

Interaction Between Computer Architecture and Artificial Intelligence



Alexandre Valentin Jamet

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

Je dédie cette thèse de doctorat à ma famille et mes amis.

Leur soutien indéfectible, leurs encouragements et leur amour ont été une source constante d'inspiration tout au long de ce parcours. Cette réalisation est autant la leur que la mienne, et je suis profondément reconnaissant de leur présence à mes côtés.

DECLARATION

I, Alexandre Valentin Jamet, declare that this thesis, entitled "*Interaction Between Computer Architecture and Artificial Intelligence*" is entirely my own work. Where reference is made to the works of others, it is duly acknowledged and attributed. This thesis has not been submitted for any other degree or qualification at any other university.

I understand that plagiarism is the unacknowledged inclusion of another person's work or ideas and can result in severe penalties. I have read and understood the UPC's policy on plagiarism and declare that this thesis is my original work and has been written in accordance with the ethical standards set by the UPC.

Any assistance I received in the preparation of this thesis, including technical, financial, and intellectual support, is acknowledged in the text and, where applicable, in the acknowledgment section of the thesis.

Alexandre Valentin Jamet
March 2024

ABSTRACT

SINCE its inception with the first computing systems, computer architecture has lived through many revolutions and saw plenty of technological innovations. However, a significant challenge has persisted throughout the evolution of computing systems: the Memory Wall. To address this challenge, architects have devised various latency tolerance techniques, including cache hierarchy, cache replacement policies, hardware prefetching, and off-chip prediction.

Cache hierarchy involves the use of intermediate memories, such as caches, to store frequently accessed data close to the processor, thereby reducing memory access latencies. Cache replacement policies determine which data blocks should be stored or evicted from caches based on predictions of future reuse. Hardware prefetching mechanisms aim to bring data blocks that are likely to be needed in the near future into the cache proactively. Off-chip prediction predicts whether a load demand request will benefit from cache access or if it will require a DRAM access, allowing for speculative fetching of data blocks from DRAM to hide memory access latencies.

This thesis addresses the challenges of cache management and memory access optimization in modern computer architectures, focusing on improving performance and energy efficiency across a variety of workloads. It presents three main contributions.

The first contribution critically assesses the effectiveness of contemporary Last Level Cache (LLC) replacement policies across a diverse spectrum of workloads, encompassing graph processing, scientific, industrial applications, as well as standard benchmark suites like SPEC CPU 2006 and SPEC CPU 2017. Despite exhibiting notable performance enhance-

ments in conventional benchmark scenarios, these existing LLC replacement policies often falter in capturing the nuanced access patterns characteristic of modern High-Performance Computing (HPC) and big data workloads. In response to this challenge, two novel LLC replacement policies, namely Multi-Sampler Multiperspective (MS-MPPPB) and Multiperspective with Dynamic Features Selector (DS-MPPPB), are introduced and rigorously evaluated. Demonstrating superior efficacy across a broad array of workloads, these innovative policies offer heightened performance benefits tailored specifically for HPC and big data applications.

The second contribution is dedicated to enhancing memory access patterns tailored specifically for graph-processing workloads, renowned for their irregular memory access patterns and suboptimal data locality. Introducing the innovative Large Predictor (LP), this endeavor aims to discern between regular and irregular memory accesses, channeling irregular accesses efficiently through a dedicated Side Data Cache (SDC). By synergizing LP with SDC, notable performance enhancements are achieved, surpassing conventional cache hierarchies and state-of-the-art cache replacement policies, particularly within the realm of graph-processing applications.

The third contribution presents the Two Level Perceptron (TLP) predictor, a sophisticated approach that integrates off-chip prediction with adaptive prefetch filtering within the first-level data cache (L1D). Leveraging a dual-layered structure composed of the First Level Predictor (FLP) and Second Level Predictor (SLP), TLP effectively mitigates average DRAM transactions while enhancing overall performance across both single-core and multi-core workloads.

Collectively, these contributions advance the state-of-the-art in cache management and memory access optimization, providing insights and techniques to enhance the performance and energy efficiency of modern computer architectures across a variety of workloads.

These contributions advance the state-of-the-art in cache management and memory access optimization, offering insights and techniques to enhance the performance and energy efficiency of modern computer architectures across a variety of workloads.

THESIS ORGANIZATION

THIS doctoral dissertation is divided into eight chapters. These chapters guide the reader through the work done over the time of this PhD. The dissertation's structure is the following.

In Chapter 1, we delve into the Memory Wall bottleneck, which serves as the cornerstone for understanding cache management policies. This chapter covers the fundamental principles of the memory hierarchy, both at a basic and advanced level. Furthermore, it highlights the importance of enhancing, through novel methods, cache management policies and outlines the key contributions of this doctoral thesis.

Chapter 2 furnishes comprehensive background information concerning the Memory Wall, the intricate memory hierarchy found in contemporary processors, and the intricate landscape of cache management policies. Furthermore, it offers an extensive exploration of work characterization, cache replacement strategies, hardware prefetching techniques, and off-chip prediction methods. This chapter forms a robust foundation of knowledge essential for readers to thoroughly grasp the content presented in this doctoral thesis.

In Chapter 3, we immerse ourselves in the initial contribution of this doctoral thesis. This work provides a comprehensive assessment and characterization of contemporary workloads, highlighting the influence of modern cache replacement policies on their behavior. Furthermore, this chapter unveils two micro-architectural design proposals aimed at mitigating cache bypassing issues within the last-level cache. This chapter is based on contributions presented at the 2020 IEEE International Symposium on Workload Characterization (IISWC 2020).

Chapter 4 presents the Side Data Cache (SDC), a development of the memory hierarchy designed to isolate memory request predicted to go off-chip in an effort to clean up the access

patterns. This chapter is based on contributions presented in the 2024 IEEE International Parallel & Distributed Processing Symposium (IPDPS 2024).

Chapter 5 presents the Two-Layer Predictor, an evolution of the Hermes prediction mechanism that aims to reduce the main memory bandwidth consumption through a combination of off-chip prediction, and off-chip prediction aided prefetch filtering. This chapter is based on contributions presented at the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA 2024). In addition, Appendix A presents the Artifact Evaluation (AE) provided to reviewers of the HPCA 2024 conference.

Chapter 6 summarizes the contributions of this doctoral thesis while discussing its broader impact and acknowledging its main supporters.

Chapter 7 presents a list of publications that have been accepted in peer-reviewed conferences as well as on-going works and collaborations.

CONTENTS

Thesis Organization	ix
List of Figures	xvii
List of Tables	xxi
1 Introduction	1
1.1 The Processor-Memory Gap	3
1.2 Exploring Novel Avenues: AI-Inspired Designs in Computer Architecture .	4
1.2.1 Artificial Neural Networks: Emulating Human Intelligence	5
1.2.2 Applying AI Techniques to Enhance Computer Architecture	7
1.3 Motivation	7
1.4 Thesis Approach	9
1.5 Thesis Contributions	9
1.5.1 Characterizing the impact of last-level cache replacement policies on big-data workloads	10
1.5.2 Practically Tackling Memory Bottlenecks of Graph-Processing Work- loads	11
1.5.3 A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering	12

2	Background	15
2.1	The Processor-Memory Gap	16
2.2	Cache Replacement Policies	18
2.2.1	The Optimal Replacement Strategy: Belady's MIN Algorithm . . .	19
2.2.2	Reuse Distance Prediction	21
2.2.3	Signature-based Hit Predictor	22
2.2.4	Multiperspective Reuse Prediction	23
2.2.5	Optimal Replacement Approximation	26
2.2.5.1	The <i>OPTgen</i> algorithm	26
2.2.5.1.1	Usage and Liveness Intervals:	26
2.2.5.1.2	Occupancy Vector:	27
2.2.5.1.3	Training the predictor with <i>OPTgen</i> :	29
2.2.5.1.4	Issuing an <i>OPTgen</i> prediction:	29
2.2.5.2	Operation of the Hawkeye Replacement Policy	29
2.2.5.3	Advancements in the Hawkeye Replacement Policy . . .	30
2.3	Hardware Prefetching	30
2.3.1	The IPCP prefetcher	31
2.3.1.1	Constant-stride Prefetching	32
2.3.1.2	Complex-stride Prefetching	32
2.3.1.3	Global-stream Prefetching	33
2.3.2	The Berti prefetcher	34
2.3.2.1	Training the Prefetcher	35
2.3.2.2	Issuing predictions	36
2.3.3	The IMP prefetcher	36
2.3.4	The DROPLET prefetcher	39
2.4	Off-Chip Prediction	40
2.4.1	LP: Cache Level Prediction	40
2.4.2	Hermes: Off-Chip Load Prediction	42
3	Characterizing the impact of last-level cache replacement policies on big-data workloads	45
3.1	Introduction	46
3.2	Workloads	47
3.2.1	SPEC CPU Benchmark Suites	48
3.2.2	GAP Benchmark Suite	48
3.2.2.1	Graph kernels	48
3.2.2.2	Input graphs	49

3.2.3	XSbench workloads	50
3.2.3.1	Problem size	50
3.2.3.2	Grid type	50
3.2.3.3	Number of cross-section look-ups	50
3.2.4	Industrial workloads	51
3.3	Cache Replacement Policies	51
3.3.1	Reuse Distance Prediction	51
3.3.2	Signature-based Hit Predictor	52
3.3.3	Multiperspective Reuse Prediction	52
3.3.4	Optimal Replacement Approximation	53
3.4	Motivation	54
3.5	Design proposals	56
3.5.1	Multi-Sampler Multiperspective	56
3.5.2	Multiperspective with Dynamic Features Selector	57
3.6	Methodology	58
3.6.1	Workloads	58
3.6.2	Experimental setup	59
3.6.3	Replacement policies simulated	59
3.7	Results, analysis and discussion	60
3.7.1	Misses	60
3.7.2	Performance	61
3.7.3	Reuse distances	62
3.7.3.1	SPEC benchmarks	63
3.7.3.2	XSbench benchmarks	64
3.7.3.3	Qualcomm benchmarks	65
3.7.3.4	GAP benchmarks	65
3.8	Summary	68
4	Practically Tackling Memory Bottlenecks of Graph-Processing Workloads	69
4.1	Introduction	70
4.2	Background and Motivation	72
4.2.1	Graph-Processing Workloads	72
4.2.2	Characterizing the Memory Hierarchy Behavior of Graph-Processing Workloads	73
4.3	Managing Memory Accesses of Graph-Processing Workloads	76
4.3.1	Side Data Cache (SDC)	76
4.3.2	Large Predictor (LP)	77

4.3.2.1	LP Operation	78
4.3.2.2	LP Update	78
4.3.2.3	LP Replacement	79
4.3.3	Coherence	79
4.3.4	Putting It All Together	80
4.3.5	Context Switches	81
4.4	Methodology	81
4.4.1	Simulation Infrastructure	81
4.4.2	Graph-Processing Applications	82
4.4.3	Single-Threaded Workloads	83
4.4.4	Multi-Threaded Workloads	83
4.4.5	Alternative Approaches	84
4.5	Evaluation	84
4.5.1	Single-Core Evaluation	85
4.5.2	Design Space Exploration	87
4.5.2.1	SDC Size	87
4.5.2.2	LP Organization	89
4.5.2.3	Global Threshold	90
4.5.3	Comparison with An Expert Programmer	90
4.5.4	Multi-Core Evaluation	91
4.5.5	Evaluation on SPEC Benchmarks	93
4.5.6	Hardware Cost	93
4.6	Related work	93
4.6.1	Replacement Policies	94
4.6.2	Hardware Prefetching	94
4.6.3	Memory System Optimizations for Graph-Processing	94
4.6.4	Pre-processing Algorithms	95
4.6.5	Graph-Processing Accelerators	95
4.7	Summary	96

5	A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering	99
5.1	Introduction	100
5.2	Background	102
5.2.1	Off-Chip Prediction	102
5.2.2	Prefetch Filtering	103
5.3	Motivation	103

5.3.1	Cache Behavior of Modern Workloads	104
5.3.2	Impact of Hermes	104
5.3.2.1	DRAM Transactions	105
5.3.2.2	Analysis of Hermes Predictions	106
5.3.3	Off-Chip Prediction for L1D Prefetch Filtering	107
5.4	Two Level Perceptron Prediction	109
5.4.1	First Level Perceptron (FLP) Predictor	110
5.4.2	Second Level Perceptron (SLP) Predictor	112
5.4.3	Building a Multi-Level Perceptron Predictor	114
5.4.4	TLP Hardware Requirements and Latency	115
5.5	Experimental Methodology	116
5.5.1	Simulation Methodology	116
5.5.2	Workloads	116
5.5.3	Single-Core Evaluation	117
5.5.4	Multi-Core Evaluation	118
5.5.5	Alternative Techniques	119
5.6	Evaluation	119
5.6.1	Single-Core Evaluation	119
5.6.2	Multi-Core Evaluation	121
5.6.2.1	Performance Contribution of Each TLP Component	123
5.6.2.2	Sensitivity Analysis on DRAM Bandwidth	124
5.6.3	Designs Enhanced with TLP's Storage Budget	125
5.7	Related Work	126
5.7.1	Hit/Miss Prediction	127
5.7.2	Data Prefetching	128
5.7.3	Cache Bypassing	128
5.7.4	Memory Optimizations for Graph-Processing Applications	128
5.7.5	Redesigning the Cache Hierarchy	128
5.8	Summary	129
6	Conclusion	131
6.1	Future Work	132
6.1.1	Cache Hierarchy and Cache Level Prediction	133
6.1.2	Integration with Advanced Prefetching Techniques	133
6.1.3	Scalability, Energy Efficiency, and DRAM Bandwidth Considerations	133
6.1.4	Enhancing Neural Predictors and Integration with Emerging Memory Technologies	134

6.1.5	Integration within GPUs	134
6.1.6	Final Words	134
7	Publications	137
	References	139
Appendix A Artifact Evaluation – A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering		151
A.1	Abstract	151
A.2	Artifact check-list (meta-information)	152
A.3	Description	152
A.3.1	How to access	152
A.3.2	Hardware dependencies	152
A.3.3	Software dependencies	153
A.3.4	Data sets	153
A.4	Installation	153
A.5	Experiment workflow	153
A.6	Evaluation and expected results	154
A.7	Methodology	154

LIST OF FIGURES

1.1	Disparity between processor and main memory speeds.	4
1.2	Winged TAAΩN (Talos) armed with a stone.	5
1.3	Diagram representing a perception.	6
2.1	Disparity between processor and main memory speeds.	17
2.2	Anatomy of a modern cache hierarchy composed of three cache levels. The latency and capacity parameters are extracted from a recent Cascade Lake chip [26].	18
2.3	Operation of a cache and its replacement policy.	19
2.4	Example of Belady's MIN algorithm applied to an example sequence of memory accesses. Green boxes indicate the newly inserted blocks.	20
2.5	Example of a simplified Multiperspective design leveraging two features based on the PC and the physical address.	24
2.6	Example of the process of updating the occupancy vector for a cache set. In this example, <i>OPTgen</i> is applied to a cache set containing two ways.	28
2.7	Representation of a graph's adjacency matrix in memory <i>Compressed Sparse Column/Compressed Sparse Row</i> (CSC/CSR) formats. Graph's adjacency matrix are commonly represented as sparse matrices in memory as they usually contain a significant amount of zeros. In addition to the adjacency matrix, this figure represents the <i>Offset Array</i> (OA) and the <i>Neighbours Array</i> (NA).	37

3.1	LLC MPKI using the LRU replacement policy and performance improvement of state-of-the-art LLC replacement policies for the different benchmark suites.	55
3.2	LLC MPKI using the LRU replacement policy and performance improvement of LLC replacement policies for the 50 most intensive workloads.	61
3.3	Distribution of the reuse distances of the cache blocks per benchmark suite.	63
3.4	Distribution of the reuse distances of the cache blocks for the different benchmark suites.	64
3.5	LLC MPKI reduction and speedup of a 16 LLC over the baseline 2 LLC. . .	67
4.1	Graph representations using the CSR/CSC format.	72
4.2	Misses-Per-Kilo-Instruction (MPKI) across the different levels of the cache hierarchy triggered by graph-processing workloads.	75
4.3	Probability of accessing DRAM for accesses exhibiting different strides. Data corresponds to the <i>cc.friendster</i> workload.	76
4.4	Operation of the LP on a prediction event.	78
4.5	Organization of the LP predictor and a functional example of its update sequence.	79
4.6	Hardware support for SDC coherence.	80
4.7	Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture.	85
4.8	L2C and LLC MPKI ratios of SDC+LP and Baseline.	86
4.9	L1D and SDC MPKI ratios of SDC+LP and Baseline.	87
4.10	Exploration on the size of the SDC.	88
4.11	Speed-up of SDC+LP considering a fully-associative LP with different entry counts.	89
4.12	Speed-up of SDC+LP considering different LP associativities.	90
4.13	Speed-up over Baseline of SDC+LP and the Expert Programmer approach, which manually categorises cache averse and cache friendly accesses. . . .	91
4.14	Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture in a multi-core scenario.	92
5.1	MPKI of all caches (L1D, L2C, LLC) across the SPEC (SPEC CPU 2006 and SPEC CPU 2017) and GAP workloads.	104
5.2	Increase in DRAM transactions due to Hermes off-chip predictions relative to a baseline without off-chip prediction mechanism. X-axis ticks represent different SPEC and GAP workloads. Both SPEC and GAP workloads are separately sorted based on LLC MPKI. Lower is better.	105

5.3	Increase in DRAM transactions due to Hermes off-chip predictions relative to a baseline without off-chip prediction mechanism in the 4-core context. X-axis ticks represent 200 different 4-core workload mixes of SPEC and GAP workloads. Both SPEC and GAP workload mixes are separately sorted based on LLC MPKI. Lower is better.	105
5.4	Location of a block upon a Hermes off-chip prediction. A large fraction of the inaccurate off-chip predictions could have been avoided if Hermes would wait to issue a speculative DRAM access until the L1D lookup is resolved.	106
5.5	Location where the inaccurate L1D prefetch requests are served across two state-of-the-art L1D prefetchers. Both SPEC and GAP workloads are separately sorted based on LLC MPKI, similar to Figure 5.2.	108
5.6	Location where the accurate L1D prefetch requests are served across two state-of-the-art L1D prefetchers. Both SPEC and GAP workloads are separately sorted based on LLC MPKI, similar to Figure 5.2.	109
5.7	Flowchart of FLP. Diamonds indicate decision points.	111
5.8	Flowchart of SLP. Diamonds indicate decision points.	113
5.9	Organization and operation of the Two Level Perceptron (TLP) prediction mechanism.	114
5.10	Performance evaluation in the single-core scenario.	120
5.11	Increase in DRAM transactions in the single-core scenario. Lower is better.	121
5.12	Accuracy of the L1D prefetchers.	122
5.13	Performance improvement in the multi-core scenario.	123
5.14	Increase in DRAM transactions in the multi-core scenario. Lower is better.	124
5.15	Performance contribution of each TLP component.	125
5.16	Impact of DRAM bandwidth on geometric mean performance and average number of DRAM transactions in the multi-core context.	126
5.17	Performance improvement of designs enhanced with TLP's storage budget.	127

LIST OF TABLES

3.1	Hardware budget of the proposed designs.	58
3.2	System simulation parameters.	60
3.3	Main parameters of the distribution of reuse distances for cache blocks in the different benchmakrs.	62
4.1	System configuration	82
4.2	Graph kernels	83
4.3	Input Graphs	83
4.4	Hardware budget per core	93
5.1	List of features used by the FLP and the SLP.	110
5.2	Storage overhead of TLP.	115
5.3	System configuration.	117
5.4	Graph kernels	117
5.5	Input Graphs	118

CHAPTER

1

INTRODUCTION

Le véritable voyage de découverte ne
consiste pas à chercher de nouveaux
paysages, mais à avoir de nouveaux
yeux.

Marcel Proust, La Prisonnière, 1923

THE advent of modern workloads and their ever-increasing demands on computational and memory resources has pushed the field of computer architecture to its limits. This *renaissance* of the field has led to the emergence of a wide range of new ideas. In the last few years, architects have started re-imagining the designs at the heart of modern processors' micro-architecture to cut through Moore's Law [67, 68, 22] and further push the limits of performance. Similarly, the industry sends clear signals that a *renaissance* of computer architecture is currently at play. In recent years, several tech giants such as Google, Meta, Tesla, Apple, OpenAI, and many more [7] have started to design and build custom hardware tailored to the specific needs of the workloads they are running. Equally, modern *High Performance Computing* (HPC) facilities leverage more and more custom hardware. Common examples of such usage encompass accelerators built on *Field Programmable Gate Arrays* (FPGAs), *Tensor Processing Units* (TPUs), etc.

