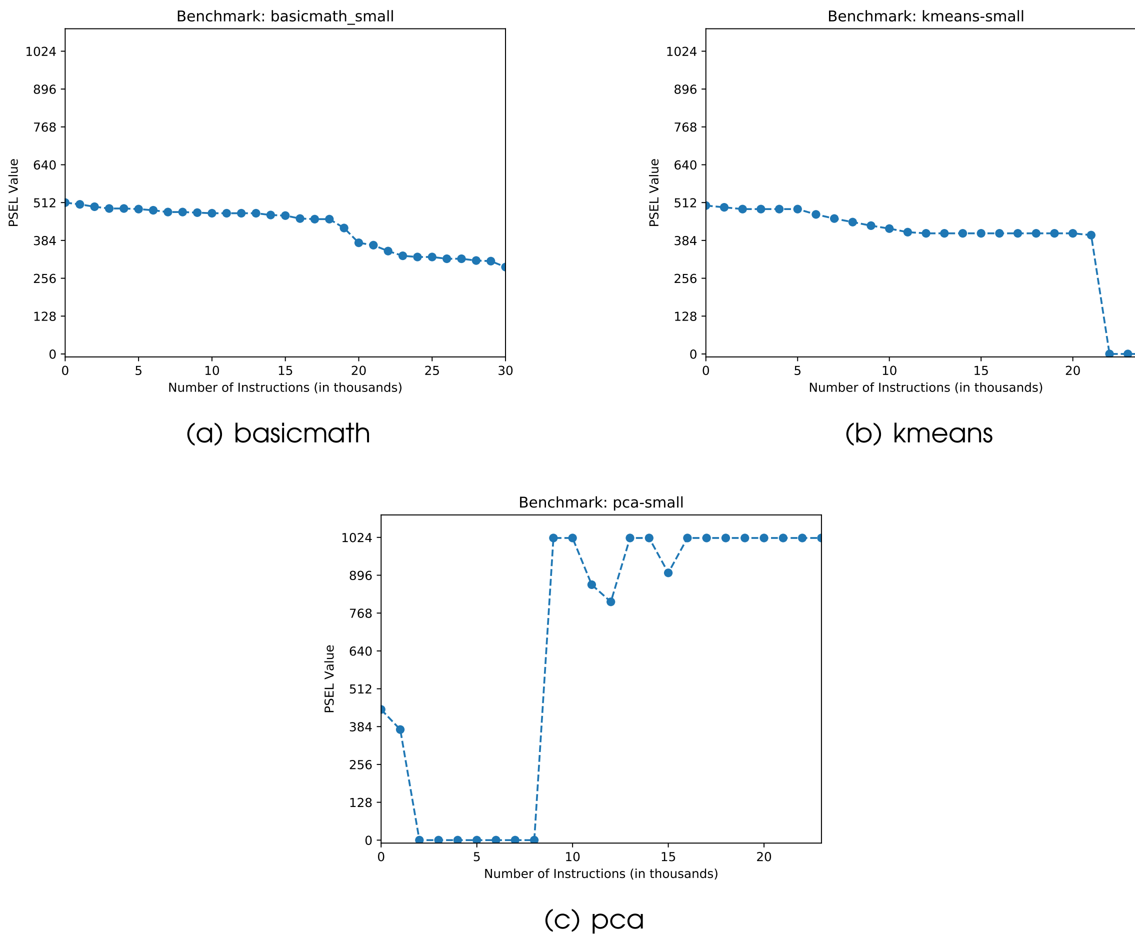
Figure 5.6 – Number of cache misses as a function of the number of ways

does not show any improvement. It can be concluded, then, that despite the importance of *psel* in the choice of policies that make up the DIP, its size is not a parameter that has a great impact on the functioning of the DIP policy.

5.4 PARTIAL CONSIDERATIONS

In this chapter, we introduced the implementation of the DIP (Dynamic Insertion Policy) in Cachegrind and conducted a comprehensive evaluation of its performance and behavior. The DIP policy, equipped with its adaptive insertion mechanism, demonstrated its effectiveness in dynamically selecting the optimal cache replacement policy between LRU and BIP for various benchmarks and cache configurations. By adapting to the specific characteristics of each application and its execution phases, the DIP policy showcased its capacity to enhance cache performance and reduce cache misses.

A key contribution of this chapter lies in highlighting the significance of



(a) basicmath

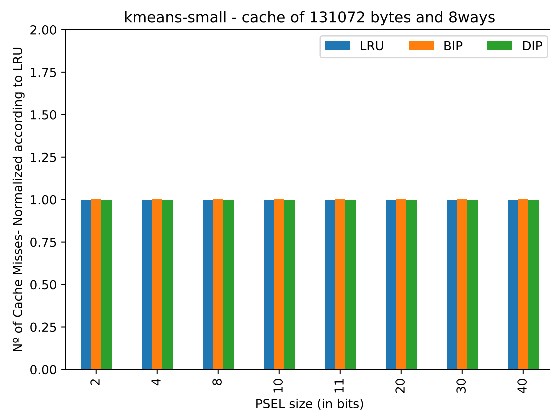
(b) kmeans

(c) pca

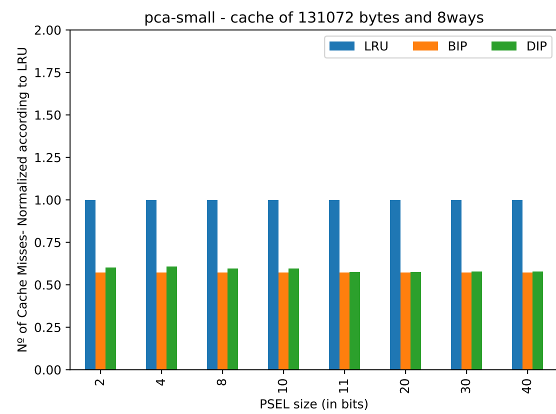
Figure 5.7 – Variation of the value of *psel* during execution

adaptive insertion policies within modern cache management. Unlike traditional cache replacement policies like LRU, which are static and do not account for the diverse behaviors and requirements of different applications, the DIP policy dynamically chooses the most appropriate replacement policy based on observed cache behavior during runtime. This adaptability enables the DIP policy to better align with the access patterns and temporal locality of each application, ultimately leading to improved cache utilization and a reduction in cache misses.