Recent scientific literature and industry feedback have shown that emergent workloads exhibit high complexity. This complexity manifests itself in many instances in modern micro-architecture. For example, server workloads have been shown to incur very high stress on virtual memory systems. Server workloads, for both instructions and data, usually exhibit enormous footprints, incompatible with the small size of the *Translation Lookup Buffers* (TLBs) and *L1 Instruction* (L1I) cache. Indeed, it is common for several server workloads to be executed on the same core (*i.e.*, SMT- n workloads, where n is the number of workloads interleaved on a single processor's core) [99, 34]. While this allows running several workloads in parallel, this incurs additional pressure on the cache hierarchy as unrelated workloads' access patterns and memory footprint interleave. Similarly, due to their data inputs spanning gigabytes of data, graph-processing workloads greatly stress the cache hierarchy and the entire memory system.

Tackling the highly complex behavior of emergent workloads with classical computer architecture solutions has proven challenging. Until now, classical prediction mechanisms have relied on assumptions that remained true until the introduction of these workloads: (i) workloads exhibit temporal and spatial locality, (ii) individual instructions only access a limited portion of the memory. In recent years, to combat the high complexity of modern workloads, computer architects have introduced concepts from artificial intelligence to build better and more resilient micro-architectural prediction mechanisms. Techniques derived from the artificial intelligence field answer the need for methods capable of extracting knowledge from vastly complex data sets. Using this information to take well-informed actions will eventually boost computer system performance. In 2001, Jimenez and Lin, in their paper "*Dynamic Branch Prediction with Perceptrons*" [50] were the first to successfully introduce a perceptron [83] – the simplest form of neural networks – in the micro-architecture

of a processor to achieve branch prediction. This first attempt, even though primitive – as it only relies on a single input (*i.e.*, the branch addresses) to perform its predictions –, is the start of a new line of micro-architectural prediction mechanisms that nowadays spread all across the board [96, 51, 45, 87, 19].

However, to fully understand the research direction of this thesis, we must first introduce a few concepts. In Section 1.1, we briefly discuss the Processor-Memory Gap, while in Section 1.2, we swiftly cover the main ideas behind AI-inspired designs.

1.1 The Processor-Memory Gap

In modern computing, the relentless pace of technological advancement has propelled computational capabilities to unprecedented heights. However, amid the triumphs of faster processors, enhanced micro-architectures, and sophisticated algorithms, a persistent challenge has remained for decades — the *Processor-Memory Gap*. This gap encapsulates the disconcerting disparity between the computational speed of processors and the comparatively sluggish pace at which memory systems (*i.e.*, in most cases DRAM main memories) can deliver data. Historically, the evolution of processors has outpaced the developments in memory technologies. While processors have undergone remarkable improvements in clock speeds, parallelism, and power efficiency, the rate of accessing and retrieving data from the main memory has not kept pace. The result is a growing disjunction between the time it takes processors to execute instructions and the time it takes to fetch the necessary data from the main memory. Figure 1.1 showcases the aforementioned gap in speed between processors and memory systems.

The *Processor-Memory Gap* is not merely an abstract concern for computer architects and hardware designers; it has profound implications for the overall performance and efficiency of computing systems. As processors operate increasingly faster, the time spent waiting for data from memory becomes a bottleneck [113], leading to sub-optimal processing power utilization. At the heart of this disparity lie various challenges and bottlenecks, including memory latency, bandwidth limitations, and the increasing complexity of computational tasks. These issues hinder the seamless synergy between processors and memory and impede the realization of the full potential of advanced computing technologies.

In the context of this thesis, understanding and addressing the *Processor-Memory Gap* is of paramount importance. As we delve into the *Interaction between Artificial Intelligence and Computer Architecture*, the efficient utilization of computational resources is central to overcoming the challenges posed by this gap. By bridging this divide using AI-inspired techniques and designs, we aim to unlock new frontiers in computer architecture and con-

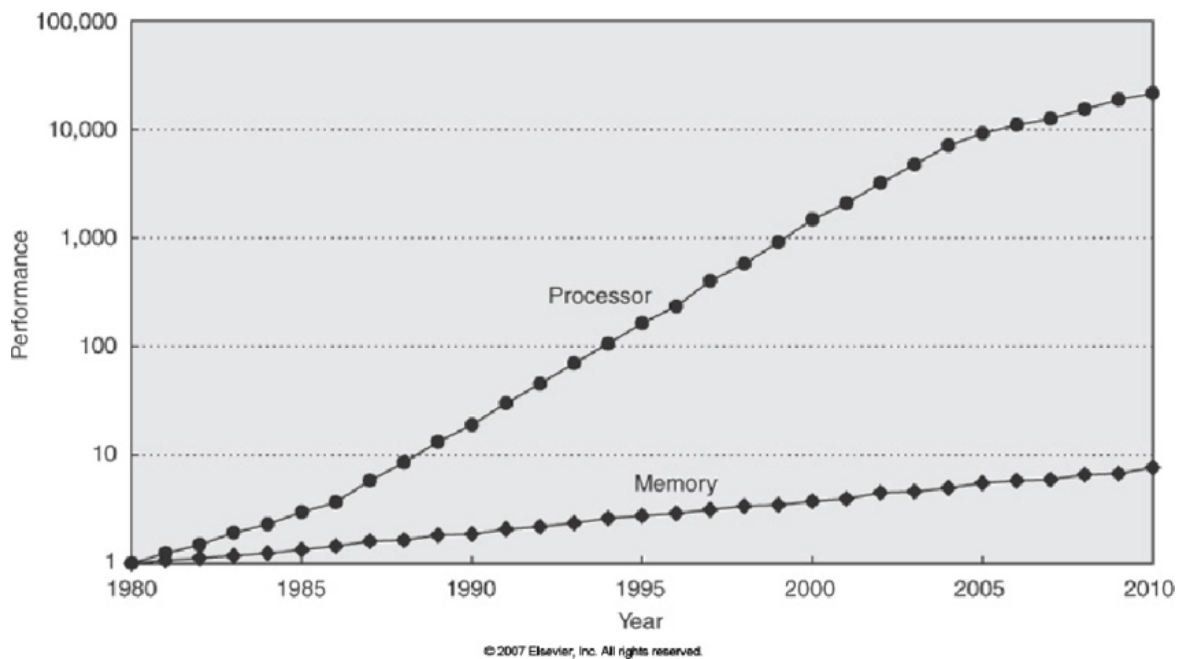


Figure 1.1 Disparity between processor and main memory speeds.

tribute to the broader discourse on optimizing the interplay between processors and memory. Therefore, this study seeks to explore the nuances of the *Processor-Memory Gap* and propose innovative solutions and strategies to mitigate its impact. By doing so, we aspire to advance the field of computer architecture, particularly the memory sub-systems of modern computing systems, and pave the way for more efficient and powerful computing systems.

1.2 Exploring Novel Avenues: AI-Inspired Designs in Computer Architecture

In recent years, *Artificial Intelligence* concepts have started to infuse the realm of computer architecture. As mentioned above, Jiménez and Lin, with their paper "*Dynamic Branch Prediction with Perceptrons*" [50] proposed one of the first designs leveraging the simplest case of neural networks – a single perceptron – and applying it to the branch prediction problem¹. In this section, to help the reader better understand the concept of Artificial Intelligence applied to the field of Computer Architecture, we will first cover some basic history and concepts of *Artificial Intelligence* (*cf.*, Section 1.2.1) and second, we will explore how Artificial Intelligence can help enhance modern micro-architectures (*cf.*, Section 1.2.2).

¹The branch prediction problem in computer architecture entails predicting the outcome of conditional branches to optimize instruction pipelines, minimizing performance penalties associated with mispredictions.

1.2.1 Artificial Neural Networks: Emulating Human Intelligence



Figure 1.2 Winged ΤΑΛΩΝ (Talos) armed with a stone.

In Western culture, *Artificial Intelligence* can be traced back to Greek mythology. ΤΑΛΩΝ (Talos) is a guardian constructed of bronze who acted as guardian for the island of Crete. He was forged by Ἡφαιστος (Hephaestus) – *i.e.*, the Greek god of blacksmiths, metallurgy, and sculpture – and a cyclop. The automaton was then offered as a gift to Minos, King of Crete. The myth of *Talos* is one of the first representations of an artificial being. Figure 1.2 depicts the automaton Talos on an ancient silver didrachma.

The myth of *Talos* shows us that since time immemorial, man have dreamt of craft man-like intelligence to help them in tedious tasks (*i.e.*, patrolling the island of Crete in the myth of *Talos* and eliminating threats). In the context of this myth, deciding whether something or someone is presents a threat is a complex decision undertook by *Talos*. Such a task, requires carefully examining the nature of the object or the person encountered, as incorrect prediction could lead to the destruction or death or an innocent. The idea of *Artificial Intelligence* has traveled through the ages, from Ancient Greece until modern times. Throughout the years, the concept of *Artificial Intelligence* has evolved and went from creating artificial beings (*i.e.*, Golems in medieval times, etc.) to engineering systems able to learn information and provide predictions about complex matters such as financial markets, consumption habits, etc.

As science often gets inspiration from nature, contemporary scientists have attempted to mimic the functioning of the human brain. Continuing upon this trend, at the end of the 1940s, McCulloch and Pitts, in their article "*A logical calculus of the ideas immanent in nervous activity*" [64] formalized methods to emulate the behaviour of the human neurons through mathematical tools. Among these tools was presented the first theoretical rendition of a perceptron.

$$y = \theta(w \cdot x^T + b) = \theta\left(\sum_{1 \leq i \leq n} w_i x_i + b\right) \quad (1.1)$$

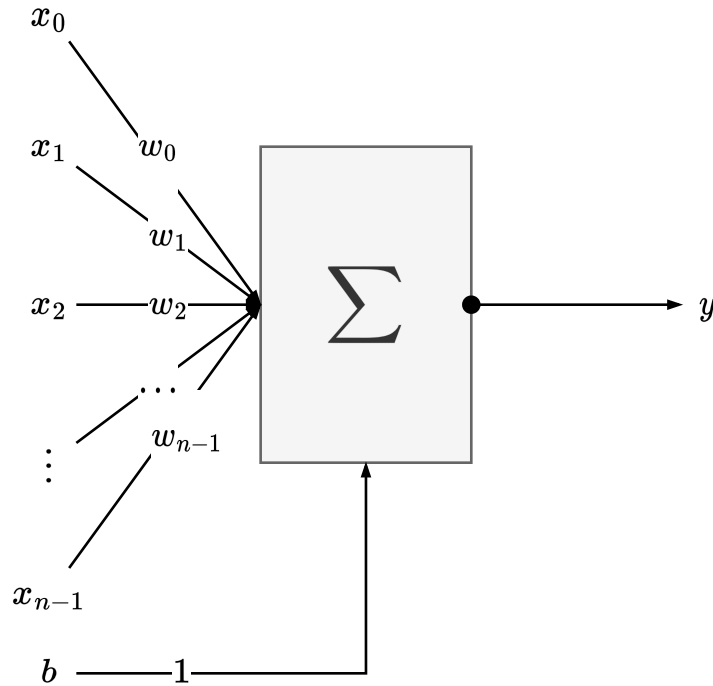


Figure 1.3 Diagram representing a perception.

In 1957, Frank Rosenblatt produced the first hardware implementation of a perceptron at the Cornell Aeronautical Laboratory, and his subsequent article "*The perceptron: a probabilistic model for information storage and organization in the brain*" [83] discussing how the perception was built. A perceptron is essentially a binary classifier that takes a vector x as an input and outputs a binary prediction y . One of the features of a perceptron is that it weighs inputs differently thanks to a weight vector w . Equation 1.1 shows how y is formally computed using x , w , b a bias value, and θ the Heaviside step-function function.

Now that the concept of a perceptron has been formally defined, one has to acknowledge that intelligence is nothing without means of learning. In the context of a single perceptron, the learning rule simply consists of updating weights by adding to the current weight value $w_i(t)$ a value proportional to the distance between the expected output d_j and the actual output $y_j(t)$. Equation 1.2 formally shows how a perceptron's weight is updated, where r is the learning rate of the perceptron. Figure 1.3 depicts Equation 1.1 through a flowchart diagram showing the input, output, and parameters of a *perception*, a meta-parameter² of the model.

²A meta-parameter, or hyperparameter, guides a machine learning model's behavior, set externally before training, impacting performance.

$$w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t)) \cdot x_i(t) \quad (1.2)$$

However very simple, this paradigm of *Artificial Intelligence* is at the base of some the latest innovations in the field of *Computer Architecture*. As a matter of fact, it allows inexpensive classification of data input in a n -dimensions space. A non-trivial problem for human minds.

1.2.2 Applying AI Techniques to Enhance Computer Architecture

In the dynamic realm of computer architecture, a transformative wave is reshaping the landscape—a wave propelled by the infusion of artificial intelligence techniques into the fabric of system design. This exploration delves into how AI-inspired designs act as catalysts, ushering in innovative micro-architectural prediction mechanisms and unlocking a realm of possibilities for performance optimization. The integration of AI techniques into micro-architecture marks a pivotal moment in the evolution of computing systems, aiming to advance the application of AI-inspired designs for enhancing micro-architectural prediction mechanisms.

This study investigates the intricate interplay between AI-inspired designs and micro-architectural prediction mechanisms. Leveraging AI-inspired designs, our research crafts predictive models that dynamically adapt to the intricacies of memory access patterns, addressing challenges posed by the *Processor-Memory Gap* and opening new frontiers for predicting and optimizing memory sub-system behavior. The fusion of *Artificial Intelligence* and *Computer Architecture* gives rise to adaptive architectures capable of learning and adjusting to the unique demands of diverse workloads. We delve into the design principles underpinning these adaptive architectures, exploring how AI-inspired systems can further optimize overall system performance.

To ground these discussions, practical examples of AI-inspired micro-architectural prediction mechanisms will be explored in the following chapter (*cf.*, Chapter 2). This includes case studies and noteworthy research proposals extracted from the literature, highlighting the tangible benefits and potential challenges of integrating AI-inspired designs into the fabric of modern computing systems, specifically within the memory sub-system.

1.3 Motivation

In the ever-evolving landscape of *Computer Architecture*, the *Processor-Memory Gap* is a persistent challenge that hampers the full realization of computational potential. As

processors continue to advance in speed and complexity, the disparity in performance between processors and memory systems becomes increasingly pronounced. This disjunction poses a critical bottleneck, hindering the seamless synergy between processing units and memory access and resulting in sub-optimal utilization of computational power. The memory sub-system, an essential component of computer architecture, plays a pivotal role in bridging or exacerbating the *Processor-Memory Gap*. Challenges like memory latency, bandwidth limitations, and the escalating demands of diverse workloads necessitate a reevaluation of the memory hierarchy of modern micro-architectures.

The motivation for this study stems from the imperative need to address and transcend the limitations imposed by the *Processor-Memory Gap* within the memory sub-system. Beyond a theoretical concern, this gap has practical implications for various applications, from data-intensive scientific simulations to real-time artificial intelligence processing. Emerging workloads, such as graph-processing, impose an exceptionally high stress on the cache hierarchy and the memory sub-system. This heightened demand exacerbates the challenges associated with the *Processor-Memory Gap*.

Computer architects have traditionally fine-tuned their designs for classical workloads such as SPEC CPU 2006 and 2017 (*cf.*, Chapter 3). However, these designs prove inadequate when applied to emergent workloads. The mismatch between traditional designs and the demands of evolving applications further accentuates the challenges the *Processor-Memory Gap* poses. Recognizing the limitations of current designs, this study proposes to employ AI-inspired designs. These designs bring forth new classes of techniques that have the potential to significantly enhance the performance of emerging workloads. By leveraging artificial intelligence in micro-architectural prediction mechanisms, we aim to bridge the gap between classical designs and the evolving demands of modern computing.

Furthermore, integrating artificial intelligence techniques into micro-architectural prediction mechanisms presents an exciting avenue to enhance the predictive capabilities of memory systems. Leveraging AI-inspired designs for micro-architectural predictions promises to mitigate the impact of the *Processor-Memory Gap* and optimize memory access patterns dynamically. This study aims to contribute to the field of *Computer Architecture* by providing innovative solutions to these challenges. By exploring the intricate relationship between processors, memory, and AI-inspired designs, we seek to uncover new opportunities for enhancing computing systems' performance, efficiency, and adaptability.

In the following chapters, we will delve into our research's investigations and practical implications to shed light on the transformative potential of addressing the *Processor-Memory Gap* leveraging AI-inspired designs.

1.4 Thesis Approach

In the dynamic landscape of computer architecture, the persistent challenge of the *Processor-Memory Gap* continues to impede optimal system performance. Despite remarkable technological advancements, the introduction of novel and demanding workloads, such as graph-processing, has heightened the strain on the memory sub-system, particularly within the cache hierarchy. This decline in performance emphasizes the urgent need for innovative solutions to alleviate the escalating pressure on computing systems.

This thesis is dedicated to mitigating the adverse impacts of emerging workloads (*i.e.*, mostly graph-processing workloads for the matter of this thesis) on system performance. Our primary objective is to propose inventive designs leveraging AI-inspired techniques. These designs are intricately crafted to specifically target reducing access time to memory and alleviate the burdensome nature of costly DRAM requests. By integrating AI-inspired designs into micro-architectural prediction mechanisms, we aim to usher in a paradigm shift in how computing systems optimize memory utilization.

These AI-inspired techniques are geared towards optimizing access time and intelligently managing DRAM requests, thereby mitigating the detrimental effects of latency on overall system performance. Beyond addressing memory access challenges, our research delves into the intricate relationship between the cache hierarchy and the *Processor-Memory Gap*, particularly concerning emerging workloads. We shed light on how the meticulously designed and fine-tuned cache hierarchy for classical workloads (*cf.*, Chapter 4 and 5) exacerbates the challenges posed by the widening *Processor-Memory Gap* when confronted with the distinctive demands of emerging applications.

In the subsequent chapters, we will delve deeper into the methodologies employed, present empirical findings, and expound upon the implications of our proposed AI-inspired techniques. Our overarching objective is to contribute to advancing *Computer Architecture* by offering inventive solutions that enhance the performance, efficiency, and adaptability of computing systems grappling with the challenges of the ever-evolving *Processor-Memory Gap* and emerging workloads.

1.5 Thesis Contributions

Motivated by the *Processor-Memory Gap*, exacerbated by the emerging workloads - as seen in Section 1.3 - this thesis makes three contributions. The first contribution studies the Impact of the Last-Level Cache (LLC) Replacement Policies (*cf.*, Chapter 3). This first contribution focuses on emerging workloads such as graph-processing [15] and compares the behavior of

cache replacement policies to more classic workloads such as SPEC CPU 2006 and 2017 [91, 92]. While focusing on graph-processing workloads, the last two contributions (cf.: Chapter 4 and 5) of this thesis propose and evaluate two designs that take advantage of the pressure incurred by these workloads on the cache hierarchy and, in broader terms, on the memory sub-system.

1.5.1 Characterizing the impact of last-level cache replacement policies on big-data workloads

In this first study, we delve into the critical realm of cache management policies, specifically addressing the pronounced latency gap between the Last Level Cache (LLC) and main memory accesses. While prior works in *Computer Architecture* have made significant strides in LLC replacement policies, a notable limitation exists in their evaluation scope, often focusing narrowly on benchmarks like SPEC CPU 2006 and 2017 [91, 92]. These benchmarks, however, represent only a subset of the diverse and complex workloads prevalent in contemporary HPC environments. Our investigation expands beyond traditional evaluation boundaries by incorporating a diverse mix of workloads, including graph-processing (GAP) [15], scientific simulations (XSBench) [98], industrial applications (Qualcomm) [27], and the established SPEC CPU 2006 and 2017 [91, 92]. Notably, we assess the performance of state-of-the-art LLC replacement policies such as Multiperspective Reuse Prediction (MPPPB) [51], Glider [87], Hawkeye [45], SHiP [106], DRRIP, and SRRIP [47, 80] across this broad spectrum of workloads.

The evaluation conducted in this work reveals its main contribution. A critical gap exists in the efficacy of existing state-of-the-art LLC replacement policies when applied to HPC and big data workloads. Despite their success in improving performance for SPEC CPU benchmarks, these policies struggle to capture the intricate access patterns inherent in the highly complex behavior of emerging workloads. To address the findings of this work, we design and propose two novel LLC replacement policies derived from MPPPB to address this gap. The first, Multi-Sampler Multiperspective (MS-MPPPB), deploys multiple samplers dynamically, selecting the best-behaving sampler to enhance reuse distance predictions. The second, Multiperspective with Dynamic Features Selector (DS-MPPPB), optimizes prediction accuracy by dynamically selecting the best-performing features from a set of 64 distinct features.

Our experimental results demonstrate the superior performance of these newly introduced policies. MS-MPPPB achieves a geometric mean speed-up of 8.3 % over the traditional Least Recently Used (LRU) policy, while DS-MPPPB outperforms LRU with a geometric

mean speed-up of 8.0 %. Particularly noteworthy is their effectiveness in addressing the demands of big data and HPC workloads, surpassing the performance benefits of established approaches like MPPPB, Glider, and Hawkeye, which yield geometric mean speed-ups of 7.0 %, 5.0 %, and 4.8 % over LRU, respectively.

This first contribution provides a nuanced understanding of LLC replacement policies, highlights their limitations in diverse workloads, and introduces innovative policies that significantly enhance performance across a broad spectrum of contemporary computing demands.

1.5.2 Practically Tackling Memory Bottlenecks of Graph-Processing Workloads

In the second part of this thesis, our objective is to make a substantial contribution to the field of computer architecture. Specifically, we aim to address the shortcomings incurred by graph-processing workloads. This endeavor involves a focused effort to enhance our understanding of the intricacies inherent in handling graph-processing workloads within the realm of computer architecture.

This work underscores the inefficiency of traditional cache hierarchies for graph-processing workloads. Notably, a substantial proportion (78.6 %) of L1 Data Cache (L1D) misses leads to corresponding misses in the L2 Cache (L2C) and Last Level Cache (LLC), necessitating resource-intensive DRAM accesses.

This work introduces a fundamental observation, demonstrating the feasibility of predicting whether a memory access in graph-processing workloads will be served from DRAM. This prediction relies on analyzing strides between accesses triggered by instructions with the same Program Counter (PC), forming the basis for the proposed design: the Large Predictor (LP). This work introduces LP, a cost-effective micro-architectural predictor to discriminate regular memory accesses from irregular memory accesses in graph-processing workloads. Thus, LP enables the identification of irregular accesses, strategically bypassing them through the cache hierarchy.

To tackle the findings of this work, we present the Side Data Cache (SDC), a novel hardware cache placed alongside the L1D in the cache hierarchy. Specifically engineered for highly irregular memory accesses, the SDC significantly reduces latency costs associated with such accesses without compromising the integrity of the overall cache hierarchy. Any requests missing in the SDC trigger an access to the DRAM without going through the rest of the cache hierarchy, thus avoiding the overhead of the L2C and LLC.

Through a thorough experimental campaign, we showcase the effectiveness of the combination of LP and SDC, yielding notable geometric mean speed-ups of 20.3 % and 20.2 % in single- and multi-core scenarios, respectively. These advancements outpace conventional optimization techniques to cache hierarchies across contemporary graph-processing workloads. LP and SDC surpass the Transpose-based Cache Replacement (T-OPT) [10], one of the prevailing cache replacement policy for graph-processing applications. The proposed approach exhibits superior speed-ups of 10.9 % and 13.8 % in single-core and multi-core contexts, respectively.

LP and SDC are implemented with a modest hardware budget, demanding 10KB of storage per core. This highlights the practicality and efficiency of the proposed solution within real-world hardware constraints.

In conclusion, this chapter presents a comprehensive approach that not only identifies the limitations of conventional cache hierarchies in the context of graph-processing workloads but also introduces innovative solutions, LP and SDC, substantially enhancing overall performance and efficiency across diverse computing scenarios.

1.5.3 A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering

The third and last work of this thesis introduces the *Two Level Perceptron* (TLP) predictor as a novel solution to mitigate the performance and DRAM bandwidth constraints of contemporary applications with large data footprints.

Through this work, we make two main contributions: i) First, we observe that Off-Chip Prediction mechanisms such as Hermes or the LocMap+PLD designs [19, 48] can be repurposed and used for L1D prefetch filtering, ii) we observe that Hermes effectively improves the performance of a large set of workloads ranging from SPEC CPU 2006 and 2017 [91, 92] and graph-processing workloads [15]. However, this performance improvement comes at the cost of increased DRAM pressure. DRAM bandwidth being a scarce resource, this must be addressed as it directly correlates with the *Processor-Memory Gap* problem.

To tackle the different findings of this work, we introduce the TLP framework. This framework features a two-level perceptron architecture composed of the *First Level Predictor* (FLP) and the *Second Level Predictor* (SLP). The FLP excels in accurate off-chip prediction by leveraging program features derived from virtual addresses and program counters. As compared to Hermes, It introduces a novel selective delay component to enhance off-chip prediction accuracy. The SLP strategically uses off-chip prediction to drive L1D prefetch filtering. It utilizes physical addresses, program counters, and FLP predictions as features,

making it the first hardware proposal targeting off-chip prediction and prefetch filtering using a multi-level perceptron approach. Crucially, TLP achieves remarkable hardware efficiency, requiring only 7KB of storage.

Comprehensive experiments compare TLP's performance with state-of-the-art approaches on a range of single-core and multi-core workloads. TLP significantly reduces average DRAM transactions by 30.7 % and 17.7 % across single-core and multi-core workloads, respectively, as compared to a baseline with state-of-the-art cache prefetchers but no off-chip prediction mechanism. TLP achieves geometric mean performance speed-ups of 6.2 % and 11.8 % across single-core and multi-core workloads, respectively, showcasing its efficacy in enhancing overall system performance.

Finally, the evaluation demonstrates the effectiveness of TLP independently of the L1D prefetching logic, highlighting its versatility and applicability in various computing scenarios.

CHAPTER

2

BACKGROUND

La guerre, c'est comme une partie de
cartes : l'élément le plus important,
c'est d'avoir de bonnes cartes.

Maréchal Ferdinand Foch

THIS chapter serves as a comprehensive exploration of the diverse array of subjects encapsulated within this thesis. Initially, in Section 2.1, we delve deep into the intricacies of the Processor-Memory gap, dissecting its implications and consequences. Subsequently, in Section 2.2, we meticulously introduce the core concept of cache replacement policies, elucidating their fundamental principles, distinctive properties, and associated challenges. We delve into a comprehensive analysis of contemporary techniques employed in this domain. Moving forward, in Section 2.3, we focus on hardware prefetching, a complementary strategy to cache replacement policies. In this section, and similarly to the previous section, we introduce the concept of hardware prefetching, the fundamental principles of this class of designs, along with the associated challenges. Within this section, we unravel the intricacies of hardware prefetching techniques. Lastly, in Section 2.4, we venture into a more recent and innovative approach to cache management, off-chip prediction, providing an in-depth exploration of its concepts and applications in the evolving landscape of memory access optimization.

2.1 The Processor-Memory Gap

The Processor-Memory Gap, or the *Memory Wall* as the term was first coined by Wulf and McKee in 1995 in their article called "*Hitting the Memory Wall: Implications of the Obvious*" [107], represents a fundamental challenge at the heart of computer architecture. It underscores the glaring disparity between the rapid advancements of CPU processing speeds and the comparatively sluggish improvement in memory access latencies. As a matter of fact, over the past decades, through innovations in the domain of computer architecture - from refinements of micro-architecture to groundbreaking fabrication processes - processors' performance has dramatically increased. Figure 2.1 gives a depiction of the Processor-Memory gap by showing the disparity between main memory access latencies and processor clock cycles.

In this article, Wulf and McKee depict a very grim outlook on the future of micro-architecture. Through the few pages of this article, they provide us with two key learnings that have since guided research efforts in the domain:

1. As per the trends of both processor and memory speeds available, at the time, to the authors, "*Hitting the Wall*" seemed inevitable as processor and memory speeds kept on branching away.
2. The authors, in addition to their predictions, provided the readers with potential research avenues to explore in an effort to push that performance wall as far back

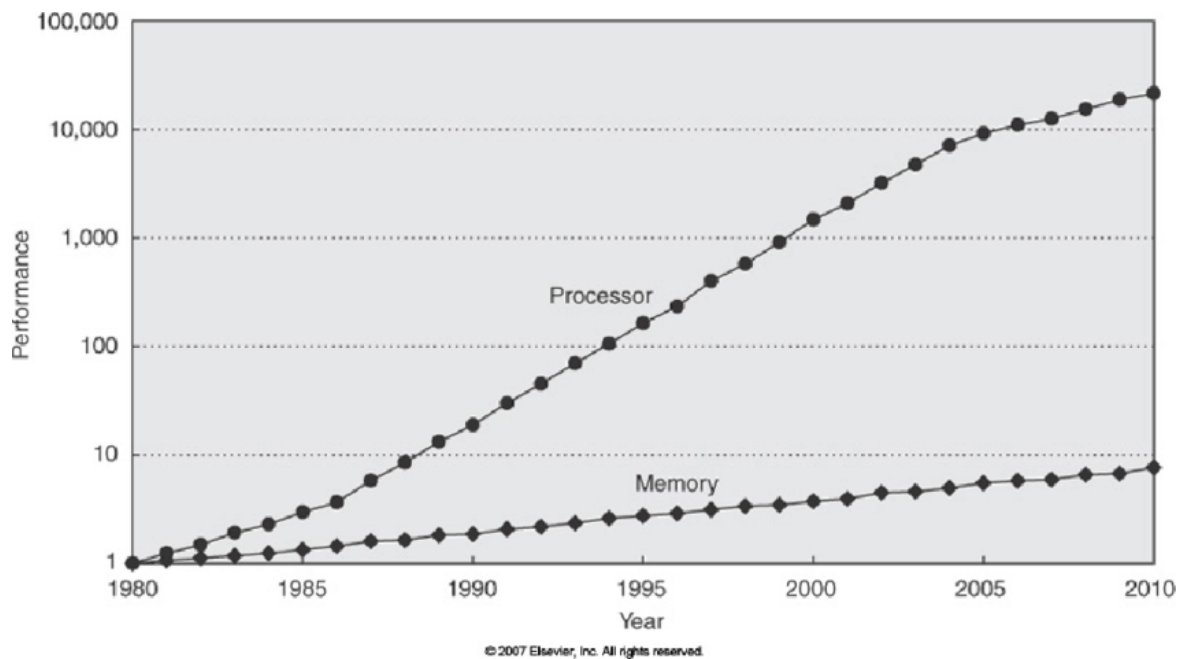


Figure 2.1 Disparity between processor and main memory speeds.

as possible. Such avenues consisting in either, lowering the need to access the main memory, and lowering the access time to the said memory.

One aspect of micro-architecture that significantly evolved over the past decades is the cache hierarchy. Hierarchies of caches are historically the first memory latency-tolerance technique developed and used in vendors' products. Smaller caches are usually placed closer to the core as the capacity of a cache is directly proportional to its access time. The placement, design, size, and property decisions regarding the cache hierarchy of a given micro-architecture belong to the team of computer architects in charge of the development of a given micro-architecture.

Building on Wulf and McKee premises, researchers have studied various approaches to bridge the Processor-Memory gap. Through advanced cache replacement policies, hardware prefetching mechanisms, computer architects have pushed the performance wall further away. Thus proving Wulf and McKee wrong for workloads fitting the premises of caches: i) a memory footprint fitting the capacity of the caches; ii) there is no one single optimal strategy as each workload has its own characteristics when it comes to access patterns. Regarding most recent workloads demonstrating behavior opposite to the premises of caches, recent work from researchers on off-chip prediction allows to push this wall away.

Nowadays, vendors have pushed the concept of multi-level cache hierarchy as far as employing three distinct cache levels. The first and second level are called the L1 and L2

caches. The L1 cache is further divided into the L1 Data cache (L1D), and the L1 Instruction cache (L1I). As their names suggest, these L1 caches are dedicated to storing data, and instructions, respectively, therefore making modern processors' micro-architecture practical implementations of Harvard Architecture [82, 78]. In addition, the first two levels of the cache hierarchy are usually small and private. That is to say, these caches can only be accessed by the cores they are attached to. The third level of the cache hierarchy, commonly called Last Level Cache (LLC) consist in a larger cache, shared among the core of the system. Figure 2.2 shows the anatomy of a modern cache hierarchy, including latency and capacity specifications for a Cascade Lake Intel chip [26].

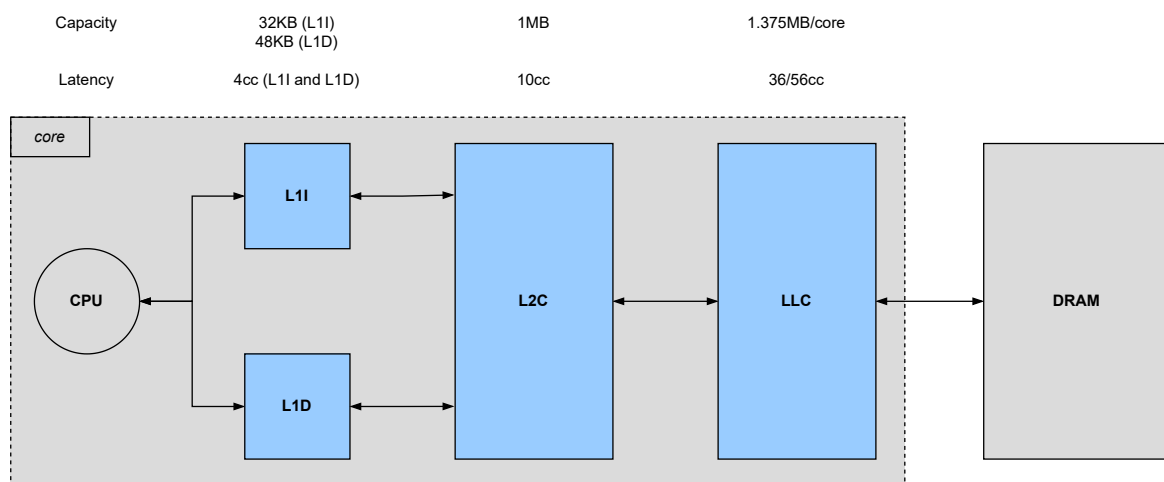


Figure 2.2 Anatomy of a modern cache hierarchy composed of three cache levels. The latency and capacity parameters are extracted from a recent Cascade Lake chip [26].

As mentioned above, through the course of the past decades, researchers have developed a variety of ideas and designs aiming at pushing back the performance wall, as multi-level cache hierarchies are merely the first step towards breaking it down. In the rest of this chapter, we will cover cache replacement policies, hardware prefetchers, and off-chip predictions as they constitute another step towards bridging the Processor-Memory Gap.

2.2 Cache Replacement Policies

As we discussed in the previous section, caches are employed in modern micro-architectures as a way to give the user the impression that they are accessing the main memory with low latency hiding the latency of the main memory. Thus, caches influence the performance of workload. However, as the vast majority of modern workloads have a memory footprint that far exceeds the capacity of the biggest caches on the market (*e.g.*: graph-processing

workloads typically process input graphs spanning gigabytes of data), it is necessary to decide what data should be kept in the caches, and how long it must remain there. Cache replacement policies are the components responsible for these decisions.

Figure 2.3 depicts the operation of a cache and its attached replacement policy. Upon an access to the cache ❶, there are two possible outcomes: i) the requested block is found in the cache memory. This event is called a *hit*; ii) on the contrary, the requested block is not found in the cache memory. This event is called a *miss*. When a hit occurs in the cache, the replacement policy is consulted to update its internal state taking as input the accessed memory block. Conversely, upon the occurrence of a miss, a longer, more complex sequence of events takes place. First, the replacement policy is consulted to select victim. This victim block will be evicted ❷ from the cache to make space for the requested block. If the victim block was tagged as *dirty*¹, it is written back to a lower level of the cache hierarchy. Finally, the requested block is fetched from a lower level of the cache hierarchy or from the main memory ❸.

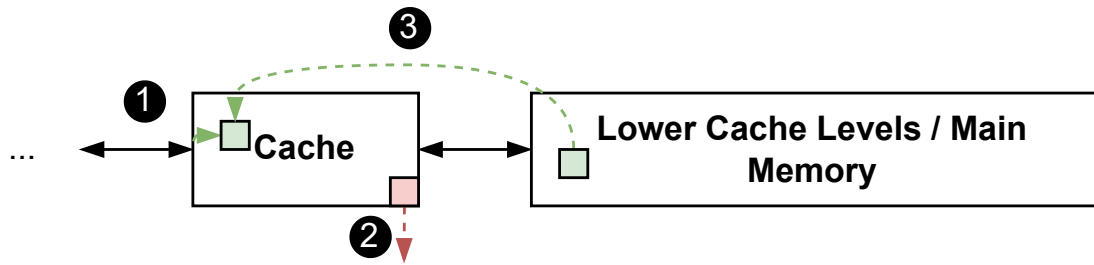


Figure 2.3 Operation of a cache and its replacement policy.

2.2.1 The Optimal Replacement Strategy: Belady's MIN Algorithm

The core idea of a cache replacement policy is to approximate as closely as possible the optimal replacement strategy. This strategy, first mentioned in 1966 by Belady in *"A study of replacement algorithms for a virtual-storage computer"*, and later coined as Belady's MIN [18] algorithm, is considered the theoretical optimal cache replacement strategy. This strategy can be summarized in a very simple manner: *"Upon an eviction, the elected victim is the block that will experience a reuse the furthest in the future"*.

As one can see, this algorithm requires knowledge of the future accesses to memory. Therefore, implementing this algorithm is, in practice, impossible. However, even though

¹The dirty bit in a cache line is a flag indicating whether the data has been modified since loading. It facilitates efficient memory management by selectively writing back only the modified data during cache eviction, minimizing unnecessary memory transfers and improving overall system performance.

this algorithm is not practical for implementation, one can extract some knowledge from it by running an example. Figure 2.4 shows an example of a sequence of memory accesses to a cache. This example shows a cache composed of only two entries receiving a memory access request composed of blocks **A**, **B**, **C**, and **D**. This figure shows how the content of the cache entries would change over the sequence of accesses **A**, **B**, **C**, **D**, **A** applying Belady's MIN algorithm. On the left-hand side of the figure, the reader can find the sequence of memory accesses where blocks denoted in bold have been accessed; others are yet to be accessed. The rationale behind Belady's MIN is apparent on epoch 4 when an access to block **C** is triggered. On that epoch, the cache contains blocks **A** and **B**. As **B** is the block that will be accessed the furthest in the future (*i.e.*, **B** does not belong to the rest of the access sequence, its reuse distance can be thought to be infinite), **B** is thus evicted from the cache and replaced by **C**. Similarly, upon epoch 5, block **D** is accessed, and as block **C** now has an infinite reuse distance, it is thus evicted from the cache living space to block **D** to be inserted.

Access sequence

A, B, C, D, A	...	...
A , B, C, D, A	A	...
A , B , C, D, A	A	B
A , B , C , D, A	A	C
A , B , C , D , A	A	D
A , B , C , D , A	A	D

Figure 2.4 Example of Belady's MIN algorithm applied to an example sequence of memory accesses. Green boxes indicate the newly inserted blocks.

Over the years, researchers have shown their craftsmanship at building more and more advanced cache replacement policies, trying to approximate as closely as possible the decisions of Belady's MIN algorithm. In Sections 2.2.2, 2.2.3, 2.2.4, and 2.2.5 we delve into some of the more relevant state-of-the-art replacement policies.

2.2.2 Reuse Distance Prediction

Through the knowledge learned in the example provided by Figure 2.4, in the previous section (*cf.*, Section 2.2.1), we have learnt that the concept of reuse distance acts as a proxy for the replacement decision. To be specific, the concept of reuse distance refers to the distance² between two consecutive accesses to the same memory block. Thus, the line *"Upon an eviction, the elected victim is the block that will experience a reuse the furthest in the future"* can be rephrased as *"Upon an eviction, the elected victim is the block with longest reuse distance"*. Therefore, this concept is cornerstone in computer architecture and more specifically when it comes to cache replacement policies, as understanding the relative order of the reuse distances between the elements stored in a cache gives away the replacement decision. Going on the previous section's example (*cf.*, Section 2.2.1), block A is an example of a block that is accessed several times in the access sequence. As three other accesses separate the two consecutive accesses to block A, therefore three is the reuse distance between these two consecutive accesses.

In order to approximate reuse distances at run-time, and to make the best possible replacement decisions, researchers have proposed a practical yet simple method to approximate these reuse-distances. This method of approximation is performed through Re-reference Interval Prediction (RRIP). This class of replacement policies initially proposed by Jaleel et al. [47, 80] consist of three basic policies: i) SRRIP; ii) BRRIP; and iii) DRRIP.

The main idea behind RRIP is the classification of blocks into re-reference classes. SRRIP is a, scan-resistant, is limited to always inserting new coming blocks in a fixed class. In contrast, BRRIP provides more flexibility by frequently inserting blocks in the distant re-reference class and infrequently inserting blocks in the long re-reference class. Finally, DRRIP leverages Set-Dueling to determine which of SRRIP and BRRIP is best suited for a given workload or program phase, making it both scan and thrash resistant.