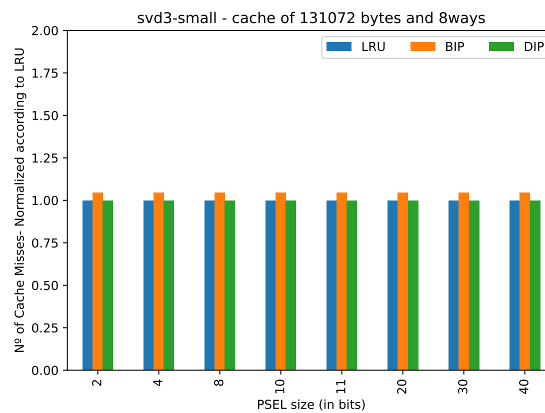
Furthermore, we delved into the impact of various cache parameters, such as cache partition size and the number of cache ways, on the DIP policy's performance. Our experiments unveiled that the DIP policy's performance is indeed influenced by these cache parameters, and specific configurations exist where it exhibits superior performance. Careful selection of the optimal cache partition size and number of cache ways empowers system designers and developers to enhance both the predictability and performance of the overall system.



(a) `kmeans` with a cache partition of 4096 bytes and 4-bit *psel*



(b) `pca` with a cache partition of 131072 bytes and 10-bit *psel*



(c) `svd3` with a cache partition of 4096 bytes and 4-bit *psel*

Figure 5.8 – Number of cache misses as a function of the size of *psel*

These insights shed light on the interplay between diverse cache configurations and strategies and their effects on the effectiveness of cache replacement policies, as also explored in Chapter 3. The causal relationship between cache partition size and policy performance suggests that specific partition sizes trigger performance shifts, influencing policy selection. A deeper investigation into the underlying factors contributing to these shifts could unveil critical insights into cache management.

The findings of this chapter have been formally published in the XI Brazilian Symposium on Computing Systems Engineering (ARAUJO et al., 2021), underscoring their significance and contribution to the field. With these results, we have successfully achieved specific Objective 3, which aimed to evaluate the DIP policy and its behavior in different cache configurations and benchmarks.

Building upon the insights gained from this chapter, the next chapter will focus on the development of an algorithm designed to assist in selecting the

optimal cache parameters for a given application or workload. This algorithm aims to further enhance the predictability and performance of cache management by providing automated guidance on cache partition sizes, the number of cache ways, and other relevant parameters. Leveraging the knowledge and experiences gained from evaluating the DIP policy, this algorithm holds the potential to refine cache configuration decisions and simplify the process for system designers and developers.

In summary, this chapter underscored the significance of adaptive insertion policies, such as the DIP policy, in improving cache performance by dynamically selecting the most suitable cache replacement policy. Additionally, it underscored the importance of cache parameter selection for optimizing system predictability. The upcoming chapter will delve into the development of an algorithm aimed at facilitating the selection of optimal cache parameters, further advancing the field of cache management and system performance.

6 CACHE OPTIMIZATION TECHNIQUES

The previous chapters have highlighted the impact of cache partition size and the number of ways on cache replacement policies, such as BIP, LRU and DIP. It has been demonstrated that different parameter configurations yield varying cache failure rates across different applications.

This suggests that selecting the optimal parameters and policy for each application can lead to improved system performance, enhancing metrics such as WCET and the schedulability of critical tasks.

Altmeyer et al. (ALTMEYER et al., 2014) study underscored the significance of cache partitioning as a technique to ensure task isolation and enhance cache utilization in real-time systems. It discussed the impact of partitioning granularity and allocation strategies on cache utilization and schedulability. The findings emphasized the necessity of carefully considering partitioning strategies based on specific system requirements.

The premises of this work are rooted in the insights and findings presented by Altmeyer et al. (2014), who evaluated cache partitioning for hard real-time systems, and the previous evaluation of the impact of different cache replacement policies on the performance of a real-time system (ALTMEYER et al., 2014).

Therefore, the objective of this chapter is to devise an algorithm capable of analyzing and selecting the optimal cache configurations within a cache architecture that facilitates task partitioning and the customization of cache parameters for each partition.

By incorporating the concept of cache partitioning, as discussed by Altmeyer et al. (2014), the proposed algorithm strives to optimize cache utilization and enhance the performance of real-time systems. It recognizes the importance of considering task-specific requirements and allowing flexibility in the selection of cache parameters per partition. This approach aligns with the objectives of improving the worst-case execution time (WCET) of tasks and bolstering system schedulability (ALTMEYER et al., 2014).

This chapter begins with an overview of Altmeyer et al.'s algorithm (Section 6.1). It then incorporates the findings presented in Chapters 5 and 4 to modify the algorithm and introduce the proposed Optimization Algorithm (Section 6.2). Next, the chapter evaluates both algorithms through three different comparisons (Section 6.3). Finally, it summarizes the findings and presents important considerations regarding the work (Section 6.4).

6.1 OVERVIEW OF OPTIMAL CACHE PARTITION ALGORITHM

The Optimal Cache Partition Algorithm (OCPA) was proposed in the work of Altmeyer et al. (2014). In the paper, the authors address the challenges posed by shared resources in multicore platforms and explore cache partition techniques to improve the predictability and timing guarantees of hard real-time systems. The algorithm aims to allocate cache partitions to tasks in a manner that optimizes cache utilization and enhances system schedulability (ALTMEYER et al., 2014).

Through their evaluation, Altmeyer et al. demonstrate the effectiveness of the proposed algorithm in improving cache utilization and system schedulability in hard real-time systems. They provide quantitative results and comparative analyses to support their findings, showcasing the benefits of cache partitioning in enhancing the performance and predictability of real-time tasks (ALTMEYER et al., 2014).

In summary, the algorithm presented in the paper offers a systematic approach to cache partitioning that considers task characteristics, cache requirements and scheduling constraints. It provides a practical framework for optimizing cache utilization in hard real-time systems, contributing to the overall system performance and meeting real-time requirements. But the results presented in Chapters 4 and 5, indicate another parameter related to cache that was not take into account in OCPA, the interference of replacement policies in the predictability of the cache. In the next section, we proposed an algorithm that provides per-task cache partitioning considering cache replacement policies based on OCPA.

6.2 THE PROPOSED ALGORITHM

The proposed cache partitioning algorithm is based on the principle that different tasks exhibit varying memory access patterns and temporal characteristics, which significantly impact the performance of cache replacement policies. It builds upon the OCPA by introducing the selection of the best replacement policies for each task in a per-task partitioned cache.

In the context of optimizing hard real-time systems, the main criterion is tasks set schedulability. Similar to the OCPA, the proposed algorithm achieves optimal cache partitioning if it identifies a configuration where the tasks can be scheduled, assuming such a configuration exists. To accomplish this, the algorithm utilizes benchmark performance results for different cache replacement policies while varying the cache partitioning size, as discussed in Chapter 5.

The proposed algorithm is summarized in Algorithm 6.1. The code begins

by defining the task set and the total cache size. It initializes the remaining cache size as the total cache size and sets the variable i to 0.

The $cacheAllocation$ function is responsible for allocating cache partitions for each task. It takes as input the task, remaining cache size, and the index i for selecting the partition size and replacement policy.