This class of cache replacement policies relies on Re-reference Prediction Value (RRPV), which are counters of arbitrary size dictated by the architects that enable the classification of cache line. A common way RRPV are used in micro-architectural proposals is through n -bits saturating counters, where $n > 0$. The values of these counters, thus ranging between 0 and $2^n - 1$ where 0 encodes a *near-immediate reuse*, and $2^n - 1$ a *distant reuse*. All the values in between provide a gradient for fine-grained predictions.

²The reuse distance are measured in terms of memory accesses.

2.2.3 Signature-based Hit Predictor

Expanding on the foundational research work by Jaleel et al. [47, 80], which introduced the reuse distance prediction framework elaborated in Section 2.2.2, and building upon the prior contributions of Khan et al. [58] in the development of a program-counter-based dead block prediction mechanism, Wu *et al.* [105] have advanced the field with their innovative proposal for a last-level cache (LLC) replacement policy design that capitalizes on the program counter as a predictive feature.

Fundamentally, the *Signature-based Hit Predictor (SHiP)* [105] last-level cache (LLC) replacement policy uses the SRRIP replacement policy (*cf.*, Section 2.2.2) as its foundational building block and modifies it to integrate some of the learnings provided by other research works. This prediction mechanism relies on the following premise: a significant proportion of the LLC accesses are in fact accesses to blocks that will never experience reuse during their lifetime in the cache. Such blocks are called *dead blocks*, or are said to be *dead-on-arrival*. Dead blocks are a major problem for caches as they occupy cache lines for a given period of time, while experiencing no reuse. Thus they create cache pollution, a phenomenon that can be described as a cache system allocating resources to a memory block that could better be used for another, more valuable block. Dead blocks, because they consume valuable and scarce caching resources cause higher miss rates and longer memory access latency. In their work "*Sampling Dead Block Prediction for Last-Level Caches*" [59], Khan *et al.* show that building a program-counter based prediction mechanism for dead block prediction can leverage significant performance improvement by not inserting blocks that are predicted to be *dead-on-arrival*. The SHiP predictor compounds all these different insights and designs and modifies SRRIP to favour the eviction of *dead blocks*. Through a prediction table that dynamically learns whether a given PC signature is likely to provides cache hits, SHiP learns whether a given PC tends to produce accesses to dead blocks, or accesses to blocks that experience reuse in the cache. To do so, the predictor leverages a simple learning logic: upon a hit on a given signature, the associated entry in the prediction table is incremented, conversely, upon a miss, that entry is decremented. If the PC-based prediction table predicts that a given PC is likely to produce dead blocks, any cache block related to that PC will be inserted in the distant re-reference interval.

As we explored in details the SHiP design and its many benefits, we believe that it is necessary to put into perspective this design and explore some of its shortcomings. As mentioned above, SHiP solely relies on program counters to build its prediction table. For conventional workloads such SPEC 2006 and SPEC 2017 [91, 92], this is enough to build a robust predictor as PCs tend to exhibit distinct behaviors. Thus allowing PC-based predictors to easily distinguish the behavior of the PCs. Conversely, while considering modern

workloads such as graph-processing, this assertion turns untrue. As we will see in Chapter 3, such workloads only have a handful of PCs whose behavior can vary dramatically over time, thus making classical PC-based prediction canvas less reliable for accurate predictions.

2.2.4 Multiperspective Reuse Prediction

Building upon the ideas and learnings introduced in the preceding sections, specifically referring to Sections 2.2.2 and 2.2.3, Jiménez and Teran have introduced an innovative cache replacement policy known as the *Multiperspective Reuse Prediction* [55] (MPPPB). This novel approach harnesses the principles of feature analysis and dead block prediction to achieve substantial performance enhancements when compared to the conventional Least Recently Used (LRU) replacement policy. The foundation of this replacement policy is an extension of the concept of *features*, as initially formulated in earlier research endeavors [58, 105, 96]. These features serve as the fundamental building blocks for the primary component of the replacement policy, a perceptron predictor. In conjunction with the predictor, a sampler³ plays a crucial role in supplying the required learning signals to the perceptron predictor, leveraging reuse and eviction observations. The perceptron, utilizing these learning inputs triggered by the sampler, continuously updates its prediction tables, resulting in a more accurate and effective cache replacement strategy.

MPPPB relies on the idea of correlating reuse prediction of a cache block with a large number of features⁴. These features range from bits extracted from parts of the history of demand load program counters to characterizing access patterns in terms of bursts (*i.e.*, a cache block that is accessed more than once consecutively). Specifically, in the context of a perceptron predictor for cache replacement, a feature can be defined as a hash function that takes as an input any micro-architectural metadata available and associated to the cache block upon which the prediction is meant to be issued. Additionally, a feature might perform additional operations in the very specific context of the MPPPB policy, such as *XOR*-ing the hash result with the most recent load demand program counter. Finally, these features are associated with an associativity value which serves to emulate a sampler with shorter associativity when it comes to that particular feature (*i.e.*, if a feature has an associativity of 10, any sampler block seeing its LRU stack position going past that value is considered evicted for that feature). Such metadata can be elements of the history of demand load program counters, the virtual/physical address of the cache block accessed, etc. In a perceptron

³In computer architecture, a sampler observes access patterns by strategically sampling sets, noting accesses, insertions, and evictions. This tool aids in optimizing replacement policies for improved system performance.

⁴In their paper Jiménez and Teran leverage a total of sixteen features. All features are determined through a genetic algorithm [55].

predictor, each feature has a prediction table attached to it. Upon a prediction request, the perceptron uses micro-architectural metadata as inputs to its various features and produces indices to index its prediction tables. By indexing these tables, the weights are obtained and summed in a single prediction value and compared to a set of thresholds to drive actions such as bypass, promotion, and placement.

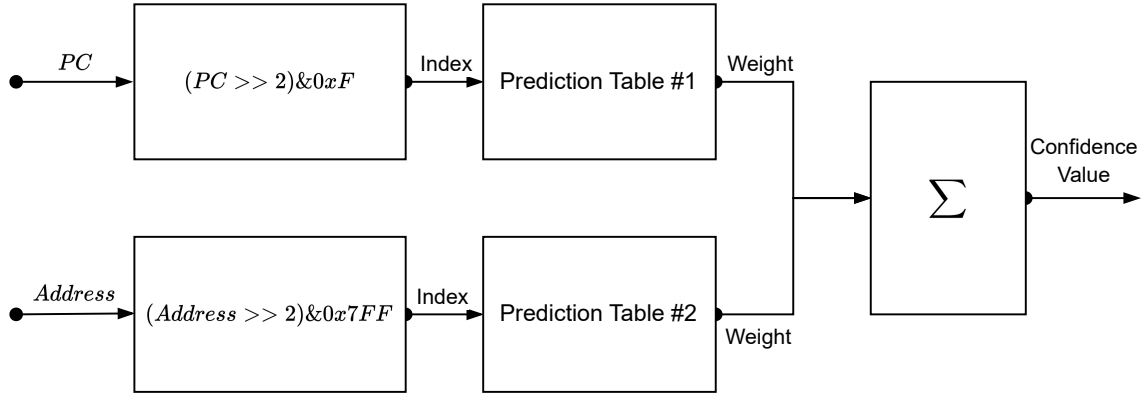


Figure 2.5 Example of a simplified Multiperspective design leveraging two features based on the PC and the physical address.

Figure 2.5 provides an example of a simplified Multiperspective predictor leveraging only two features. This toy example allows to understand how, starting with basic micro-architectural metadata (*i.e.*, the PCs that triggered an access to an LLC cache line or the physical address referenced by such an access), the prediction tables are indexed and the final confidence value is constructed. In this example, the predictor leverages two features: i) the first, based on the PC, ii) the second based on the physical address. The hash function associated to the PC gets rid of the first 2 LSB and only retains bits 3 to 6. The second hash function, associated to the physical address gets rid of the first 2 LSB and retains bits 3 to 12.

The process of finding an adequate set of feature is not straight forward as it involves optimization algorithms over a research space made of a very high number of dimensions. Equation 2.1 provides a generalized mathematical modeling of a feature f as a hash function. In this function T represent the type of feature used, A denotes the associativity attached to the features that is used by the sample to trigger learning signals, B denotes the first bit to be considered in the input value provided to the feature, E denotes the last bit to be considered in the input value, W denotes the W^{th} most recent memory access instruction, finally X is true if the feature is XOR-ed with PC of the current memory access instruction. In the MPPPB framework, a total of seven feature types are available: i) the *program counter* (PC), ii) the *address* strip of its block offset value, iii) a *bias* which can behave either as a global counter or as a simple PC feature depending on the X parameter, iv) *burst* which tracks consecutive

reuses of blocks, v) *insertion* which tracks the accesses that lead to cache insertions, vi) *lastmiss* which tells if the last access to a cache set was an insertion, and vii) the *offset* which consists in the offset bits of the address accessed.

$$f = \text{hash}(T, A, B, E, W, X) \quad (2.1)$$

For instance, the example feature provided in Equation 2.2, and rewritten using our generalized notation system can be interpreted as a feature using bits 6 to 20 of the current memory access program counter. The associativity used by the sampler to trigger learning signals on that feature is 17. Finally, bits 6 to 20 of the current memory access program counter are XOR-ed with the entire current memory access program counter.

$$f = pc(17, 6, 20, 0, 1) = \text{hash}(pc, 17, 6, 20, 0, \text{True}) \quad (2.2)$$

As mentioned above, the sampler, through reuse or eviction events, provides learning signals that are passed to the perceptron predictor. Through the perceptron learning algorithm described in previous work [58, 51], MPPPB updates the weights of the different prediction tables according to the learning rules. Specifically, a reuse of sampler block (*i.e.*: a hit in the sampler) while its position in the LRU stack is lower than the associativity of a given feature results in a positive training for the weight in the prediction table associated with that feature. Conversely, if a sampler block gets demoted beyond the associativity of a feature, the weight in the prediction table associated with that feature is trained negatively.

With this work, Jiménez and Teran demonstrated the usefulness and impact of combining multiple features. Among the many features employed, the history of load demand program counters is revealed to be one of the most important. However, MPPPB is positioned in the LLC, the third level of the cache hierarchy in their simulated system. This positioning leads to a very high degree of filtering in the program counters by the other levels of the cache hierarchy, thus decreasing the accuracy of such a predictor. Therefore, this work shows that introducing additional features, such as bits extracted from the memory addresses, helps mitigate the inaccuracy introduced by a filtered sequence of program counters. The main takeaway of this work is the idea of combining several features, leveraging very different micro-architectural metadata to yield higher prediction accuracy. As we will see in next works (*cf.*, Section 2.4.2), other more recent works have leveraged this idea of a perception predictor leveraging several features to provide high prediction accuracy.

Similarly to the SHiP replacement policy design that we covered previously (*cf.*, Section 2.2.3) that we covered previously, we put into perspective the MPPPB design and put forth some of its shortcomings. This design addresses the concerns we raised about SHiP

(cf. Section 2.2.3) by leveraging a staggering amount of features of very different kind. This allows to generalize to a far greater number of workloads. However, the feature selection methodology seems to over fit over the workloads used for training. As a matter of fact, Chapter 3 shows that when applied to modern workloads such as graph-processing, MPPPB fails to deliver performance improvement. Thus, hinting for a more diverse approach to feature training.

2.2.5 Optimal Replacement Approximation

Approximating optimal replacement has always been the goal of computer architects. Through different heuristics such as LRU or RRIP-based policies (cf., Section 2.2.2), computer architects have tried to approximate the optimal algorithm, Belady's MIN [18]. In their paper "*Back to the future: Leveraging Belady's algorithm for improved cache replacement*" [45], Jain and Lin proposed the Hawkeye predictor, the first predictor of its kind, an affordable and implementable approximation to Belady's MIN [18] optimal replacement policy. The Hawkeye replacement policy is composed of two main components: i) *OPTgen*, a component that produces an approximation to the optimal caching strategy; ii) a PC-based predictor that learns from *OPTgen*. The following paragraphs provide detailed information about the *OPTgen* algorithm and its implementation.

2.2.5.1 The *OPTgen* algorithm

OPTgen provides a binary output about the block of interest: whether the block in question would have been cached if Belady's MIN [18] replacement policy had been used in the cache. *OPTgen* operates by looking at the memory reference history and, starting from the oldest reference and moving forward, assigns available cache capacity to lines in the order they are reused. In their paper, Jain and Lin define concepts and structures that enable the *OPTgen* algorithm. These concepts and structures are the following:

2.2.5.1.1 Usage and Liveness Intervals: *OPTgen* defines a cache line's *usage interval* as the time period between the first reference to the line and the reference immediately before its next reuse. The *liveness interval* is the period during which a line remains in the cache under Belady's MIN replacement policy. One of the idea behind the *OPTgen* algorithm is to assume that the usage interval of X' represents its demand on the cache. The authors further assuming that a cache miss on X can be identified through the concepts of usage and liveness intervals we have just introduced. As a matter of fact, " X would be a cache miss if at any

point in its usage interval, the number of overlapping liveness intervals matches the cache's capacity"⁵.

2.2.5.1.2 Occupancy Vector: *OPTgen* employs an *occupancy vector* to keep track of the occupied cache capacity over time. Each entry in this vector contains the number of liveness intervals that overlap at a particular time. As modern micro-architecture rely on set-associative caches, *OPTgen* employs one occupancy vector per cache set.

When it comes to the update of the occupancy vectors, such actions are performed upon an access to a cache line. For instance, upon an access to the cache line X the following algorithm is used to update the occupancy vector of the set containing the cache line X :

1. If an access to X leads to an insertion, no changes are made to the occupancy vector.
2. If an access to X results in a reuse, *OPTgen* checks whether every element in the occupancy vector corresponding to X 's usage interval is less than the cache capacity.
3. If all elements within the usage interval meet this condition, *OPTgen* increments the elements in the occupancy vector within the usage interval. This implies that X would have been a hit in the cache at that time.
4. If the aforementioned condition is not met for any element within the usage interval, the occupancy vector is not modified, indicating that X would have been a cache miss.

The occupancy vector is updated based on this logic to determine whether a reference to a cache line would have resulted in a cache hit or miss under the OPT policy.

In Figure 2.6, we show an example of *OPTgen* updating an occupancy vector based on the accesses to a cache set. This example assumes a cache set with only two ways. This example shows a total of 12 accesses to the set. In this figure, accesses predicted to be hits for Belady's MIN are highlighted in blue. Similarly, accesses predicted to be misses for Belady's MIN are highlighted in red. Based on the access sequence described in Figure 2.6, the occupancy vector is updated as follows:

1. On $t = 0$, block A is accessed. The cache is empty, thus, A is inserted, and the occupancy vector is left untouched.
2. $t = 1$ corresponds to the first access to block B . It is inserted in the second way of the cache set as, thus far, this way remains unused. The occupancy vector is left unchanged.

⁵The cache associativity when considering set-associative caches.

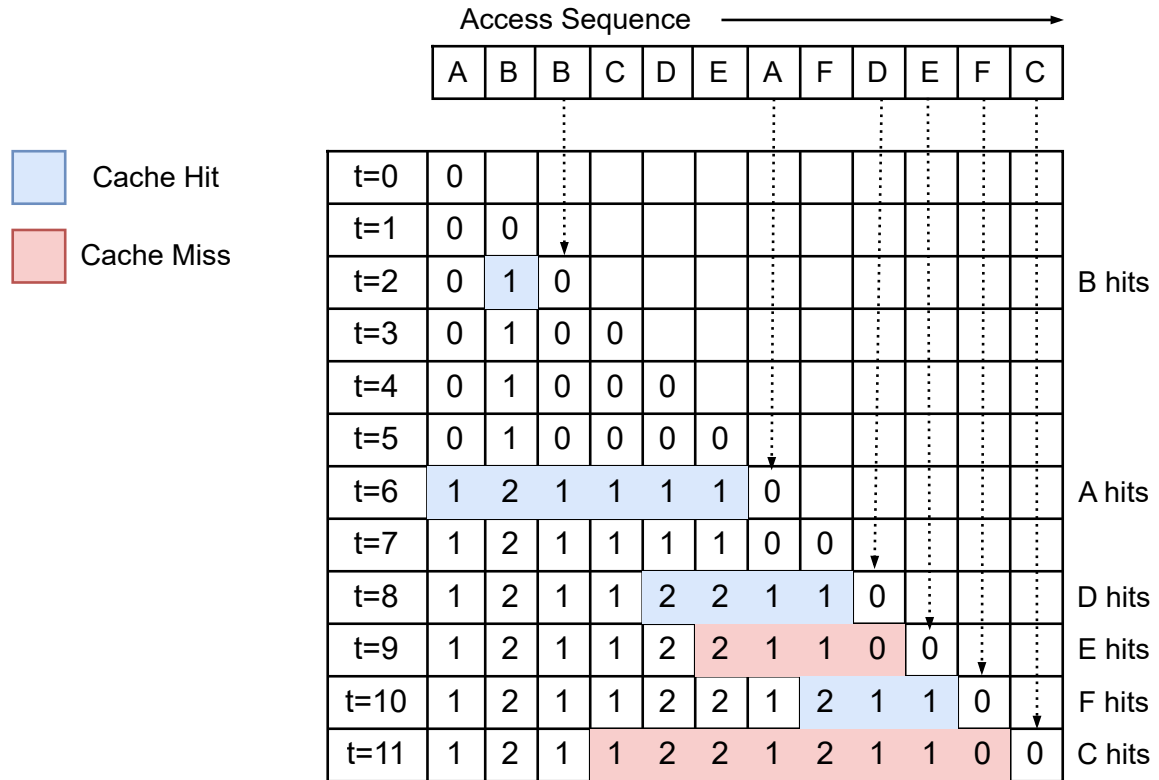


Figure 2.6 Example of the process of updating the occupancy vector for a cache set. In this example, *OPTgen* is applied to a cache set containing two ways.

3. $t = 2$ is a reuse of B . The occupancy vector element $vector[1]$ is incremented. This access would have been a hit for Belady's MIN as no value of the occupancy vector in B 's usage interval exceeds 2.
4. From $t = 3$ to $t = 5$, the cache accesses refer to blocks that have never been seen before. Thus, the occupancy vector is left unchanged.
5. At $t = 6$ the accesses A again. At this time, in the A 's usage interval, all values are below 2. Thus, the access to A would have been a hit for Belady's MIN. Finally, all values in A 's usage interval are incremented.
6. At $t = 7$, F is accessed. The block has never been seen before. Thus, the occupancy vector is left unchanged.
7. At $t = 8$, D is accessed. As it is a reuse, the occupancy vector is looked up in D 's usage interval. As no value is equal to 2, the access would have been a hit for Belady's MIN. Finally, the occupancy vector is incremented in D 's usage interval.

8. At $t = 9$, E is accessed. As it is a reuse, the occupancy vector is looked up in E 's usage interval. We see that an entry is equal to 2. This indicates a miss for Belady's MIN. Finally, the occupancy vector is left untouched.
9. At $t = 10$, F is accessed. As it is a reuse, the occupancy vector is looked up in F 's usage interval. As no value is equal to 2, this access would have been a hit for Belady's MIN. Finally, the occupancy vector is incremented.
10. At $t = 11$, C is accessed. As it is a reuse, the occupancy vector is looked up in C 's usage interval. We see that a few entries have values equal to 2. This indicates a miss for Belady's MIN. Finally, the occupancy vector is left untouched.

2.2.5.1.3 Training the predictor with *OPTgen*: As stated earlier, the *OPTgen* algorithm is responsible for providing predictions on whether an access to a cache line X would result in a cache hit or in a cache miss. However, *OPTgen* on its own is impractical as it cannot provide predictions for the wide variety of memory accesses that will come down to the LLC.

To handle that task, a simple PC-based predictor is placed alongside the LLC and is in charge of learning the behavior of PCs when it comes to cache hits or cache misses as predicted by *OPTgen*. Upon a cache hit prediction from *OPTgen* on cache line X , the PC that last accessed X is trained positively. Conversely, upon a cache miss prediction, the PC that last accessed X is trained negatively.

2.2.5.1.4 Issuing an *OPTgen* prediction: As we have seen before, to determine whether a reference to a cache line, denoted as X , would result in a cache hit or miss under Belady's MIN policy, *OPTgen* checks if, at any point within X 's usage interval, the number of overlapping liveness intervals matches the cache's capacity. If there is no match, it is considered a cache hit; otherwise, it is considered a cache miss.

OPTgen is able to recognize both long-term and short-term reuse patterns that are cache-friendly and can be implemented in hardware with minimal logic because the occupancy vector can be efficiently maintained using simple read, write, and compare operations. This technique provides insight into how the MIN cache replacement policy would perform in practice based on the historical memory reference patterns.

2.2.5.2 Operation of the Hawkeye Replacement Policy

Upon a prediction request from the replacement policy, the PC-based predictor is consulted to get a prediction. The prediction has one of the following outcomes: i) the block is predicted to be *cache-friendly*; ii) the block is predicted to be *cache-averse*.

Upon a cache-friendly prediction, the accessed block is either inserted or promoted to the immediate re-reference class (*i.e.*, $RRPV = 0$) to favor that block remaining in the cache. Conversely, upon a cache-averse prediction, the block is either inserted or placed in the distance re-reference class (*i.e.*, $RRPV = 2^n - 1$, where n is the number of bits used for RRPV values), thus favoring the block for eviction.

2.2.5.3 Advancements in the Hawkeye Replacement Policy

After thorough exploration and multiple iterations of the Hawkeye replacement policy, Lin and his team advanced the field by constructing cache replacement designs that are more intricate and resilient, building upon the foundations laid by Hawkeye. One of these innovative designs, known as Glider [87], capitalizes on insights gained from offline training of a sophisticated machine learning model⁶. This meticulous process enabled the authors to craft a refined iteration of Hawkeye, resulting in notable additional performance improvements.

Finally, and to reflect on what we have done for the other replacement policies covered in this dissertation (*cf.*, Section 2.2), we put this work in perspective. As stated previously in the final paragraph on the SHiP predictor (*cf.*, Section 2.2.3), relying solely on the PC has great limitations that only appear when using modern workloads such as graph-processing and not only the conventional SPEC 2006 and SPEC 2017 [91, 92].

2.3 Hardware Prefetching

In the relentless pursuit of optimizing the performance of modern computer systems, the efficient management of memory access remains a paramount concern. Efficiently accessing memory is not solely dependent on the speed of memory devices but also relies heavily on the ability of the cache hierarchy to timely provide the data required by the core. Modern processors employ hardware mechanisms to predict and fetch ahead-of-time data blocks that will be consumed by the processor in the near future. This proactive data retrieval process, known as hardware prefetching, has become a critical component in bridging the ever-widening gap between CPU processing speeds and memory access latencies. The concept of hardware prefetching has evolved over the decades in response to the increasing complexity of applications and the growing disparities between memory and processor speeds. Traditional memory hierarchies, comprising multiple cache levels and the main memory, have been augmented with sophisticated hardware prefetchers to minimize the performance bottlenecks arising from memory latency. These hardware prefetchers, often

⁶The authors leverage LSTMs couples an attention to label PCs as either *cache-friendly* or *cache-averse*. The insights obtained through the analysis of the attention layer serve to build the prediction mechanism.

embedded within the CPU micro-architecture, play a pivotal role in improving the overall system throughput by predicting and fetching data ahead of time from the processor.

A rich history of innovation and continuous adaptation characterizes the landscape of hardware prefetching. Researchers and engineers have explored various prefetching strategies, each tailored to address specific memory access patterns, workloads, and architectural constraints. As we will see later in this section, these strategies range from simple techniques like the stream and stride prefetchers to more advanced approaches that leverage intricate pattern recognition algorithms and classification mechanisms.

Moreover, as the computing ecosystem continues to evolve with the advent of multi-core processors, heterogeneous computing, and novel memory technologies, the role and significance of hardware prefetching have undergone transforming changes. The challenges posed by increasingly diverse workloads, constrained memory bandwidth, and energy efficiency considerations have spurred researchers to develop innovative prefetching solutions that go beyond conventional paradigms.

In this background section, we explore the realm of hardware prefetching. We will dive into some of the field's most recent and relevant works. We will highlight some of the pressing challenges such designs address and outline the trajectory of the prefetching research field over the years. Specifically, we will delve into the latest advancements [76, 71, 110, 13], such as IP Classifier-based spatial Prefetching (IPCP) in Section 2.3.1, Berti in Section 2.3.2, Indirect Memory Prefetcher (IMP) in Section 2.3.3, and DROPLET in Section 2.3.4. These works showcase the cutting-edge innovations in hardware prefetching and their implications for contemporary computing systems.

2.3.1 The IPCP prefetcher

First presented by Pakalapati and Panda, in their paper: *"Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching"* [76], the IP Classifier-based spatial Prefetching, or IPCP, is a pioneering hardware prefetching framework that comes with a significant number of novel ideas, advancing the field of hardware prefetching and in a broader sense, the domain of memory access optimization as a whole. IPCP represents a paradigm shift in the world of prefetching by harnessing the power of IP (Instruction Pointer) classification to dynamically and efficiently prefetch data. This innovative framework classifies access patterns into three distinct classes: i) constant-stride access patterns, ii) complex-stride access patterns, and iii) global-stream access patterns. In the paragraphs below, we provide ample details about these access patterns and how prefetching is performed in each situation.

2.3.1.1 Constant-stride Prefetching

An instruction that shows a constant stride in terms of cache line aligned addresses belongs to this class. It is a common pattern seen by instructions and can be prefetched using an IP-stride prefetcher. The goal of such a predictor is to predict memory accesses based on constant strides in memory addresses.

An IP-stride prefetcher is commonly built around a prediction table, known as the IP table, which is indexed by the Program Counter (PC). Each entry in this table contains information about the observed strides (constant step) and a confidence counter. The confidence counter is incremented when the same stride is repeatedly observed and decremented otherwise. It helps to determine when to prefetch data using the constant stride information. Additionally, the entry stores information about the last virtual page and the last cache line offset within a page. Changes in these values indicate a page change, which is used to calculate the stride. For example, a change from offset 63 to 0 with a page change in the forward direction indicates a stride of one. The process of using the IP table goes through the following two main phases: i) a training phase where the prediction table builds confidence upon a constant stride for a given PC, and ii) a prediction phase where the predictor has built enough confidence in a stride for a given PC. During the training phase, a PC is trained by observing memory accesses until it gains sufficient confidence to prefetch data. Once a PC is trained, it starts prefetching data based on the learned stride. The prefetch address is calculated as $(\text{current cache line address}) + k \times (\text{learned stride})$, where k varies from one to the prefetch degree. If a PC loses confidence, it temporarily stops prefetching and resumes when confidence is regained.

Overall, this technique, when leveraged upon constant-stride access patterns, helps improve memory access performance by predicting and prefetching data based on constant stride patterns observed in program execution.

2.3.1.2 Complex-stride Prefetching

Complex strides refer to memory access patterns that cannot be explained by the simple constant stride model, making them more challenging to predict. In an effort to accurately prefetch complex strides access patterns, the authors introduce the *Complex Stride* (CPLX) prefetcher.

Prefetching complex strides differs from the process of prefetching constant strides in several ways. For access patterns such as $A, A + 3; A + 6; A + 10; A + 13; A + 16; A + 20$ with strides 3; 3; 4; 3; 3; 4, a constant-stride prefetch would only provide sub-optimal performance. As a matter of fact, a constant stride prefetcher would not be able to predict stride 4. Thus,

the prefetcher would only provide 66 % of coverage. The CPLX prefetch is built around two tables: i) an IP Table that stores a hashed PCs associated with an n -bit signature encoding the past n strides seen by a given PC and ii) the *Complex Stride Prediction Table* (CSPT). The CSPT predicts future complex strides based on the signature and includes a 2-bit confidence counter similar to the Constant Stride class predictor presented above. The training and prediction phases are similar to the Constant Stride class, where confidence is built by repetitively seeing the similar complex-stride patterns, and prefetching is initiated based on the confidence level.

Even though they are build in relatively similar fashion the CPLX prefetcher is distinct from SPP [61]. While SPP captures memory deltas within a page, CPLX is IP-driven and focuses on the local order of complex strides caused by control and data flow. It is suitable for cases where memory accesses are not powers of two or where loops at different levels create varying stride patterns. CPLX is designed for L1D prefetching and offers reduced latency on the critical path, unlike SPP, which is optimized for L2C prefetching.

In summary, the CPLX prefetcher is designed to handle complex memory access patterns that don't fit the constant stride model, making it a valuable addition to improve prefetching accuracy in certain scenarios, especially in the L1D.

2.3.1.3 Global-stream Prefetching

In their research paper [76], Palakapati and Panda present an innovative prefetching scheme tailored for *Global Streams* (GS). They define a global stream as a collection of cache-aligned memory accesses within a compact region, typically characterized by a bursty pattern. This novel prefetching approach consists of two core components: i) the *Region Stream Table* (RST) and ii) an *IP table*.

The primary role of the RST is to establish an understanding of the density of memory accesses in one given region. The RST is structured as a table, with each entry containing several key fields, the most notable of them being the following three: i) a 32-bit bit-vector field, which enables the RST table to keep track of which memory blocks have previously been accessed; ii) the dense-count counter which counts the number of blocks in a given region that has been visited by load demand access, and iii) the trained bit which allows the RST to know whether or not a region has been deemed *dense* and its associated information can be used for prefetching. In the design proposed by Palakapati and Panda [76], each RST entry monitors a 2KB memory region⁷.

⁷Bigger regions have been considered in this work. However, the authors noticed that bigger regions lead to longer training time coupled with minimal performance gains.

When a memory region is accessed for the first time, an RST entry is allocated through an LRU replacement policy, the corresponding bit in the bit-vector is set to match the exact block that was accessed in this region, and the dense-counter is incremented. Upon other accesses to the same memory region, its associated dense-counter is incremented. Eventually, when the dense-counter surpasses a predefined threshold (e.g., 75 % of cache blocks accessed within a region), the region is considered dense, and the trained bit is set.

Upon a load demand access, the GS predictor is looked up to issue a prefetch request. To do so, the prefetcher examines the trained bit in the matching RST entry. If this bit is set, the PC associated with the load demand access is categorized as a GS PC. Once classified as a GS PC, the prefetcher issues a handful of prefetch requests to retrieve the next k cache lines based on the trained direction, where k is the prefetch degree.

Pakalapati and Panda, with their careful design proposal, enabled the IPCP prefetcher with pattern recognition capabilities. Being able to identify and classify access patterns in one of the three aforementioned classes allows IPCP to leverage a predictor tailored to such a class of access patterns, thus dramatically improving the prefetching accuracy and timeliness. The IPCP prefetcher can, in case it cannot confidently classify access patterns within one of the three aforementioned classes, fall back to the simple *Next Line* (NL) prefetching scheme. This NL simply prefetches the next k cache lines, where k is the prefetching degree of the NL prefetcher.

Finally, what sets IPCP apart is its remarkable ability to provide a good combination of accuracy, coverage⁸, and timely data prefetching with minimal hardware overhead, requiring a mere 895 bytes per core. IPCP's effectiveness extends beyond the L1D, communicating its classification information to the L2C, making it an attractive solution for multi-level cache hierarchies. Its exceptional adaptability and performance have positioned IPCP as a trailblazer in the quest for efficient memory access management in modern computing systems.

2.3.2 The Berti prefetcher

Berti, a prefetching mechanism presented by Navarro-Torres in "*Berti: an Accurate Local-Delta Data Prefetcher*" [71], stands out as a groundbreaking innovation in the domain of hardware prefetching. It makes a compelling case for L1D prefetching based on local, real-time deltas. What sets this prefetcher apart is its unwavering commitment to both the promptness and precision of prefetching, driven by its remarkable capacity to comprehend and adjust to a wide array of memory access patterns.

⁸IPCP shows accuracy of 80 % in the L1D, and coverage of 60 % in the L1D, 79 % in the L2C, and 83 % in the LLC.

Across a spectrum of workloads encompassing SPEC 2017 and GAP workloads [92, 16], Berti consistently proves its mettle by delivering exceptional combination of accuracy and coverage⁹, along with timely retrieval of data into the cache hierarchy. Even in resource-constrained scenarios with limited DRAM bandwidth and the dynamic landscape of multi-core architectures, Berti continues to excel. As a matter of fact, Berti outperforms other state-of-the-art prefetching mechanisms such as IPCP [76] (*cf.*, Section 2.3.1). When it comes to multi-core architectures, the authors show that Berti outperforms previously published works such as IPCP [76] (*cf.*, Section 2.3.1). When using four cores Berti yields 16.2 % geometric mean speed-up over the baseline as compared to about 14 % geometric speed-up for IPCP.

Furthermore, Berti distinguishes itself as an energy-efficient solution, showing the lowest dynamic energy consumption within the memory hierarchy when compared to existing prefetching techniques [9, 61, 76, 110]. As a matter of fact, Berti, when compared to IPCP, is the design that consumes the least extra energy over no prefetching. Considering SPEC 2017 workloads, this extra energy is limited to only 9.0 % as compared to 30.1 % for IPCP. Finally, when the GAP workloads are considered, this extra energy is as high as 14.3 % for Berti, while IPCP shows a staggering 86.9 % increase in energy. With its modest storage overhead of merely 2.55 KB per core, Berti represents a promising stride in bridging the *Processor-Memory Gap*, offering an unprecedented combination of performance and space efficiency.

Operating within the L1D, the Berti data prefetcher strives to achieve high accuracy through the learning and prediction of timely prefetch deltas using virtual addresses. It makes informed decisions about the precise timing and location for issuing prefetch requests.

2.3.2.1 Training the Prefetcher

The training process of Berti involves three key operations. First, Berti measures the fetch latency for cache lines in the L1D, monitoring both demand misses and prefetch requests to determine when prefetch requests should arrive at the L1D on time. Second, Berti learns timely prefetch deltas by analyzing the access history and their timestamps associated with the same PC. It identifies which accesses would have triggered timely prefetch requests through timestamp comparisons, enabling the discovery of timely prefetch deltas. Lastly, Berti calculates the coverage of each delta by comparing the frequency of their occurrence in searches within the history to the total number of searches. High-coverage deltas are those

⁹The authors [71] show that, on average Berti reduces demand L1D MPKI by 8.7 % and 33.4 % for respectively GAP and SPEC 2017 workloads [16, 92].

that frequently appear, indicating high accuracy. Coverage plays a critical role in accuracy; when a delta covers 100 % of cache lines, it results in 100 % accuracy.

2.3.2.2 Issuing predictions

Berti employs different prediction strategies based on three distinct delta coverage levels: i) the *high-coverage watermark*, triggering prefetches to populate all cache levels up to the L1D when coverage is high, and L1D MSHR occupancy is low; ii) the *medium-coverage watermark*, leading to prefetches aimed at filling caches up to the L2C regardless of L1D MSHR occupancy, as long as coverage is considered medium; iii) the *low-coverage watermark*, which results in prefetches only populating the LLC when coverage is low.

When initiating a prefetch request, Berti adds the selected delta to the address of the current access and places the resulting address into the Prefetch Queue (PQ). These prefetch requests are processed in a first-in, first-out (FIFO) order. Notably, Berti is trained with virtual addresses, so the generated prefetch requests also reside in the virtual address space. The physical address is acquired from the L2 translation look-ahead buffer (STLB). Successful translation results in block prefetching, while unsuccessful requests are discarded.

Berti's approach emphasizes prefetch accuracy through training with timely deltas. It ensures that prefetches are issued when they provide the most benefit and efficiently manages the generation and placement of prefetch requests in the cache hierarchy.

2.3.3 The IMP prefetcher

In their paper, "*IMP: Indirect Memory Prefetcher*" [110], Yu *et al.* introduce a pioneering solution in the domain of memory access optimization. IMP intends to address the intricate challenge of prefetching for indirect memory access patterns, a common occurrence in complex workloads like graph-processing and *Sparse Matrix-Vector Multiplication* (SpMV).

The authors observe that kernels of this nature exhibit a distinctive method for accessing memory, characterized by a recognizable $A[B[i]]$ pattern. Specifically, in graph-processing workloads [16], the array B is accessed in a contiguous manner and contains the set associated of indices pointing at the neighbours of a given vertex, while A denotes the vertices of the graph. Notably, although B is accessed sequentially, the content within it often follows an unpredictable pattern, giving rise to accesses to A following an *Irregular Access Pattern*.

Figure 2.7 shows a high-level memory representation of the adjacency matrix of a graph and attempts to illustrate the essence of what is an *Irregular Access Pattern* on a practical example. Adjacency matrix are typically represented as sparse matrices as the number of

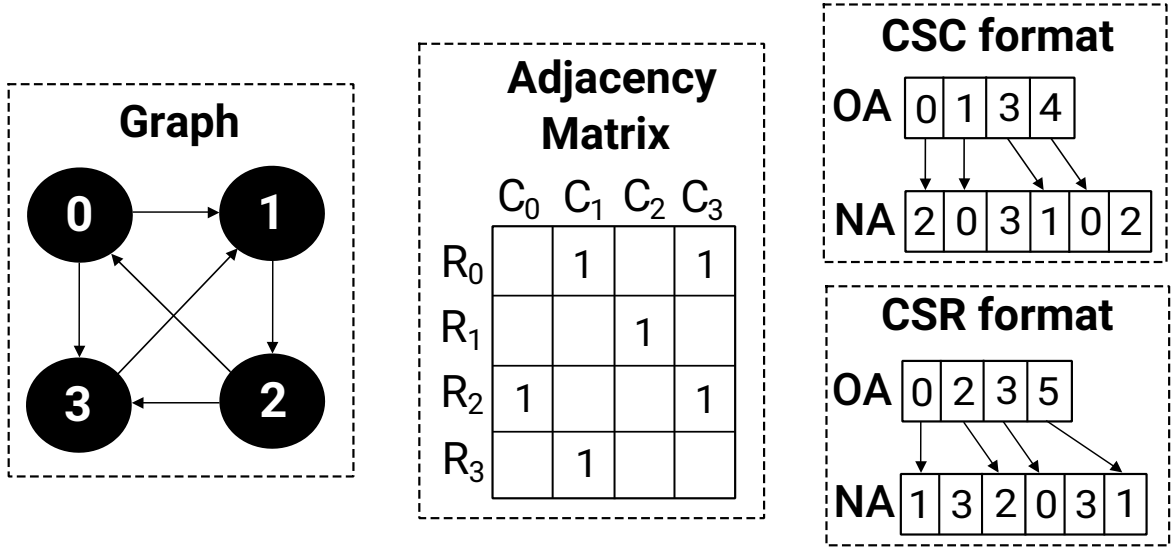


Figure 2.7 Representation of a graph's adjacency matrix in memory *Compressed Sparse Column/Compressed Sparse Row* (CSC/CSR) formats. Graph's adjacency matrix are commonly represented as sparse matrices in memory as they usually contain a significant amount of zeros. In addition to the adjacency matrix, this figure represents the *Offset Array* (OA) and the *Neighbours Array* (NA).

zeros¹⁰ they contain is usually very high. More specifically, the figure presents two arrays: i) the Offset Array (OA), and ii) the Neighbours Array (NA). An additional array is usually used to store data attached to vertices, the *Property Array* (PA). In order to index the PA, one has to go through the following steps:

1. For any given integer i between 0 and the size of the OA, we take to indices from the OA, $i_0 = OA[i]$ and $i_1 = OA[i + 1]$.
2. Values from the NA are collected from indices i_0 until i_1 . We denote these indices as $I = \{NA[i] \mid i \in \mathbb{N}, i_0 \leq i < i_1\}$.
3. Finally, PA is indexed using the set of index I collected from the NA.

As one can see, while the accesses to the OA and NA arrays are sequential; therefore presenting a significant amount of locality. Accesses to the PA array are very likely to point to very distant elements of the PA array. For instance, spanning through the entirety of the OA and NA arrays in the CSR format would have generated the index sequence $I = \{1; 3; 2; 0; 3; 1\}$ for the PA. One should also keep in mind that on such kind of workloads [16], datasets usually span several gigabytes, making such access patterns even more irregular.

¹⁰A zero in an adjacency matrix represents the absence of an edge between two vertices of a graph.

What sets IMP apart from other prefetching mechanisms is its ability to effectively identify *Irregular Access Patterns* and accurately prefetch for these access patterns. As opposed to other proposed mechanisms [3], IMP does not rely on any sort of software-provided information. This hardware prefetcher delivers innovative techniques that enable the analysis of memory access patterns and are able to discriminate *Irregular Access Patterns* from Regular ones. Such classification allows the prefetching logic to effectively anticipate which blocks will be required by the core in the future and fetch these blocks ahead of time. This classification task is accomplished through the three main components of the prefetchers: i) a *Stream Table* responsible for recognizing contiguous access to the array B , ii) an *Indirect Table* tasked with storing the characteristics of an *Irregular Access Pattern*, and iii) the *Indirect Pattern Detector* (IPD). These three components are effectively responsible for the discovery of *Irregular Access Patterns*.

$$\begin{aligned} ADDR(A[B[i]]) &= C \cdot B[i] + BaseAddr \\ &= (B[i] \ll shift) + BaseAddr \end{aligned} \tag{2.3}$$

In their work, Yu *et al.* propose a simple, yet comprehensive model of an *Indirect Memory Access*. This model is reflected in Equation 2.3. The computation of the *shift* and *BaseAddr* parameters for a specific Indirect Access Pattern relies on the collaborative functionality of the *Stream Table*, the *Indirect Table*, and the IPD. These hardware components work in tandem to solve Equation 2.3, providing the necessary parameters. Once this equation is successfully solved through the specified hardware modules, the prefetcher gains enough confidence in the identified Indirect Memory Access pattern. Consequently, the prefetcher starts issuing prefetch requests relative to the identified *Indirect Memory Access* pattern.