Within the function, the algorithm determines the partition size, WCET and replacement policy for the current task using the $findPartition$ function. It then checks the schedulability of the system with the current partition configuration by calling the $checkPartition$ function.

If the system is schedulable, the remaining cache size is updated by subtracting the partition size allocated for the task from the total cache size. The algorithm moves on to the next task.

If the system is not schedulable, the algorithm selects the next set of partition size and replacement policy by incrementing i. If i is less than 11 (indicating the number of available partition sizes), the $cacheAllocation$ function is called recursively with the updated i value to try different partition configurations.

If all partition configurations have been exhausted for a task and the system is still not schedulable, the algorithm goes back to the previous task and tries a different partition for it by calling the $cacheAllocation$ function recursively with the updated remaining cache size and i set to 1.

This process continues until a schedulable partition configuration is found for each task or it is determined that the system cannot be optimized. The algorithm explores different partition sizes and replacement policies, considering the remaining cache size at each step.

```
1  define task_set
2  define total_cache
3
4  int remaining_size = total_cache;
5  int i=0;
6
7  function cacheAllocation (task, remaining_size, i){
8
9      task_partition[task] = findPartition(task,remaining_size,i)[
        partition_size];
10     task_demands[task] = findPartition(task,remaining_size,i)[wcet];
11     task_policy[task] = findPartition(task,remaining_size,i)[policy];
12
13     if checkPartition(task_partition, task_demands) = isSchedulable{
14         remaining_size = total_cache - task_partition[task];
15         next task;
```

```
16      } else {
17          if i < 6{
18              i++;
19              cacheAllocation (task, remaining_size, i);
20          } else {
21              cacheAllocation (previous task, remaning_size +
                 task_partition[task], 1);
22          }
23      }
24  }
```

Algorithm 6.1 – Overview of proposed algorithm for optmized cache partitioning.

The main difference between the proposed algorithm and OCPA relies on the choosing of the replacement policy which will be applied in the cache partition. To define this policy, the proposed algorithm is capable to receive data of WCET related to partition sizes for each of the studied policies, then the algorithm evaluates these replacements policies, choosing the one with smaller WCET for each partition size and then running the $cacheAllocation$ function to finish the determination of the parameters applied in all task partition of the cache.

6.3 EXPERIMENTAL COMPARISON

To evaluate the proposed algorithm, it was running experiments that result in three comparisons between our approach and OCPA using three set of benchmarks listed in Table 6.1. The experiments aim to assess the utilization, WCET and schedulability for tasks sets under both approaches of cache partitioning.

Table 6.1 – Benchmarks used in the experiments.

Set	***Benchmark***
Parsec	canneal, ferret, raytrace, streamcluster, swaptions, vips
MiBench	adpcm, bf, bitcount, dijkstra, fft, gsm, jpeg, patricia, rijndael, stringsearch
Cortex Suite	disparity, kmeans, lda, liblinear, me, mser, multi_ncut, pca, rbm, sift, spc, sphinx, srr, stitch, svd3, texture_syntehesis, tracking

The parameters used in the experiments were established based on the one used in OCPA paper (ALTMEYER et al., 2014) and are summarized in Table 6.2. The cache partition could be sized in a range from 1 KB to 1 MB and the

number of cache ways was set to ranged from 2 to 32 while cache line size was kept constant as 64 bytes.

Table 6.2 – Cache parameters applied in the experiments.

Parameter	Set up
Partition Size	1kB, 2 kB, 4 kB, 8 kB, 16 kB, 32 kB, 64 kB, 128 kB, 256 kB, 512 kB, 1 MB
Ways	2, 4, 8, 16 e 32
Line size	64 B
Processors	A8, A53, i7, x86

By analyzing the results, we could draw conclusions about the benefits and limitations of the proposed algorithm compared to OCPA in the context of cache partitioning for hard real-time systems.

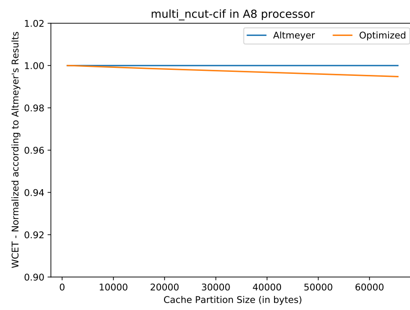
6.3.1 First Experiment: WCET x Cache Partition Size

The first experiment focused on evaluating the Worst Case Execution Time (WCET) of tasks in relation to the allocated cache partition size. The goal was to determine the extent to which the partition size influenced the WCET under both approaches.

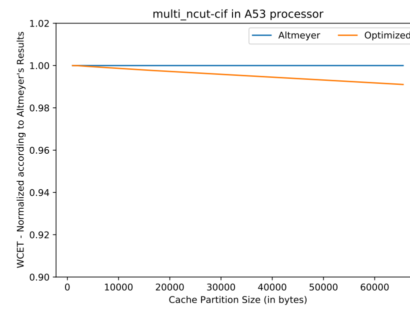
Figures 6.1 present some results of this experiment for the `multi_ncut` benchmark across all four processors. The graphs depict the cache partition sizes on the x-axis and the normalized WCET values according to Altmeyer's OCPA. The blue line represents the results of Altmeyer's OCPA, while the orange line represents our optimized proposed algorithm. Across all four processors, the optimized proposed algorithm consistently outperforms Altmeyer's OCPA, particularly for larger partition sizes, obtaining an improvement of approximately 1% for larger partition sizes on top of A8 processor, and a gain of around 2% for larger partition sizes on top of i7 processor.

Figure 6.2 demonstrates the results for the `disparity` benchmark. In this case, smaller partition sizes show no significant difference in performance between the two approaches. However, from medium-sized partitions, the superiority of the optimized proposed algorithm becomes evident (achieving about 0.5% of gain in the better case) and this difference becomes even more pronounced for larger cache partitions (presenting gains of about 0.5% in the worst case scenario and about 2% in the better case scenario).

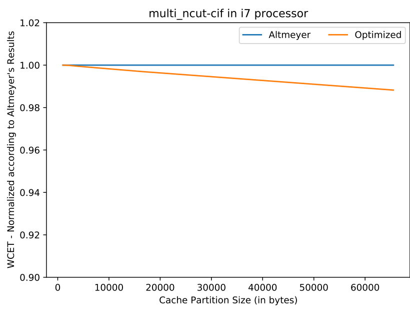
The relationship between partition size and WCET directly affects the schedulability of the systems. Therefore, this experiment serves as a means to verify the efficiency of the optimized proposed algorithm compared to Altmeyer's OCPA in terms of schedulability.