By doing so, IMP significantly reduces memory access latencies and enhances overall system performance. Its remarkable adaptability and ability to operate without specific software assistance make IMP a valuable addition to the toolkit of memory access optimization techniques, especially in scenarios where indirect memory accesses pose a substantial challenge. However, it is important to note that IMP is limited in its applicability by the nature of the workloads it is tailored towards.

2.3.4 The DROPLET prefetcher

In "*Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads*" [13], Basak *et al.* present DROPLET, a prefetching mechanism tailored to graph-processing workloads [16]. Through a thorough and extensive analysis of the behavior of graph-processing workloads, the authors provide numerous insights. These insights are later leveraged to build a comprehensive prefetching mechanism.

The first insight concerns *Memory Level Parallelism* (MLP) or the lack thereof. The authors, through meticulous analysis in the *ReOrder Buffer* (ROB), understood that the limitations in MLP were mainly due to chains¹¹ of load instructions with an average length of 2.5 linked by a true data dependency. Specifically, the authors mapped these instruction dependency application-level characteristics of these workloads. As discussed in Section 2.3.3, to access the PA array, one first needs to acquire structural information about the graph. Such information is typically acquired through the OA and NA arrays. In their paper, the authors refer to these as *consumer loads* and *producer loads*, respectively. This true data dependency between the *consumer load* and the *producer load* maps well to the indexing of the PA array of a graph, as one needs to acquire the index associated with a vertex from the NA and OA arrays.

The second insight provided by Basak *et al.* lies in the performance sensitivity of the PA array (referred to as *Property Data* by the authors). It is found that, in terms of performance, the PA array responds positively to an increase in the size of the LLC. As a matter of fact, the authors show that multiplying the size of the LLC by 4 provides the best performance improvement, while increasing the size of the L2C does not show any significant improvement in performance¹².

Finally, the third major insight proposed by the authors, as a corollary of the two previous findings exposed here, states that differentiating the prefetching between the *Property Data* and the *Structure Data*, respectively the PA array and a combination of the NA and OA arrays as per Section 2.3.3, would be beneficial for performance.

Building on these findings, the authors propose a prefetching mechanism composed of two main components: i) an L2 stream prefetcher dedicated to the *Structure Data* of the graphs; ii) a *Memory Controller-based Property Prefetcher* (MPP). The L2 stream prefetcher is dedicated to prefetching *Structure Data* and is tailored to strictly prefetching *Structure Data* of the graphs into the L2C. As discussed in Section 2.3.3, *Structure Data* (*i.e.*, denoted

¹¹In the context of this paper, a chain of load instructions is defined as two load instructions in the ROB, in which the latter depends on the former.

¹²The authors find that multiplying the size of the LLC by 4 increases performance by 17.4 % over the baseline, while changing the capacity of the L2C does not affect performance in any way.

as the NA and OA above) are accessed in a sequential manner, forming a very predictable and prefetch-friendly pattern. Finally, the MPP leverages the *Structure Data* to prefetch *Property Data* in the L2C.

2.4 Off-Chip Prediction

As mentioned above, in Section 2.1, researchers have recently explored new ways to extract performance and bridge the infamous *Processor-Memory Gap*. One of these techniques that we will be delving into in this section is called *Off-Chip Prediction*. *Off-Chip Prediction* can be understood as an array of techniques attempting to predict whether, upon a load demand access issued by the core, if the cache hierarchy will serve this access or if it will result in a costly access to the main memory. Getting such a prediction allows architects to use optimization techniques such as speculatively issuing an access to the main memory, or directly accessing the predicted cache level, such that the latency of the load demand request is partially hidden. Throughout this section, we will be exploring two of the main Off-Chip predictions. Firstly, in Section 2.4.1, we will dive into the Level Prediction (LP) mechanism proposed by Jalili and Erez in their paper, "*Reducing Load Latency with Cache Level Prediction*" [48]. Secondly, in Section 2.4.2, we will explore the more recent Hermes design proposed by Bera et al. in their paper, "*Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction*" [19].

2.4.1 LP: Cache Level Prediction

In their paper titled "*Reducing Load Latency with Cache Level Prediction*" [48], Jalili and Erez shed light on the challenges arising from modern workloads, particularly in the context of complex multi-level cache hierarchies within contemporary micro-architectures. While these multi-level cache structures have proven to be a centerpiece in enhancing the performance of traditional workloads like SPEC 2006 [91], SPEC 2017 [92], and PARSEC [90], they can lead to long-lasting load access latencies when dealing with modern workloads such as graph-processing. The authors illustrate that, for certain workloads such as *Page Rank* (PR) [16], the L2C and LLC caches struggle to effectively eliminate cache misses from the upper levels, exerting substantial pressure on the main memory. Indeed, the authors demonstrate that the ratios between L1D and L2C MPKI, as well as L2C and LLC MPKI, are very close to one. These metrics strongly suggest that both the L2C and the LLC fail to fulfill their originally intended roles. This shows that a significant portion of the L1D misses eventually require a DRAM access, thus missing in all caches levels. However, upon misses,

caches introduce a latency overhead which further slows down the execution and increase the power required by the processor¹³.

To address this issue, the authors propose a memory hierarchy level predictor that facilitates direct cache access, effectively reducing access latency with minimal modifications to the existing memory hierarchy. This innovative approach blends the advantages of decreased access latency achieved through non-sequential (parallel) lookups while preserving the cost-efficiency and simplicity of a sequential hierarchy.

The *Level Predictor* (LP) is designed to predict the cache level in which a specific data block is located. This prediction is then leveraged to retrieve the block directly from its location in the cache hierarchy. As a matter of fact, a read request is sent to the predicted cache level and MSHR entries are allocated in the caches laying between the predicted cache level and the core. Thus, this technique has potential to dramatically reduce the load access latency. Its positive impact extends even to applications characterized by high cache hit rates¹⁴.

The LP architecture comprises two vital components: i) the *LocMap* and ii) the *Popular Level Detector* (PLD). The *LocMap*, organized as a flat data structure, resides in a designated section of the main memory, storing information regarding the memory level of each block. With a storage requirement of just 2 bits per block, a single entry of the *LocMap* provides location information for 256 blocks. Therefore, the *LocMap* incurs an overhead of only $\frac{2}{512} = 0.39\%$ of the main memory size. To complement this, a small 2 KB cache, known as the *MetaData Cache*, maintains a high hit rate while keeping access energy consumption at a minimum. The *PLD* utilizes a set of 32-bit counters, one per cache level, to record hits in each level. Leveraging a simple heuristic, the authors utilize these counters to deliver a cost-effective prediction upon a metadata cache miss.

This study delivers several notable findings and contributions. First, it highlights the potential benefits of non-sequential cache access for graph-processing and scientific computing applications, even those characterized by high cache hit ratios. Second, this work also presents an efficient and cost-effective level prediction mechanism, the *Level Predictor* (LP), whose performance surpasses that of conventional cache miss predictors when used alongside advanced prefetchers. Furthermore, the LP architecture is thoughtfully designed to minimize the need for micro-architectural modifications. It adeptly manages any mispredictions by leveraging the cache coherence directory and existing address-matching logic. Notably, this

¹³These ratios are close to one for benchmarks exhibiting highly irregular access patterns such as graph-processing workloads. When considering SPEC 2006 and SPEC 2017 workloads, these ratios are usually much higher, indicating a good efficacy of the cache hierarchy.

¹⁴Such workloads, for the most part, belong to traditional benchmark suites, such as SPEC 2006, SPEC 2017, and PARSEC [91, 92, 90].

approach significantly enhances various workload types, such as general, scientific, and graph-processing. On average, it leads to a remarkable 7.8 % performance improvement compared to a baseline configuration with aggressive prefetchers. Additionally, the LP approach reduces energy consumption within the cache hierarchy by an average of 18 %, eliminating unnecessary miss lookups.

2.4.2 Hermes: Off-Chip Load Prediction

In contemporary high-performance processors, the performance of the core can be significantly hampered by long-latency load requests that eventually require a DRAM access. These type of access are often referred to as off-chip loads in the literature. These off-chip loads have the potential to stall the processor when they fail to find data in on-chip caches, thereby imposing limitations on core performance. To mitigate this issue and augment the core's performance, computer architects have traditionally relied on two approaches: i) employing advanced hardware prefetchers and, ii) continually enlarging on-chip cache hierarchies.

Nevertheless, despite the progress made in these two areas, two key trends have emerged revealing opportunities for further performance enhancements. Firstly, even the most sophisticated prefetchers available today is only able to accurately predict around half of long-latency off-chip load requests, primarily due to the inherent challenge of learning highly irregular access patterns in modern workloads. Secondly, a substantial portion of the latency associated with off-chip load requests stems from the time spent accessing multi-level on-chip cache hierarchies. This second observation results from the ever-expanding size and intricacy of on-chip caches in modern processors.

In their paper "*Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction*" [19], Bera *et al.* propose to accelerate the processing of long-latency off-chip load requests by eliminating the on-chip cache access latency from the critical path. This objective is realized through the introduction of a novel technique named Hermes. Hermes operates by predicting which load requests are likely to require an DRAM (*i.e.*, off-chip) access and initiates the direct retrieval of data block from the main memory while simultaneously accessing the on-chip cache hierarchy. The ingenious aspect of Hermes lies in concealing the on-chip cache access latency beneath the main memory access latency, effectively reducing the overall latency of off-chip load requests. Furthermore, Hermes works in synergy with hardware data prefetchers and serves to decrease memory access latency for off-chip loads that prefetchers may not accurately predict.

Hermes also confronts the challenge of accurately identifying off-chip load requests by introducing POPET, a lightweight perceptron-based off-chip predictor. POPET is able to learn and recognize off-chip load requests based on a variety of program features. It diligently

observes these features for each load request, makes predictions regarding their off-chip nature, and subsequently triggers speculative load requests to the main memory as needed.

In summary, this work makes notable contributions:

- It identifies opportunities to enhance modern processor performance;
- It introduces Hermes, a novel off-chip prediction mechanism. A technique that efficiently reduces off-chip load request latency;
- It introduces POPET, a perceptron prediction mechanism at the core of Hermes. This predictor is built and tuned towards off-chip prediction and shows great prediction accuracy.

Similarly to previous designs, we put Hermes into perspective. As stated previously, Hermes, upon an off-chip prediction issues a speculative load request to the DRAM. These speculative DRAM accesses increase the total number of DRAM transactions. As we will see in Chapter 5, in the context of limited DRAM bandwidth, Hermes struggles at delivering significant performance improvement as the available DRAM bandwidth becomes a bottleneck.

CHAPTER

3

CHARACTERIZING THE IMPACT OF LAST-LEVEL CACHE REPLACEMENT POLICIES ON BIG-DATA WORKLOADS

L'exploration ne consiste pas seulement
à découvrir de nouveaux territoires,
mais aussi à comprendre les mystères de
la vie elle-même.

Alexandre Yersin

3.1 Introduction

THE vast disparity between main memory and CPU speed has led to hierarchical caching system in modern CPUs. The goal of the cache hierarchy is to keep data on-chip, close to the cores that are accessing it, thus avoiding hitting the memory wall [107]. Although computer architects highlighted the need for multiple levels to the cache hierarchy, the Last Level Cache (LLC) suffers from a high latency compared to the other cache levels. In addition, the LLC suffers from poor temporal and spatial locality in the access sequence as some accesses are filtered by the upper levels of the hierarchy. This phenomenon is exacerbated when considering emerging workloads such as big data or graph-processing workloads displaying highly irregular behavior. Thus, emerging workloads require more sophisticated cache replacement policies that can cope with a broader set of workloads than the traditional ones.

State-of-the-art LLC replacement policies such as MPPPB [55], Glider [87], Hawk-eye [45], SHiP [105], DRRIP, and SRRIP [47] show significant improvement when challenged by SPEC CPU 2006 [46] and SPEC CPU 2017 workloads. However, when facing workloads representative of another part of the spectrum of the HPC applications, these policies fail at delivering significant improvement over the baseline LRU policy. Such workloads with highly irregular behavior prevent the LLC replacement policies mentioned above from capturing the access patterns and, therefore, producing meaningful predictions and decisions. To address this issue, we argue that future work on LLC replacement policies should consider a more extensive set of workloads such as the one we study in this chapter, which is composed of the following benchmark suites:

- the GAP benchmark suite[16].
- the XSBench benchmark suite[98].
- Qualcomm workloads for the CVP1[28] championship.

This chapter also proposes two MPPPB variants that increase its benefits. First, we propose *Multi-Sampler Multiperspective (MS-MPPPB)*, a variant of MPPPB that uses four samplers and perceptron structures. MS-MPPPB adapts its replacement policy to the workload in a phase-wise manner, selecting the sampler that provides the best predictions out of the four available and using the most accurate sampler to make predictions and drive placement, promotion and bypass decisions in the LLC. Second, this chapter proposes *Multiperspective with Dynamic Features Selector (DS-MPPPB)*, another variant of MPPPB that is also able to adapt its behavior to the execution phases of the workloads by dynamically selecting the most accurate subset of 16 features from a bigger pool of 64 features.

This chapter makes the following contributions:

1. It evaluates state-of-the-art LLC replacement policies over a broader set of benchmark suites than usually considered in the literature. The selected benchmark suites better represent current and emerging big data and scientific workloads on HPC systems. This chapter considers the SPEC CPU 2006 and the SPEC CPU 2017 suites, a large set of workloads provided by Qualcomm for the CVP1 championship, the GAP benchmark suite, and the XSBench. This chapter also takes the opportunity to build knowledge on these workloads and analyzes their behavior and impact on the LLC and the memory hierarchy, thus paving the way for further work.
2. We present MS-MPPPB and DS-MPPPB, two novel LLC replacement policies derived from MPPPB. The main idea behind both schemes is to improve the accuracy of the predictions by dynamically selecting the most accurate features for each phase of the running workload. On a set of 50 cache intensive benchmarks, these new designs respectively yield a geometric mean speed-up of 8.3 % and 8.0 % over LRU, and outperform all the state-of-the-art approaches.

The rest of this chapter is organized as follows: Section 3.2 describes the workloads evaluated in this chapter. Thorough descriptions of state-of-the-art cache replacement policies can be found in Section 2.2. Section 5.3 motivates the need for additional benchmarks in the evaluation of LLC replacement policies. Section 4.3 proposes MS-MPPPB and DS-MPPPB, two designs derived from MPPPB that achieve higher accuracy on the studied benchmarks. Section 5.5 defines the evaluation methodology. Section 3.7 presents the results of our experiments and comment on them. Finally, Section 3.8 remarks the main conclusions of this work.

3.2 Workloads

Benchmarks are of paramount importance for the computer architecture community, as they are used in practically all the stages of processor development, from the very initial research to the final performance verification of the processors that are manufactured for all market segments, from embedded devices to the most powerful supercomputers in the world. Benchmark suites are composed of a series of codes and representative input sets, and their goal is to mimic the behavior of real workloads to define the performance goals of a processor design and to bring to light unexpected design issues. Hence, their choice is of crucial importance. This section presents a set of benchmarks that are commonly used by the community to model the behavior of different types of workloads.

3.2.1 SPEC CPU Benchmark Suites

The SPEC CPU benchmark suite [46] is a set of benchmarks aimed at studying the performance of CPU designs. These benchmarks are well-known and highly used by the computer architecture community, specially to evaluate new proposals in the area of microarchitecture. These benchmarks provide representative codes of real compute intensive workloads such as compilers, data compression, AI algorithms, and physics. These workloads are mostly scientific applications or commonly used algorithms in computer sciences such as data compression and parsers which loop over data structures in a reasonably predictable manner, which allows the cache structures of the CPUs to leverage the locality of these workloads. However, although these benchmarks cover a broad spectrum of applications, they do not represent some codes running on current HPC systems and mobile devices.

3.2.2 GAP Benchmark Suite

To help to standardize the evaluation of big data and graph processing algorithms, Beamer *et al.* proposed the GAP Benchmark Suite[16], a set of domain specific workloads that include graph computational kernels as well as representative input graphs. These domain specific workloads provide computer architects with the ability to extend their working sets of workloads. The benchmark suite provides a standardized evaluation framework for commonly used graph algorithms such as *Page Rank* and *Connected Components*, along with standard graph inputs available in industry and research.

3.2.2.1 Graph kernels

Next we provide a short description of each of the six graph kernels available in the benchmark suite.

Breadth-First Search (BFS) was proposed in 1945 by Konrad Zuse and it is one of the most well-known and widely used graph processing algorithms. Its principle is rather simple, and it comes down to a straightforward statement: first, one designates a root vertex to initiate the search algorithm, then the kernel traverses all the neighbouring vertices before moving to the next depth level.

Single Source Shortest Path (SSSP) is a prevalent problem in graph theory and engineering in general. This algorithm computes the distance to any reachable vertices from a given source vertex, being the distance between two vertices the minimum sum of edge weights along a path connecting the two vertices.

Page Rank (PR), invented by Larry Page to quantify the popularity of a web page, is a widespread algorithm in our daily life as it allows search engines to build meaningful

proposals to our questions. It is an iterative algorithm that associates a score (a PageRank) to each vertex of the graph. During an iteration, the algorithm updates the score of every vertex proportionally to the sum of the scores of its incoming neighbourhood. The algorithm stops when the variation of PageRanks in the graph falls below a limit, which means that the sum of the variations of the scores of all the vertices between two steps is below a certain threshold.

Connected Components (CC) is an algorithm meant to identify and label connected components in a graph. A connected component is a sub-graph in which its paths connect any two vertices, and the vertices of the sub-graph are not connected to any other vertex in the super-graph.

Betweenness Centrality (BC) is a crucial concept in graph and network theory that allows measuring the influence of a vertex in the data transfers of a network, assuming ideal transfers through the shortest paths.

Triangle Count (TC) is an algorithm that is mostly used in social network analysis to detect communities by detecting triangles in a graph. Triangles are a group of three vertices directly connected.

3.2.2.2 Input graphs

The GAP Benchmark suite comes with five inputs graphs of diverse origin (synthetic versus real world). The real world data sets model the connection between people, websites and roads. When selecting these real world graphs, the authors paid particular attention to the size of the graphs so that they can fit in the memory of most servers while stressing the cache hierarchy of such systems.

Twitter is a crawl of Twitter that has been commonly used by researchers to evaluate prior work and thus allows fair comparisons. It allows working with a typical example of social network topology, and its real world origin gives it interesting properties such as irregularities.

Web is a web crawl of the .sk domain. Even though it has a large size, it exhibits good locality and high average degree.

Road is an input graph that models the distances of all of the roads in the USA. Although it has a modest size compared to the other graphs available, it has a rather high diameter that can cause some algorithms to present large execution times.

Kron provides continuity with prior work as it has been used frequently in research. This graph uses the Kronecker synthetic graph generator.

Urund represents a worst case scenario as all vertices have an equal probability of being a neighbour of every other vertex.

3.2.3 XSBench workloads

XSBench workloads[98], as stated by their authors Tramm and Siegel, are meant to represent the *most computationally intensive steps of a robust nuclear core Monte Carlo particle transport simulation*. These workloads provide a variety of grid types, sizes and browsing algorithms, which allows computer architects to stress the memory hierarchy of a CPU in different ways and to expand their working set of workloads towards new scientific applications.

The XSBench suite allows customizing the code that will be effectively executed in order to stress the memory hierarchy. The benchmark suite relies on a handful of parameters to achieve this flexibility. In this work we focus on the three parameters that put more pressure on the cache hierarchy and, thus, have a higher significance for this work.

3.2.3.1 Problem size

When solving the particle transport problem, the size of the problem has a dramatic influence on performance and on the stress that is being put on the memory hierarchy. Eventually, increasing the size of the problem has a significant impact on performance as data structures are much larger, so we use the two largest sizes of grid available.

3.2.3.2 Grid type

This parameter allows the user to select among three types of grids. The *nuclide* grid is known as a naive implementation and does not require any additional memory other than what is necessary to store the point-wise cross-section data. However, it is computationally intensive as the benchmarks execute a binary search with high frequency. *Unionized* is a grid type that allows for higher performance as it uses an acceleration structure to reduce the number of binary searches triggered during the execution. Here, this optimization sacrifices memory footprint to leverage increased performance. The *hash* grid is presented as a competitive alternative to the unionized grid type as it allows to achieve similar performance while using far less memory.

3.2.3.3 Number of cross-section look-ups

This parameter sets the number of look-ups to perform per particle.

3.2.4 Industrial workloads

During the CVP1 contest, the evaluation of Value Prediction mechanisms used a set of over 2000 industrial workloads provided by Qualcomm. These are typical server and database workloads such as Redis and MongoDB, among others. Real world database workloads traverse vast amounts of data while processing a query and show low reuse of data over time. Thus, these workloads are known to be memory intensive and they stress the LLC more than the SPEC CPU 2006 and the SPEC CPU 2017 workloads.

3.3 Cache Replacement Policies

While developing new cache replacement algorithms for LLCs, one needs to evaluate the policy against a set of workloads that show the behavior of interest. This section reviews the most relevant cache replacement algorithms designed for LLCs. As this work studies the impact of emerging workloads on the LLC, we present the state-of-the-art replacement policies developed for this specific cache level. The cache replacement problem is slightly more complex in the context of the LLC than in the context of L1 and L2 caches. Although the underlying idea remains the same, the LLC suffers from poor locality as the upper-level caches filter accesses and leave only a cluttered sequence to the LLC. To cope with this particular replacement problem, researchers have come up with more and more sophisticated design ideas to leverage higher prediction accuracy and performance.

The next subsections present the most relevant state-of-the-art work on LLC replacement policies. However, More ample information about cache replacement policies can be found in Section 2.2.

3.3.1 Reuse Distance Prediction

As reuse distance is a crucial concept when it comes to cache replacement, recent works focus on proposing new techniques to build run-time approximations of the distance to the next reuse of a cache block. Re-reference Interval Prediction (RRIP) and all its derivatives are efficient yet light-weight implementations of reuse distance prediction.

The main idea behind RRIP is the classification of blocks into re-reference classes. In their work, Jaleel *et al.* propose three versions of the RRIP replacement policies[47, 80]: SRRIP, BRRIP, and DRRIP. The former, scan-resistant, is limited to always inserting new coming blocks in a fixed class. In contrast, BRRIP provides more flexibility by frequently inserting blocks in the distance re-reference class and infrequently inserting blocks in the long re-reference class. Finally, DRRIP leverages Set-Dueling to determine which of SRRIP

and BRRIP is best suited for a given workload or program phase, making it both scan and thrash resistant.

3.3.2 Signature-based Hit Predictor

Building on the reuse distance prediction [47, 80] framework built by Jaleel *et al.* and program-counter based dead block prediction [58], Wu *et al.* [105] proposed a LLC replacement policy design that uses a program-counter based signature as a feature.

As stated while describing reuse distance prediction mechanism in Section 3.3.1, SRRIP learns the re-reference intervals of the living cache blocks relatively to one another. The primary feature of *Signature-based Hit Predictor (SHiP)* [105] is that, not only it allows the SRRIP policy to learn the relative re-reference intervals, but it also tries to learn the likelihood of cache blocks to experience hits through a feature. The intuition being that cache blocks with the same signature behave comparably. In order to learn the likelihood of a cache block to experience further hits, SHiP maintains a prediction table with an entry per signature. When a signature gets hit, the associated saturating counter is incremented. Conversely, when a signature misses, the associated counter is decremented.

With the prediction values thus learned, SHiP modifies SRRIP policy for insertion by inserting new coming cache blocks in the distant re-reference interval if the prediction associated with the signature of that cache blocks is zero. A zero in the prediction table gives a strong hint that the associated signature belongs to the distance re-reference interval.

3.3.3 Multiperspective Reuse Prediction

The *Multiperspective Reuse Prediction* [55] cache replacement algorithm (hereafter MPPPB) leverages perceptron learning for reuse prediction and drives placement, promotion and bypass decisions. This replacement policy extends the idea of features developed in previous work [58, 105, 96] to achieve higher accuracy. It is essentially made of two components, a sampler and a perceptron predictor. The sampler, based on observations of block evictions relatively to its features associativity, is responsible for triggering learning signals to the perceptron. The perceptron, based on the learning signals triggered by the sampler, updates its prediction tables.

MPPPB relies on the idea of correlating reuse prediction of a cache block with a large number of features that ranges from PCs to characterizing bursty access patterns. In this context, a feature can be defined as a hash function applied to cache block characteristics such as the PC or the physical address. When a prediction request occurs, the perceptron selects weights out of its prediction tables using hashes of multiple features. Each feature

is hashed to index its prediction table, and the weights obtained are gathered in a single prediction value by a simple addition and compared to a set of thresholds to drive actions such as bypass, promotion and placement.

Perceptron learning is used to update the weights of the prediction tables through the learning algorithm. At the time a sampled block is reused or evicted, the perceptron updates the weights of the prediction tables associated with the last access to this block, according to the perceptron learning rule. For instance, if a block hits in the sampler while having its LRU stack position lower than the associativity of a feature, it is trained positively for that feature. Conversely, if a block gets demoted beyond the associativity of a given feature, it is trained negatively for that feature.

With this work, Jiménez and Teran demonstrated the usefulness and impact of combining multiple features. Among the correlating features, the sequence of PCs leading to the usage of a block is one; however, the sequence of PCs is highly filtered by the other levels of the cache hierarchy, making it inaccurate for predictions. The introduced additional features such as bits extracted from the memory address help mitigating the inaccuracy of a filtered PC sequence. MPPPB relies on this idea of combining multiple features while significantly augmenting the set of available features.

3.3.4 Optimal Replacement Approximation

The Hawkeye[45] replacement policy marked the birth of a new class of cache replacement algorithms aiming at approximating, in a relatively affordable way, optimal but unimplementable algorithms such MIN [18].

Hawkeye and its successor, Glider[87], are primarily made of two major building blocks: an optimal solution approximation component and a predictor that learns from the former component. The predictor is used to compute predictions and to trigger actions based on these predictions. The first component provides a binary output about the cache block of interest: needs to be cached or not. For this outcome, the predictor gets trained for the associated PC as it is a PC-based predictor. When the replacement policy requests a prediction to drive its decision making, the predictor is indexed, and it uses its outcome to place blocks in the matching RRIP categories, thus prioritizing eviction for blocks classified as cache-averse. Conversely, blocks identified as cache-friendly tend to stay in the immediate-reuse category.

Further work on the Hawkeye predictor provided it with a more complex predictor infrastructure. That infrastructure, named Glider, leveraged on the knowledge obtained through the offline training of a machine learning model, yielding additional performance improvements.

3.4 Motivation

To highlight the need for new benchmarks in the context of the development of new cache replacement policies for LLCs, we provide a quantitative analysis to build intuition on why the current state-of-the-art techniques need to take into account a broader set of workloads in the process of their constructions. This analysis relies on results obtained using the simulation methodology detailed in 5.5.

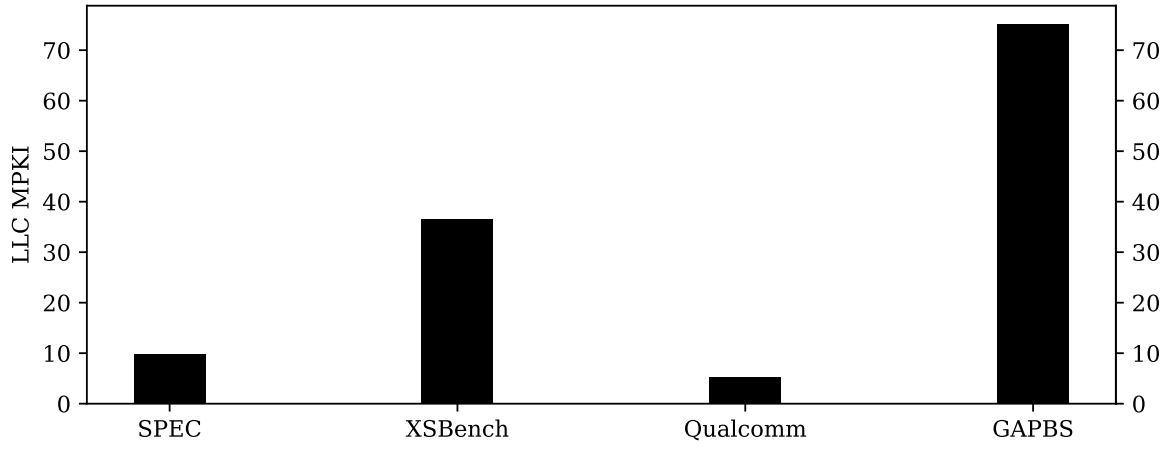
Figure 3.1a shows the average LLC MPKI for each of the benchmark suites described in Section 3.2 using the baseline LRU replacement policy. In both Figures 3.1a and 3.1b we only consider the cache-intensive benchmarks of these benchmark suites, namely the ones which present a LLC MPKI over 1.0 with the baseline LRU replacement policy. The GAP benchmark suite and all the different runs of XSBench, with respective LLC MPKI of 78.29 and 36.62, provide a significantly higher LLC MPKI than what is provided by the SPEC CPU benchmarks. The industrial Qualcomm workloads and the SPEC CPU workloads, with respective LLC MPKI of 10.63 and 15.76, do not show such a high impact on the LLC.

These results demonstrate that big data graph processing workloads like the ones modeled in the GAP benchmark suite highly stress the cache hierarchy, and particularly the LLC, much more than well-known workloads such as SPEC CPU 2006 and SPEC CPU 2017 suites do. This is due to the nature of these workloads, where moving from edge to edge in a graph structure leads to extremely unpredictable and sparse access patterns[110, 112]. Also, the memory footprint of the inputs is an important factor, as jumping from edges to edges in an extensively large graph exhibits very low spatial and temporal locality, which are two key concepts in the design of cache replacement policies.

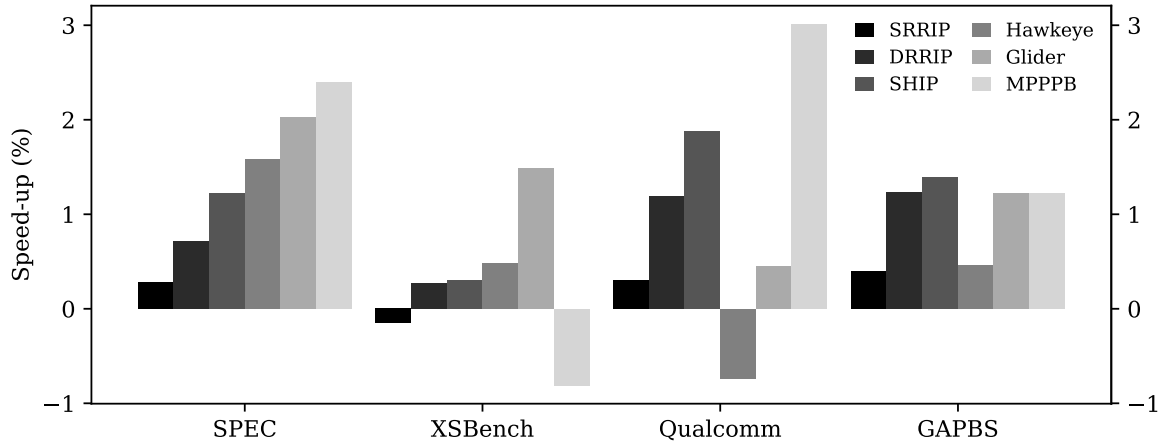
Figure 3.1b shows the speed-up of the most relevant state-of-the-art cache replacement policies presented in Section 2.2 over the baseline LRU policy for the different benchmark suites presented in Section 3.2. Each bar represents a single replacement policy, and each bar group stands for a benchmark suite. Results show that the different policies can catch a different kind of access patterns and are beneficial for different kind of workloads.

For the SPEC benchmarks, the plot shows that every single replacement policy is consistently delivering improvements over the baseline LRU and the previously proposed replacement policies in the literature, incrementally improving the performance of these benchmarks.

The replacement policies based on Reuse Distance Prediction (SRRIP and DRRIP) are consistent in their improvement over the baseline LRU, while more complex policies such as SHiP, Hawkeye, Glider and MPPPB have more difficulties generalizing to all the benchmark suites. The main reason behind this observation is that all these replacement policies rely either on assumptions about the access patterns (*e.g.*, SHiP and Hawkeye) or on a training phase over a set of workloads (*e.g.*, Glider and MPPPB). On the one hand, SHiP and Hawkeye



(a) Average LLC MPKI of the different benchmark suites using the LRU replacement policy.



(b) Geometric mean speed-up over LRU of state-of-the-art LLC replacement policies for the different benchmark suites.

Figure 3.1 LLC MPKI using the LRU replacement policy and performance improvement of state-of-the-art LLC replacement policies for the different benchmark suites.

use the observation that they can accurately learn the access patterns to the LLC using the PCs that triggered the memory accesses as a classification feature. On the other hand, both Glider and MPPPB rely on a learning algorithm that learns the access patterns of a couple of workloads and provide correlating features such as the i -th PC of the history or some bits of the physical address of the accessed block. Thus, although these state-of-the-art replacement policies deliver some performance improvements, they suffer from a structural bias that prevents them from generalizing to unexplored benchmarks in an optimal way.

The main conclusions arising from our analysis are:

1. The commonly used SPEC CPU 2006 and SPEC CPU 2017 suites no more represent a challenge for computer architects, as they are well studied and there are plenty of ingenious mechanisms that cope with their behavior.
2. The current state-of-the-art LLC replacement policies do not generalize well to new benchmarks.
3. Emerging big data and HPC workloads do represent a challenge for computer architects, as they stress more the cache hierarchy than traditional workloads. Nevertheless, they reveal the need to take into account their behavior in the design of forthcoming CPUs.

3.5 Design proposals

Along with the state-of-the-art cache replacement policies presented in Section 2.2, we introduce MS-MPPPB and DS-MPPPB, two new LLC replacement policies derived from the original MPPPB. These two policies try, in distinct manners, to adapt themselves to the behavior of the emerging big data and HPC workloads.

3.5.1 Multi-Sampler Multiperspective

With MPPPB, Jiménez and Teran provide a replacement policy based on a reuse predictor, which ultimately relies on a hashed perceptron table.

Our first proposed design, named MS-MPPPB, is based on the idea that having not just one but many hashed perceptron tables can yield higher prediction accuracy and improved performance by dynamically choosing one of the perceptrons to trigger predictions. To perform the selection of the best behaving perceptron that will eventually trigger predictions, all perceptrons are competing against each other through a two-rounds decision tree scheme [57] that duels each of the four available perceptrons and selects the one that minimizes misses in the LLC. Although not used to produce a prediction, the three perceptrons left unused are concurrently updated following the process described by Jiménez and Teran in *Multiperspective reuse prediction* [55].

The additional hardware budget required for this proposal is rather high, as a naive implementation would lead to the instantiation of 4 individual samplers along with the 4 perceptron tables bound to them. Each block of the sampler holds an indices trace of the last accessed elements of the prediction tables, which requires a maximum of 128 bits. For each block, the sampler holds a 16-bit partial tag along with a 5-bit LRU state and a 9-bits confidence value. The sampler takes the form of a cache with 80 sets and 18 ways.

3.5.2 Multiperspective with Dynamic Features Selector

In the second design proposal, named DS-MPPPB, we use an additional concept along with the already existing idea of weights. Our new concept, *Coefficients*, revisits the conception of a hashed perceptron [94] by introducing the weights used in the mathematical definition of a perceptron [104].

We thus differentiate two key concepts. The *weights* are the actual values contained in the prediction tables of a hashed perceptron. These values are meant to reflect the learned reuse distance based on the observation of past events. These events can be the occurrence of a specific PC, a physical address or any other source of information used as feature[58, 6, 63]. The *coefficients* are confidence counters that reflect how accurate is the prediction table bound to a specific confidence counter.

The original code of MPPPB is shipped with not just one set of features but four, which adds up to a total of 64 features. Each of these sets of features was developed following the hill-climbing methodology described in *Multiperspective Reuse Prediction*[55] and is designed to fit each of the possible configurations of the CRC2 contest.

We gather all the features in a single set and build a perceptron predictor using them all. Although we now have a set of 64 features, we want to select only the 16 best behaving ones among the 64 available. To do so, for each prediction triggered by the replacement policy, the predictor searches for the 16 features with the highest confidence values and uses them to build a prediction, and the other features are left unused for that prediction. However, confidence values of all features are updated following algorithm 1.

For clarity, we include a summary of the notations we use. $\mathbb{F}$ denotes the set of features, n is the total number of features and m the number of features we include in the prediction value. We denote the confidence counter of the i -th feature as $conf(f_i) = conf_i$ along with the upper bound of the confidence counters $conf_{max,i}$. We denote as t_i the prediction table associated with feature f_i .

We thus define $\check{\mathbb{F}}$, the set of all possible arrangements of unique m features taken out of $\mathbb{F}$ and $\check{\mathbb{F}}_{max}$ the element of $\check{\mathbb{F}}$ that maximizes the sum of confidence counters. Finally, we compute the prediction value by summing the weights taken out of the tables of the elements of $\check{\mathbb{F}}_{max}$.

Table 3.1 summarizes the hardware budget of each design proposal described in this section. Along with the total hardware budget required for each proposal, we also provide the budget required by each component, namely: the replacement states (here we use MDPP, a modified Tree-based PLRU[54, 95] policy that uses a custom transition vector to determine to which position an accessed block should be moved to), the sampler(s) and perceptron(s).

Algorithm 1 Updating confidence values of prediction tables

```

hit ← false
truth ← 0ℬn {A n-vector of falses.}
pred ∈ −32;31n {A vector of individual predictions.}
if Accessed block hits in the sampler then
    hit ← false
else
    hit ← true
end if
for all  $i$  such that  $0 \leq i \leq n - 1$  do
    truth[i] ← ((pred[i] < 0) = hit)
    if truth[i] = true then
        confi ← max(confi + 1, confmax,i)
    else
        confi ← min(confi − 1, 0)
    end if
end for

```

	Replacement states	Sampler(s)	Percpetron(s)	Total
MS-MPPPB	3.75	111.09	12	126.84
DS-MPPPB	3.75	95.27	12	111.02

Table 3.1 Hardware budget of the proposed designs.

3.6 Methodology

In this section we present the evaluation methodology used to report results in Section 3.7. In particular, the next subsections present the set of workloads used to evaluate the different LLC replacement policies and our workload selection methodology, a description of the simulation environment, and the evaluated replacement policies and their configuration.

Overall, we follow the same evaluation methodology as the one used by Shi *et al.* [87] with the aim of building the fairest comparison possible against state-of-the-art techniques.

3.6.1 Workloads

For the evaluation of the different LLC replacement policies we consider the following sets of workloads:

- Over 2000 Qualcomm workloads used for CVP1 contest.
- All SPEC CPU 2006 and CPU 2017 benchmarks.

- All workloads included in the GAP Benchmark Suite.
- All workloads included in the XSBench Suite.

From all these benchmarks we select the 50 most intensive workloads so that our evaluation set of workloads is a blend of each suite designated above. We use the SimPoints [79] methodology to identify intervals (hereafter *SimPoints*) representative of at least 5 % of the SPEC, GAP and XSBench workloads. Each SimPoint is 1 billion instructions long and characterizes a different phase of these workloads. Each SimPoint is executed for 200 million instructions in order to warm-up the memory hierarchy, and then it is executed for an additional 1 billion instructions to report experimental results.

We only evaluate these workloads in a single-thread context. We deliberately chose to restrict our evaluation to single-core as this work focuses on the characterization of the access patterns of the selected workloads to the LLC. The modeled architecture being composed of a shared LLC, modeling an architecture using multiple cores we would not be able to properly measure reuse distances as the different cores would be asking for distinct data in the same cache, thus compromising our measurements. The results reported per benchmark (for SPEC, GAP and XSBench) are the normalized weighted averages of the results for individual SimPoints. In contrast, the Qualcomm workloads are single-trace benchmarks that do not use such methodology.

3.6.2 Experimental setup

Our evaluation considers ChampSim[30], a detailed trace-based simulator that models an out-of-order CPU along with its cache hierarchy, prefetching mechanisms and memory sub-system. Table 4.1 provides a detailed configuration of the modeled CPU and the memory hierarchy.