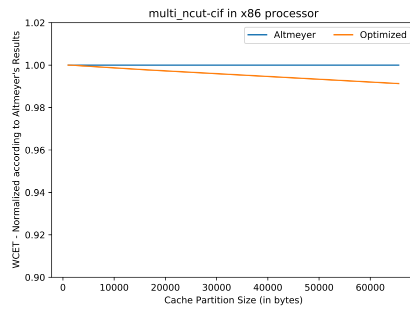
(a) On top of A8 processor



(b) On top of A53 processor

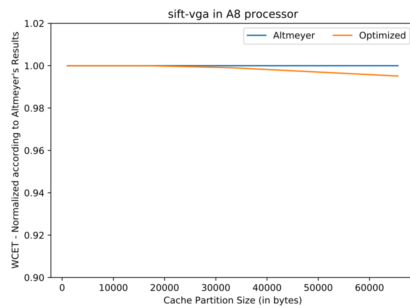


(c) On top of i7 processor

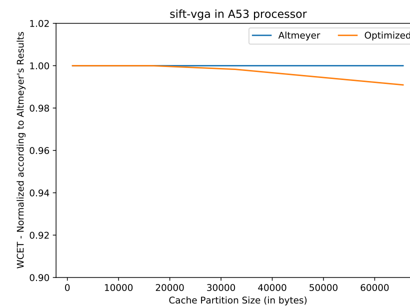


(d) On top of x86 processor

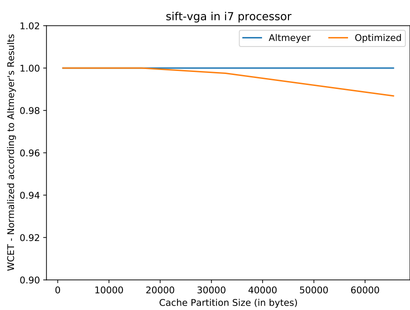
Figure 6.1 – Relation of WCET and cache partition size for `multi_ncut`.



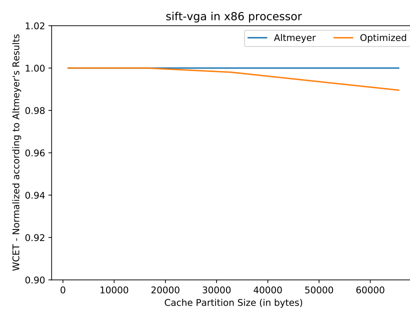
(a) On top of A8 processor



(b) On top of A53 processor



(c) On top of i7 processor



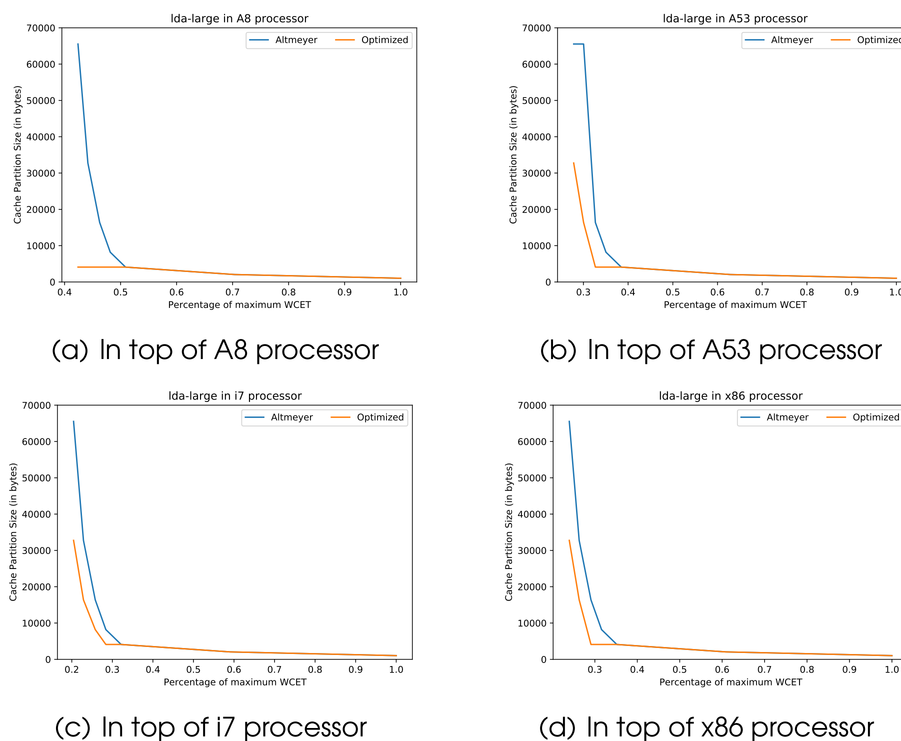
(d) On top of x86 processor

Figure 6.2 – Relation of WCET and cache partition size for `sift`.

6.3.2 Second Experiment: cache partition size x maximum WCET

The second experiment is complementary to the first one. It assesses the function between the cache partition size and the WCET, in other words, the second experiment evaluates the minimum cache partition size needed to reach a maximum WCET of the tasks and yet guarantee the system schedulability.

Figures 6.3 present some results of this experiment for the `lda` benchmark across all four processors. The graphs depict the WCET values on the x-axis and the cache partition size on the y-axis. The blue line represents the results of Altmeyer's OCPA, while the orange line represents our optimized proposed algorithm. These results show that to achieve smaller WCET, the proposed algorithm needs smaller partition size in comparison with Altmeyer's OCPA. Across all four processors, the optimized proposed algorithm consistently outperforms Altmeyer's OCPA, particularly for larger partition sizes.



(a) In top of A8 processor

(b) In top of A53 processor

(c) In top of i7 processor

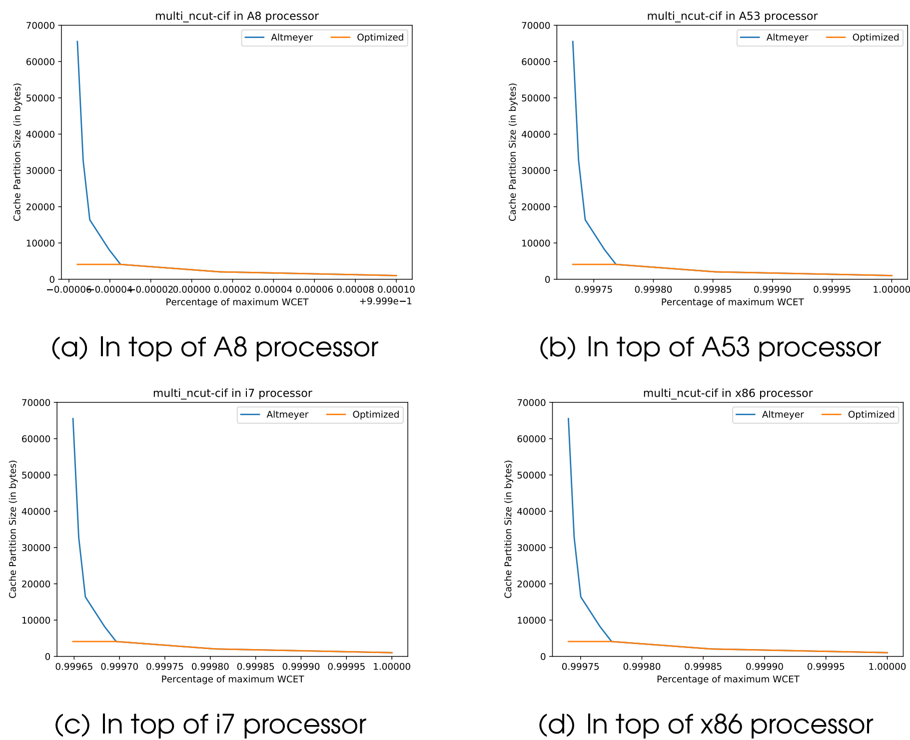
(d) In top of x86 processor

Figure 6.3 – Relation of cache partition size and maximum WCET for `lda`

Figure 6.4 continues the evaluation of the second experiment using `multi_ncut` benchmark. This benchmark shows an outstanding improvement by using the proposed algorithm.

6.3.3 Third Experiment: Schedulability

Third Experiment assess the schedulability of a synthetic generated task set in relation to the system utilization. As in Section 4.3, the task sets used in this



(a) In top of A8 processor

(b) In top of A53 processor

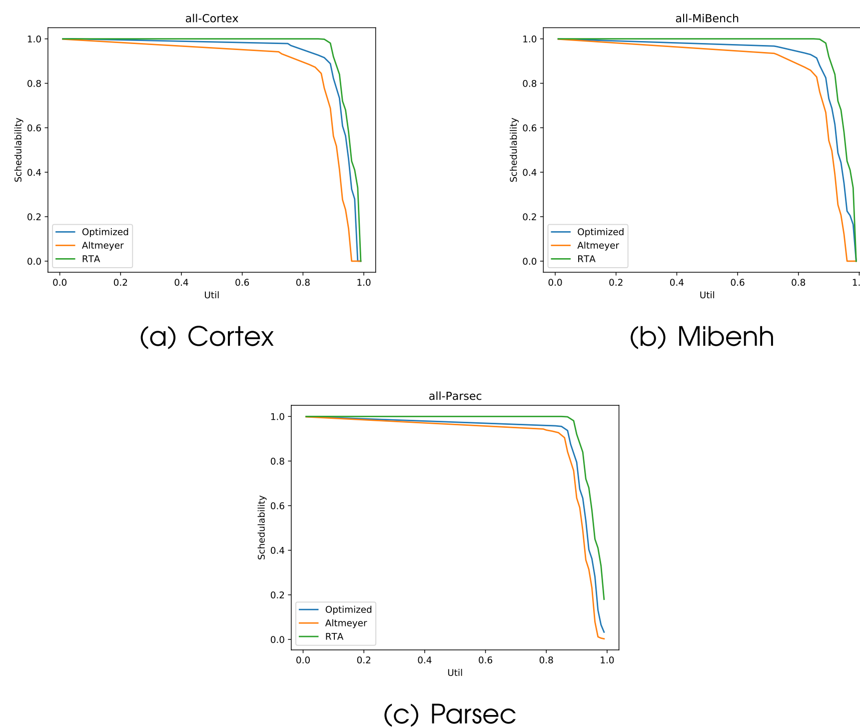
(c) In top of i7 processor

(d) In top of x86 processor

Figure 6.4 – Relation of cache partition size and maximum WCET for `lda`.

experiment were randomly generated using the impartial method proposed by Emberson et al. (EMBERSON; STAFFORD; DAVIS, 2010) to ensure a uniformly distributed random of utilization values. The utilization of the tasks was varied from 0 to 1 with a step of 0.05. Each experiment involved generating 10,000 sets with 10 tasks for each utilization value. The periods generated within the range of 10,00 to 1,000,000 microseconds. All task sets had implicit deadlines and were assigned priorities based on the Rate-Monotonic scheduling policy. For Schedulability analysis, it was used the response time analysis by Audsley et al. (AUDSLEY et al., 1993) and it was assumed that each task was assigned a separate cache partition. The experiment was repeated using the three benchmarks suite as task basis.

Figure 6.5 presents the schedulability rate results for OCPA, our optimized proposed algorithm and pure RTA analysis. The plots (a), (b) and (c) in the figure correspond to task set based on Cortex, Mibench and Parsec suite, respectively. It can be seen that the gain of our proposed algorithm over OCPA can reach values upon to 1% for Parsec, 4% for Mibench and 5% for Cortex. The results evidence the outperforming of the proposed algorithm in relation to OCPA in all cases and has presented even better performance for tasks sets based on Cortex suite, therefore showing an optimal approach for high workloads tasks sets.



(a) Cortex

(b) Mibenh

(c) Parsec

Figure 6.5 – Schedulability analysis for all benchmarks.

These findings highlight the effectiveness of the proposed flexible approach in improving schedulability rates for real-time systems by utilizing a cache partitioning method that take into account the cache behavior and the characteristics of the tasks, as well as the cache replacement policy.

6.4 PARTIAL CONSIDERATIONS

In this chapter, we proposed an algorithm for cache partitioning that aims to optimize cache utilization and enhance the schedulability of real-time tasks. The algorithm considers the specific cache requirements and timing constraints of tasks, allowing for the allocation of cache partitions tailored to their individual needs. It incorporates task profiling, partition size determination, partition allocation, cache replacement policy selection and validation steps to achieve these objectives.

- Task Profiling: The algorithm gathers information about each task's cache behavior, including cache misses, cache line utilization and working set size. This profiling step provides insights into the tasks' cache requirements and guides the subsequent partitioning decisions.
- Partition Size Determination: The algorithm determines the size of cache partitions for each task based on their cache requirements. It considers

factors such as WCET and cache replacement policy.

- Partition Allocation: Using the determined partition sizes, the algorithm allocates cache partitions to tasks. It ensures that each task is assigned a dedicated portion of the cache that meets its cache requirements while minimizing interference with other tasks.
- Cache Replacement Policy: The algorithm selects an appropriate cache replacement policy for each partition, considering the task's characteristics and cache utilization goals. It explores policies such as Least Recently Used (LRU) or Bimodal Insertion Policy (BIP) to optimize cache performance.
- Validation: The algorithm performs validation tests to evaluate the effectiveness of the cache partitioning scheme, considering the available cache capacity.

Through experimental evaluations, we compared the proposed algorithm with an approach that does not consider cache replacement policies. The results showed that the proposed algorithm outperforms the alternative approach in terms of cache utilization and schedulability. This demonstrates the effectiveness of considering cache replacement policies in cache partitioning for real-time systems.

Overall, the proposed algorithm provides a practical solution for optimizing cache utilization in real-time systems. By considering task-specific requirements and selecting appropriate cache configurations, it can improve system performance and meet timing constraints. Further research could explore additional factors, such as power consumption and energy efficiency, to enhance the algorithm's capabilities.

In the next chapter, we will present the final conclusions of this work and discuss potential future directions for research in the field of cache optimization for real-time systems.

7 CONCLUSION

In this dissertation, we investigated cache optimization techniques for real-time systems. Caches play a crucial role in improving system performance by reducing memory access latency. However, in real-time systems, the predictability of task execution is of utmost importance, and cache interference can significantly impact timing guarantees. Therefore, optimizing cache utilization while ensuring schedulability is a challenging task.

The main objectives of this research were: (i) conduct a comprehensive literature review of related works to gather insights and knowledge in the field of cache line replacement policies and their impact on real-time systems; (ii) evaluate the performance of popular cache line replacement policies, namely LRU, FIFO, RANDOM, LIP and BIP, in terms of cache misses and the scalability of real-time tasks using a diverse set of benchmarks; (iii) implement and assess the performance of the Dynamic Insertion Policy (DIP) in a cache simulator, specifically cachegrind, to determine its effectiveness in reducing cache misses; (iv) investigate and analyze the effects of different cache settings on the performance of cache line replacement policies, with a focus on their impact on critical real-time systems; and (v) develop an algorithm that optimizes cache parameters and replacement policies, considering the cache partitioning approach for tasks in a real-time system.

We began by exploring the impact of cache partition size and the number of cache ways on cache replacement policies. Through experimental evaluations, we observed that different parameter configurations can lead to varying cache failure rates across different applications. This highlighted the need for selecting optimal cache parameters and policies for each application. In the following, we summarize our results Section 7.1, present our closing remarks (Section 7.2) and discuss future work (Section 7.3).

7.1 SUMMARY OF CONTRIBUTIONS

Our research makes novel contributions in the following areas: (i) evaluation of cache replacement policies; (ii) implementation and evaluation of the DIP policy in Cachegrind and (iii) Cache Optimization Techniques. Here, we briefly recapitulate the key contributions from Chapter 4 to Chapter 6.

7.1.1 Evaluation of Cache Replacement Policies

The evaluation of cache replacement policies was a significant contribution of this research. We conducted experiments to compare the performance

of popular cache replacement policies, namely LRU, FIFO, RANDOM, LIP and BIP, in terms of cache misses and the scalability of real-time tasks. The experiments were conducted using a diverse set of benchmarks to assess the policies' effectiveness across different applications.

Through the evaluations, we gained valuable insights into the strengths and weaknesses of each replacement policy. We observed that the choice of replacement policy has a significant impact on cache performance and can affect the schedulability of real-time tasks. The results provided guidance on selecting the most appropriate replacement policy based on the specific requirements of the real-time system.

7.1.2 Implementation and Evaluation of the DIP Policy in Cachegrind

Another contribution of this research was the implementation and evaluation of the Dynamic Insertion Policy (DIP) in the Cachegrind cache simulator. The DIP policy was proposed as a novel cache replacement policy that aims to reduce cache misses by dynamically adjusting the insertion points of cache lines.

We implemented the DIP policy in Cachegrind and conducted experiments to evaluate its performance in terms of cache misses. The evaluation involved comparing the cache misses of DIP with other popular replacement policies, including LRU and FIFO, using various benchmark applications.

The results of the evaluation showed that the DIP policy achieved a significant reduction in cache misses compared to traditional replacement policies. This demonstrated the effectiveness of the DIP policy in improving cache performance and reducing memory access latency.

The implementation and evaluation of the DIP policy in Cachegrind provided empirical evidence of its benefits and highlighted its potential as an efficient cache replacement policy for real-time systems.

Overall, these contributions shed light on the performance of different cache replacement policies and provide insights into the effectiveness of the DIP policy in reducing cache misses. These findings contribute to the development of cache optimization techniques for real-time systems and serve as a basis for further research in this area.

7.1.3 Cache Optimization Techniques

The proposed algorithm takes into account the variations in memory access patterns and temporal characteristics of different tasks, as well as the impact of cache replacement policies. By dynamically selecting the best replace-

ment policies for each task in a per-task partitioned cache, the algorithm aims to optimize cache utilization and improve system performance.

Through experimental comparisons with the OCPA, the proposed algorithm demonstrates its effectiveness in terms of cache utilization, WCET, and schedulability. The evaluation results provide insights into the benefits and limitations of the algorithm in optimizing cache partitioning for hard real-time systems.

It is important to note that the proposed algorithm builds upon the foundation laid by Altmeyer et al.'s work on cache partitioning and extends it by considering the interference of replacement policies. By incorporating this additional factor, the proposed algorithm provides a more comprehensive approach to cache optimization.

However, further research and evaluation are necessary to fully understand the algorithm's performance across a broader range of benchmarks and real-time systems. Additionally, other factors such as power consumption and energy efficiency should be considered in future investigations.

Overall, the proposed algorithm presents a promising approach to cache partitioning and optimization in hard real-time systems. Its ability to adapt to varying task requirements and select the most suitable cache configurations can lead to improved system performance, better WCET and enhanced schedulability.

7.2 CLOSING REMARKS

Cache optimization plays a crucial role in enhancing the performance and predictability of real-time systems. This thesis has presented a comprehensive investigation of cache optimization techniques, focusing on cache partitioning and the impact of cache replacement policies.

The proposed cache partitioning algorithm, which incorporates the selection of optimal cache replacement policies for each task, has shown promising results in improving cache utilization and schedulability in real-time systems. By considering task-specific requirements and dynamically allocating cache partitions, our algorithm has demonstrated its effectiveness in optimizing system performance.

Future research and development in cache optimization techniques for real-time systems can further enhance the efficiency and schedulability of these systems. By exploring power and energy efficiency, integrating with dynamic voltage and frequency scaling, considering multilevel caches, evaluating on different hardware platforms, and addressing dynamic workloads, researchers can continue to advance the field and contribute to the development of more efficient and predictable real-time systems.

In conclusion, cache optimization techniques are vital for meeting the performance and timing requirements of real-time systems. The findings and algorithms presented in this thesis serve as a foundation for further research and can contribute to the development of more effective cache optimization techniques in the future.

7.3 FUTURE WORKS

While this dissertation has made significant contributions to cache optimization techniques for real-time systems, there are several avenues for further research and improvement. Some potential future directions include:

- Exploration of Power and Energy EfficiencyIn addition to performance considerations, power and energy efficiency are crucial factors in modern computing systems. Future research could explore cache optimization techniques that not only improve system performance but also minimize power consumption and enhance energy efficiency. This could involve dynamic power management strategies and cache partitioning algorithms that take power constraints into account.
- Consideration of Multilevel CachesMany modern processors employ multilevel cache hierarchies, consisting of multiple cache levels with varying capacities and access latencies. Extending cache optimization techniques to incorporate multilevel cache hierarchies could enhance the performance and schedulability of real-time systems. This could involve developing algorithms that allocate cache partitions across multiple cache levels and manage the cache hierarchy effectively.
- Evaluation on Different Hardware PlatformsThe evaluations conducted in this thesis were based on specific hardware platforms. Further research could expand the evaluation to include different hardware architectures and platforms. This would provide a more comprehensive understanding of the performance and effectiveness of cache optimization techniques across a broader range of systems.
- Consideration of Dynamic WorkloadsThe evaluations conducted in this thesis focused on static task sets and predefined workloads. Future research could explore cache optimization techniques for dynamic workloads, where the task execution patterns and memory access patterns change dynamically. This would involve developing adaptive cache partitioning algorithms that can adjust the cache configurations in real-time based on the current workload characteristics.

BIBLIOGRAPHY

Abel, A.; Reineke, J. Measurement-based modeling of the cache replacement policy. In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. (S.l.: s.n.), 2013. p. 65–74.

ABENI, L.; BUTTAZZO, G.; PALOPOLI, L. Task modeling in real-time systems. *IEEE Transactions on Software Engineering*, IEEE, v. 26, n. 11, p. 1044–1058, 2000.

ABENI, L.; BUTTAZZO, G. C.; LIPARI, G. Multiprocessor edf scheduling. *Real-Time Systems*, Springer, v. 19, n. 3, p. 235–255, 2000.

ABENI, L.; BUTTAZZO, G. C.; LIPARI, G. Schedulability analysis of periodic task systems using off-line and on-line algorithms. *IEEE Transactions on Computers*, IEEE, v. 52, n. 8, p. 1051–1062, 2003.

AL-ZOUBI, H.; MILENKOVIC, A.; MILENKOVIC, M. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: *Proceedings of the 42nd Annual Southeast Regional Conference*. New York, NY, USA: Association for Computing Machinery, 2004. (ACM-SE 42), p. 267–272. ISBN 1581138709. Disponível em: <https://doi.org/10.1145/986537.986601>.

ALTMEYER, S. et al. Outstanding paper: Evaluation of cache partitioning for hard real-time systems. In: *2014 26th Euromicro Conference on Real-Time Systems*. (S.l.: s.n.), 2014. p. 15–26.

ANDERSON, J.; BARUAH, S.; KATCHER, J. Uniprocessor and multicore real-time scheduling with hierarchical deadlines. *Real-Time Systems*, Springer, v. 53, n. 1, p. 22–66, 2017.

ARAUJO, B. A. et al. Implementation and evaluation of adaptive cache insertion policies for real-time systems. In: *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*. (S.l.: s.n.), 2021. p. 1–8.

AUDSLEY, N. et al. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, v. 8, n. 5, p. 284–292, set. 1993.

AYAV, T.; BARUAH, S. K. Resource reservation techniques for real-time systems: a survey. *Real-Time Systems*, Springer, v. 27, n. 2-3, p. 155–190, 2004.

AYDIN, H.; JOHANSSON, K. H.; SASTRY, S. *Ciber-Física: Modelagem, Análise e Controle*: Cyber-physical systems: Modeling, analysis, and control. (S.l.: s.n.), 2018.

BAKER, T. P. Analysis of rate-monotonic scheduling. *Real-Time Systems*, Springer, v. 21, n. 2, p. 137–155, 2001.

BANSAL, A. et al. Evaluating the memory subsystem of a configurable heterogeneous MPSoC. In: *Proc. of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2018)*. (S.l.: s.n.), 2018. p. 55–60.

BARUAH, S.; FISHER, N. Improved approximation algorithms for the multiprocessor scheduling problem with real-time and energy constraints. *Real-Time Systems*, Springer, v. 32, n. 1-3, p. 209–231, 2006.

BARUAH, S.; GOOSSENS, J. Scheduling real-time tasks: a survey. In: *Real-Time Systems*. (S.l.): Springer, 2005. p. 85–128.

BARUAH, S.; GOOSSENS, J.; SRINIVASAN, R. Scheduling real-time tasks with arbitrary deadlines using dynamic self-suspension. *Real-Time Systems*, Springer, v. 43, n. 1-3, p. 147–198, 2009.

BIENIA, C. *Benchmarking Modern Multiprocessors*. Tese (Doutorado) — Princeton University, January 2011.

BLETSAS, K.; SOUDRIS, D. *Real-Time Cache Memory Management*. (S.l.): Springer, 2009.

BRANDENBURG, B.; ANDERSON, J. H.; BARUAH, S. Dynamic cache partitioning for hard real-time systems. *Real-Time Systems*, Springer, v. 46, n. 2, p. 179–214, 2010.

BURKS, A. W.; GOLDSTINE, H. H.; NEUMANN, J. von. Preliminary discussion of the logical design of an electronic computing instrument. In: . (s.n.), 1946. Disponível em: <https://www.ias.edu/sites/default/files/library/Prelim_Disc_Logical_Design.pdf>.

BURNS, A.; WELLINGS, A. J. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. (S.l.): Pearson Education, 2015.

BUTTAZZO, G. C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. ed. (S.l.): Springer Publishing Company, Incorporated, 2011. ISBN 1461406757.

CACCAMO, M.; BUTTAZZO, G.; LIPARI, G. *Memory Interference Effects in Real-Time Systems*. (S.l.): Springer, 2002.

CERVIN, A.; HENRIKSSON, D.; LINCOLN, B. *Introduction to the modeling and analysis of real-time systems*. (S.l.): CRC Press, 2018.

CHEN, C.; HAN, Y.; BURNS, A. Compiler-directed scratch-pad memory management for real-time systems with mixed-criticality workloads. In: *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS '04)*. (S.l.: s.n.), 2004. p. 204–214.

DAVIS, R. I. et al. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real Time Syst.*, v. 35, n. 3, p. 239–272, 2007.

DOBBING, M. J.; DYSTER, A. B.; GERNDT, M. Improving wcet analysis by predicting the worst-case number of cache misses. In: *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '07)*. (S.l.: s.n.), 2007. p. 543–552.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. Techniques For The Synthesis Of Multiprocessor Tasksets. In: *WATERS at the Euromicro Conference on Real-Time Systems*. [S.l.: s.n.], 2010. p. 6–11.

GRACIOLI, G. et al. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 48, n. 2, nov. 2015. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/2830555>.

GRACIOLI, G.; FRöHLICH, A. A. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In: *Proc. of the 19th IEEE RTCAS*. [S.l.]: IEEE, 2013.

GRACIOLI, G. et al. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In: *31st ECRTS*. [S.l.: s.n.], 2019. p. 27:1–27:25.

GUSTAFSSON, J. et al. The Mälardalen WCET benchmarks – past, present and future. In: LISPER, B. (Ed.). *WCET2010*. Brussels, Belgium: OCG, 2010. p. 137–147.

Guthaus, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. [S.l.: s.n.], 2001. p. 3–14.

HANDY, J. *The Cache Memory Book*. Elsevier Science, 1998. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780123229809. Disponível em: <https://books.google.com.br/books?id=-7oOlb-ICpMC>.

HARDY, D.; PUAUT, I.; PAUTET, L. Cache-related preemption delays in real-time systems. *Real-Time Systems*, Springer, v. 29, n. 2-3, p. 171–205, 2005.

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.

HOORNAERT, D. et al. Improving real-time system schedulability through per-task cache eviction policy selection. In: *2019 IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 2021. p. 1–13.

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>.

LEHOCZKY, J. P.; SHA, L.; DING, Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of the IEEE*, IEEE, v. 81, n. 1, p. 62–72, 1989.

LESAGE, B.; PUAUT, I.; SEZNEC, A. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. New York, NY, USA: Association

for Computing Machinery, 2012. (RTNS '12), p. 171–180. ISBN 9781450314091. Disponível em: <https://doi.org/10.1145/2392987.2393009>.

Liedtke, J.; Hartig, H.; Hohmuth, M. Os-controlled cache predictability for real-time systems. In: *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. [S.l.: s.n.], 1997. p. 213–224.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 20, n. 1, p. 46–61, jan. 1973. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321738.321743>.

LIU, J. W. S. W. *Real-Time Systems*. 1st. ed. USA: Prentice Hall PTR, 2000. ISBN 0130996513.

MANCUSO, R.; YUN, H.; PUAUT, I. Impact of dm-lru on wcet: a static analysis. In: . [S.l.: s.n.], 2019.

NELISSEN, G. et al. Real-time cache replacement policies: A survey and empirical evaluation. *ACM Computing Surveys (CSUR)*, ACM, v. 50, n. 5, p. 74, 2017.

OLIVEIRA, R. S. de. *Fundamentos dos Sistemas de Tempo Real*. 2nd. ed. Florianópolis-SC: Edição do autor, 2020. ISBN 0130996513.

PALEM, K. K.; KUNDU, S. Modeling and optimizing embedded memory architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, v. 8, n. 4, p. 426–455, 2003.

PATEL, K.; PERUMALLA, K. S. Characterizing the impact of cache partitioning on real-time performance. In: *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. [S.l.: s.n.], 2015. p. 1–10.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269.

PAUTET, L.; PUAUT, I.; PELLETIER, S. Cache optimization techniques for real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, v. 3, n. 2, p. 380–401, 2004.

PUAUT, I.; PAIS, C. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In: *2007 Design, Automation Test in Europe Conference Exhibition*. [S.l.: s.n.], 2007. p. 1–6.

QURESHI, M. et al. A case for MLP-aware cache replacement. In: *33rd International Symposium on Computer Architecture (ISCA'06)*. [S.l.: s.n.], 2006. p. 167–178.

QURESHI, M. K. et al. Adaptive insertion policies for high performance caching. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing

Machinery, 2007. (ISCA '07), p. 381–391. ISBN 9781595937063. Disponível em: <https://doi.org/10.1145/1250662.1250709>.

QURESHI, M. K.; SRINIVASAN, V.; RIVERS, J. A. A case for inclusive, massively parallel caches. *ACM SIGARCH Computer Architecture News*, ACM, v. 35, n. 2, p. 291–302, 2007.

REINEKE, J. et al. Selfish-LRU: Preemption-aware caching for predictability and performance. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. (S.l.: s.n.), 2014. p. 135–144.

REINEKE, J. et al. Predictability of cache replacement policies. *Real-Time Systems*, v. 37, p. 99–122, 09 2007.

SANCHEZ, D.; KOZYRAKIS, C. Vantage: Scalable and efficient fine-grain cache partitioning. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. (S.l.: s.n.), 2011. p. 57–68.

SEWARD, J. et al. *Valgrind Documentation*. (S.l.), 2021. Disponível em: <https://valgrind.org/docs/manual/valgrind_manual.pdf>.

SHA, L.; RAJKUMAR, R. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, IEEE, v. 53, n. 12, p. 1612–1629, 2004.

SMITH, J. E. Cache memories. *ACM Computing Surveys*, v. 14, n. 3, p. 473–530, 1982.

SPRUNT, B.; BURNS, A.; BERNAT, G. Partitioned scheduling of hard sporadic tasks: an integrated approach. *Real-Time Systems*, Springer, v. 23, n. 1-2, p. 73–100, 2002.

SRINIVASAN, R.; BARUAH, S.; GOOSSENS, J. Optimality of earliest deadline first scheduling for sporadic task systems with arbitrary deadlines. *Real-Time Systems*, Springer, v. 47, n. 3, p. 237–264, 2011.

STALLINGS, W. *Computer Organization and Architecture - Designing for Performance (8. ed.)*. (S.l.): Pearson / Prentice Hall, 2009. ISBN 978-0-13-185644-8.

STANKOVIC, J. A.; RAMAMRITHAM, K. *Real-time systems*. (S.l.): John Wiley & Sons, 1998.

SUBRAMANIAN, R.; SMARAGDAKIS, Y.; LOH, G. H. Adaptive caches: Effective shaping of cache behavior to workloads. In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. (S.l.: s.n.), 2006. p. 385–396.

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>.

SUN, Q.; FAN, L.; DENG, Y. Cache partitioning for mixed-criticality real-time systems. In: *2018 IEEE 25th International Conference on Real-Time Networks and Systems (RTNS)*. (S.l.: s.n.), 2018. p. 129–138.

TANENBAUM, A. *Structured computer organization (5. ed.).* (S.l.: s.n.), 2006. ISBN 978-0-13-148521-1.

THOMAS, S. et al. CortexSuite: A Synthetic Brain Benchmark Suite. In: *International Symposium on Workload Characterization (IISWC)*. (S.l.: s.n.), 2014.

VENKATASUBRAMANIAN, N. et al. Dynamic management of scratch-pad memories in embedded real-time systems. In: *Proceedings of the 12th International Conference on Real-Time and Embedded Systems and Applications (RTCSA '06)*. (S.l.: s.n.), 2006. p. 405–414.

WILHELM, R.; ENGBLOM, J. (Ed.). *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*. (S.l.): Springer Science & Business Media, 2008.

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.

ZHAO, L.; MOSSE, D.; KNAG, P. V. Evaluation of cache replacement policies for real-time systems. In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*. (S.l.: s.n.), 1997. p. 44–54.