3.6.3 Replacement policies simulated

We evaluate the workloads described in Section 3.2 against the most relevant cache replacement policies proposed in the literature: SRRIP, DRRIP, SHiP, MPPPB, Hawkeye and Glider, all detailed in Section 2.2. Although there is a vast amount of work in reuse prediction available in the literature [1, 17, 33, 115, 47, 54, 58, 60, 95, 96], the aforementioned replacement policies that have been selected for the evaluation are the most recent and relevant approaches in the state-of-the-art. In addition, in the evaluation we also include the two new replacement policies proposed in this chapter, explained in Section 4.3. These two new techniques are derived from MPPPB and leverage the usage of multiple perceptrons

Component	Description
Branch Predictor	hashed perceptron
CPU	4 GHz, 4-wide out-of-order processor 6-stage pipeline, 128-entries re-order buffer
L1 ITLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle latency, 16-entry MSHR
L1-I Cache	32, 8-way, 4-cycle latency, 8-entry MSHR
L1-D Cache	32, 8-way, 4-cycle latency, 8-entry MSHR next line prefetcher
L2 Cache	256, 8-way, 12-cycle latency, 16-entry MSHR ip-stride prefetcher
LLC	2, 16-way, 26-cycle latency, 32-entry MSHR
DRAM	4, DDR4 SDRAM data-rate: 3.2 GT/s, I/O bus frequency: 1.6 GHz $t_{RP} = t_{RCD} = t_{CAS} = 24$ cycles

Table 3.2 System simulation parameters.

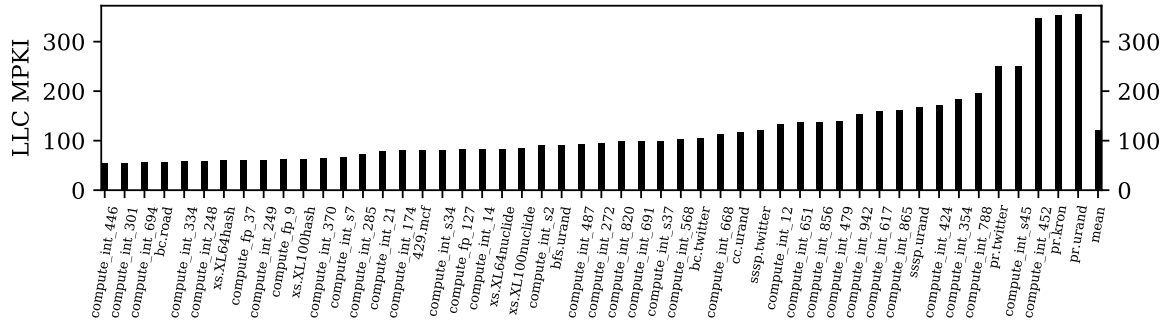
to achieve higher accuracy. For MPPPB we used the code that is publicly available on the website of the CRC2 contest [41]. For Glider we use code graciously provided by the authors.

3.7 Results, analysis and discussion

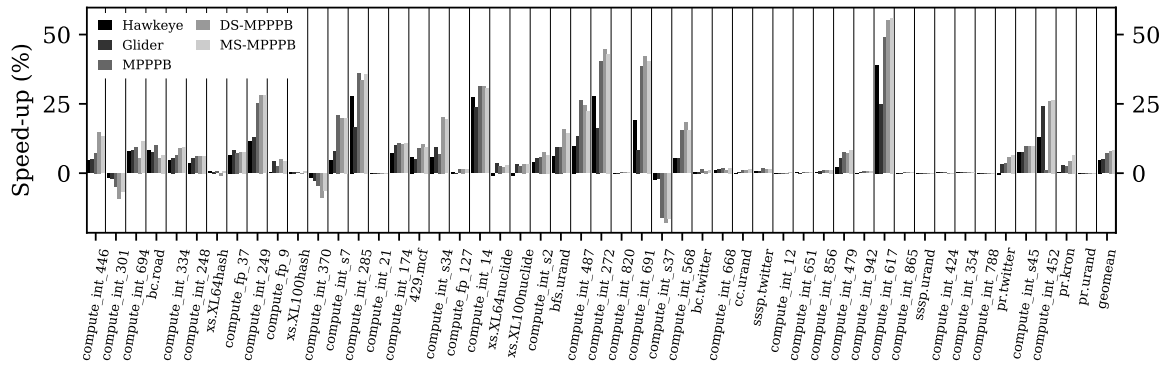
This section presents our experimental campaign along with the results obtained and the characterization of the studied workloads. Section 3.7.2 presents the performance benefits yielded by the different state of the art cache replacement policies mentioned in Sections 3.6.3 and 4.3. Section 3.7.1 studies the impact of these workloads on the LLC in terms of misses and presents the MPKI reduction obtained by the replacement policies mentioned above. Finally, Section 3.7.3 studies the behavior of the studied workloads via the reuse distance of the cache blocks to highlight the different behavior of these benchmark suites.

3.7.1 Misses

Figure 3.2a shows the LLC MPKI of the 50 most intensive workloads using the LRU replacement polity These workloads show a very high impact on the LLC, with an average MPKI of 120, much larger to the workloads used in previous work. We use this to define the set of workloads that will be used in Section 3.7.2 to evaluate state-of-the-art replacement policies along with our custom designs. These results clearly show the high impact on the LLC of Qualcomm, GAP and XSBench workloads. This selection of benchmarks is



(a) LLC MPKI using the LRU replacement policy for the 50 most intensive workloads.



(b) Speed-up over LRU of the state-of-the-art LLC replacement policies and our custom replacement policies for the 50 most intensive workloads.

Figure 3.2 LLC MPKI using the LRU replacement policy and performance improvement of LLC replacement policies for the 50 most intensive workloads.

comprised of each of the studied benchmark suites and allows to evaluate replacement policies against a broad range of workloads with different behaviors.

3.7.2 Performance

Figure 3.2b shows the speed-up of various replacement policies presented in Section 3.6.3 and in Section 4.3. LLC MPKI sorts the benchmarks with a baseline LRU policy. While Figure 3.1b was showing Glider standing out against MPPPB in some situations, Figure 3.2b clearly shows the versatility of the former and its ability to consistently deliver good performance even facing the most intensive and hard to predict workloads. MPPPB and Glider provide respectively 7.0 % and 5.0 % speed-up over baseline LRU. Thus, Glider is not able to deliver significant improvement over the baseline LRU compared with MPPPB. This fact suggests that the machine learning algorithm used to design the predictor of Glider should be trained against a wider variety of workloads, hence the need to include the workloads used in this chapter in further work on LLC replacement policies.

Besides, we justify the difference of performance improvement in Figures 3.1b and 3.2 between Glider and MPPPB compared to the results published in *Applying Deep Learning to the Cache Replacement Problem*[87] by a difference in the methodology. As a matter of fact, we use the SimPoint methodology to generate at most ten simpoints and we only use the ones accounting for more than 5 % of the whole execution, whereas the original results of Glider were reported with only a single trace per benchmark. We argue that our methodology is more robust as we cover more distinct behaviors, thus challenging the different techniques studied.

We also report significant performance improvements for our proposed techniques derived from MPPPB. The two designs proposed in this chapter, MS-MPPPB and DS-MPPPB, respectively yield 8.3 % and 8.0 % speed-up over the baseline LRU, and both new approaches also outperform MPPPB. We observe that this performance improvement is due to the ability to adapt the set of features used for prediction in a execution phase-wise manner, providing more versatility to the design than the previously discussed techniques.

3.7.3 Reuse distances

Figure 3.4 shows the distribution of reuse distances for each benchmark suite used in this work. Figure 3.4 is organized as a box plot; the horizontal line of each distribution representing its median, the box ranging from the first to the third quartiles and the whiskers representing extreme values through the first and ninth deciles of the distributions. For readability purposes, we cut the box plot to only show whiskers but not flyers. For instance, the distribution of reuse distances of the GAP workloads have ninth decile around 2000 and a median of 116. We complete figure 3.4 with Table 3.3 as it provides additional characteristics of the distribution such as maximum, median, mean and standard deviation of the observed reuse distances.

Parameter (in accesses)	SPEC	XSbench all	XSbench unionized	XSbench others	Qualcomm	GAP all	GAP kron & urand	GAP others
Maximum	21076	12677	2248	12677	59934	34836	33771	34836
Median	10	61	84	55	4	116	45	27
Mean	126.23	343.23	193.21	385.36	93.76	907.89	880.88	349.25
Standard deviation	572.94	875.68	256.33	977.54	541.80	2106.41	1885.68	1050.23
Avg. proportion of dead block accesses	48.03 %	37.32 %	46.63 %	28.01 %	10.54 %	55.84 %	59.31 %	54.06 %

Table 3.3 Main parameters of the distribution of reuse distances for cache blocks in the different benchmakrs.

In this work, we define reuse distance as the number of accesses to different cache blocks between two consecutive accesses coming from the CPU (*e.g.* read and writes but not prefetches and write-backs) to the same cache block. Looking at this figure, it is quite clear that all the benchmark suites suffer from the presence of dead blocks in their access

patterns. However, having a closer look at each of the distribution presented can provide us with valuable knowledge and give intuition about the results obtained in Sections 3.7.2 and 3.7.1. Blocks that do not experience a single reuse during the whole execution of a workload are not represented on Figure 3.4. However, in order to still provide that valuable information, we show in Table 3.3 the average proportion of accesses to dead blocks during the execution.

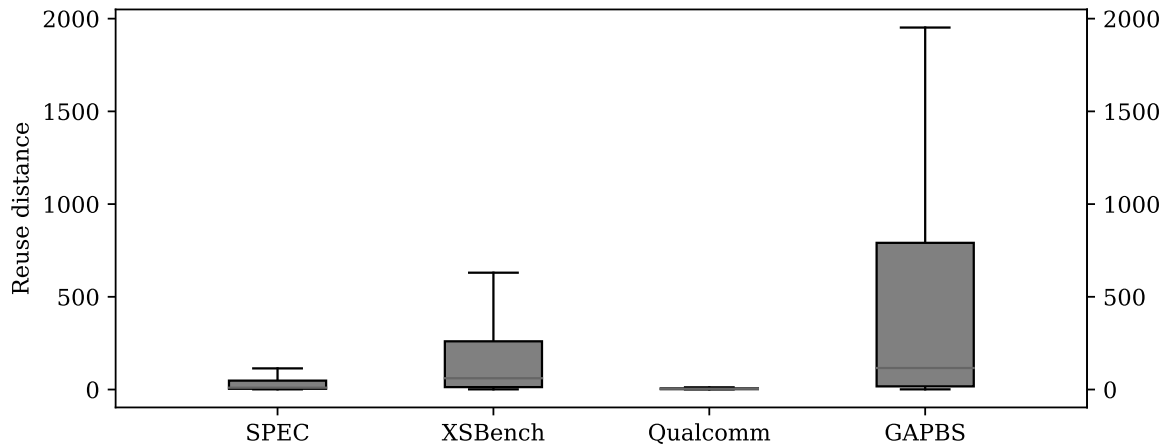


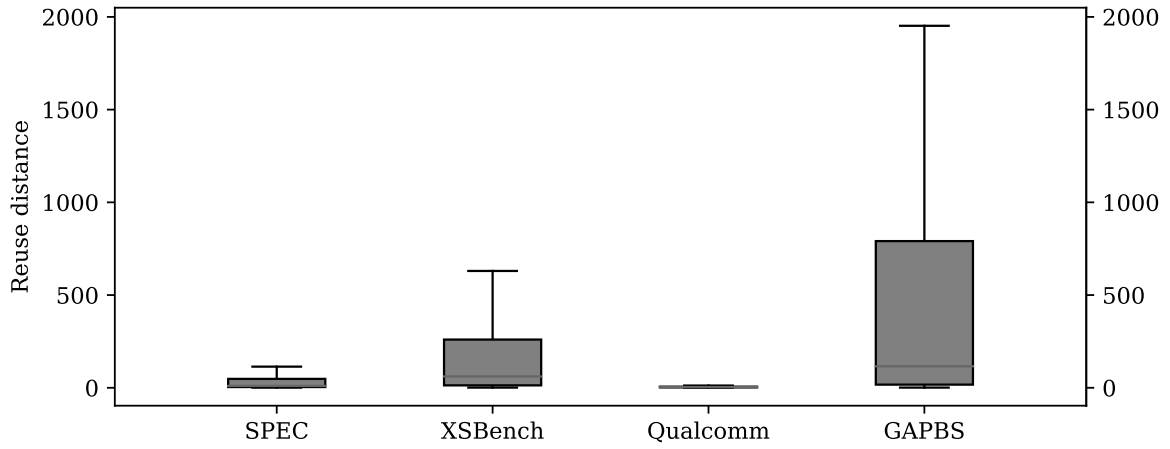
Figure 3.3 Distribution of the reuse distances of the cache blocks per benchmark suite.

For the rest of this section, we define as *cache-friendly blocks* those blocks with a reuse distance lower than the associativity of the LLC. Conversely, we define *cache-averse blocks* as blocks with a reuse distance higher than the LLC associativity. Assuming an LRU policy for the cache, if a cache block A gets inserted in the LLC and the cache sees 16 accesses to different blocks before re-referencing block A, the second access to block A will cause a miss because it will have been evicted. This block A occupies part of the cache capacity that can be used to allocate another block, hopefully cache-friendly, and can provide a hit instead of miss.

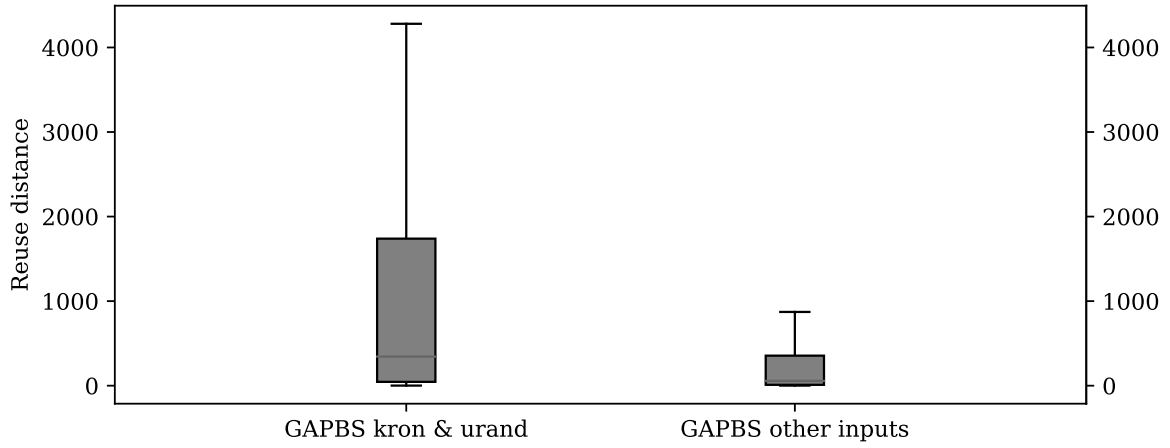
3.7.3.1 SPEC benchmarks

As Figure 3.4a and Table 3.3 show, the SPEC workloads present an average reuse distance of 126 and a standard deviation as low as 572.94, so the aggregated distribution of reuse distances for these workloads focuses mainly on low values with a relatively low standard deviation which translates into a cache-friendly behavior. Also, most accesses being cache-friendly, a bypass policy is not required to achieve reasonable performance over LRU.

This behavior explains why policies such as Hawkeye and Glider perform well when applied to SPEC workloads. These policies are focusing on prioritizing eviction for blocks that previously showed cache-averse behavior. Doing so allows to free space for cache-friendly blocks, however, as we will see in details for GAP benchmarks and XSBench



(a) Distribution of the reuse distances of the cache blocks for the different benchmark suites.



(b) Distribution of the reuse distances of the cache blocks for the GAP benchmark suite with different input sets.

Figure 3.4 Distribution of the reuse distances of the cache blocks for the different benchmark suites.

benchmarks in the next subsections, replacement policies in general are less effective when the memory footprint increases and the access patterns become unpredictable.

3.7.3.2 XSBench benchmarks

The *XSBench* benchmark suite, as expected based on the reuse distance characteristics established in Table 3.3, shows a very distinct behavior from the well-known *SPEC* CPU 2006 and *SPEC* CPU 2017 workloads. As a matter of fact, the *XSBench* workloads present an average reuse distance of 343.23 and a standard deviation as high as 875.68, which reveals a hard to predict behavior. However, they do experience less dead accesses to the LLC, the proportion of dead block accesses being 37.32 %.

With both mean and median reuse distances being the double of the *SPEC* ones, it is clear that these workloads are much more biased towards a cache-averse behavior than the *SPEC* workloads. However, with such high distribution parameters, we can still make a crucial observation when it comes to reuse prediction and dead blocks. We observe that, even though the *XSbench* workloads experience reasonable reuse of cache blocks that are accessed more than once during the execution, this happens for all the grid types except the *unionized*, which has a substantially larger memory footprint than the other grid types. We explain such behavior by the vast amount of data that the solver needs to traverse during the workloads execution and the algorithms used to do so.

3.7.3.3 Qualcomm benchmarks

When it comes to Qualcomm workloads, we observe that the distribution of reuse distances stretches towards higher values (Table 3.3 shows that the tail of the distribution goes as high as 60000 accesses). However, these workloads are relatively biased towards a cache-friendly behavior as the standard deviation is 541.80 and the dead block accesses proportion is of 10.54 %. This stretched shape leads to a rather low standard deviation and to the appearance of low reuse distances with higher probability than of high reuses distances. We understand the long-wide tail of the distribution as a low amount of dead blocks thrashing the LLC, using space that would be a better fit for cache-friendly blocks.

Thus, this observation gives us intuition about the behavior of Glider on the Qualcomm benchmarks presented in Figures 3.2a, 3.2b, and 3.1b. Glider is a replacement policy that does not implement bypass, so it keeps on inserting dead blocks in the cache even though it has learned that these blocks are cache-averse. The absence of a bypass policy explains the poor results of Glider compared to MPPPB and the two proposed approaches derived from it.

3.7.3.4 GAP benchmarks

Based on the characteristics of the reuse distance distribution shown in Table 3.3 and Figure 3.4 for the GAP benchmark suite, these graph processing workloads show a very cache-averse behavior with an average reuse distance of 907.89, which is around 5.8 times higher than the one of *SPEC* workloads, and a proportion of dead block accesses of 55.84 %. Such behavior is expected as GAP workloads are executing graph processing algorithms and are known to traverse vast amount of data in a very unpredictable way. Table 3.3 shows a median of 28 accesses, which provides us with useful information as 50 % of the blocks accessed experience a reuse distance of 28 or fewer accesses. This characteristic suggests that having higher LLC associativity could help to achieve higher performance.

In addition, these benchmarks show very high standard deviation on reuse distances, which suggests that the CPU triggers accesses to both cache-averse and cache-friendly blocks with relatively uniform probabilities, which results in the eviction of useful blocks. Thus, we deduce that using a bypass policy like MPPPB does would provide performance benefits. Results shown for GAP benchmarks on Figure 3.2 highlight this property of graph processing workloads.

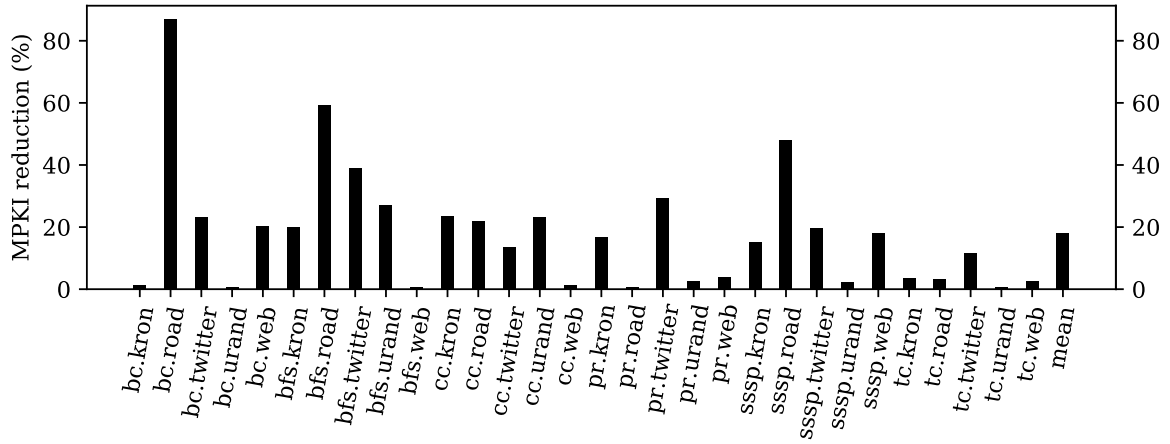
As we observe that GAP benchmarks with inputs *kron* and *urand* show dramatically low reuse of cache blocks during execution, we categorize GAP benchmarks in two categories: (i) GAP workloads using *kron* and *urand* inputs; (ii) GAP workloads using other inputs.

As figure 3.4 shows, workloads using *kron* and *urand* inputs tend to stress more the LLC. As stated by Beamer *et al.* in the specification of the benchmark, these inputs present a very high memory footprint compared to the other inputs used in this work and, moreover, their graphs have a very distinct topology. *Urands* represents a worst-case as every vertex has an equal probability of being a neighbour of every other vertex. Thus, the graph processing kernel in charge of traversing these inputs has to request memory blocks very distant from another, which eventually causes poor reuse.

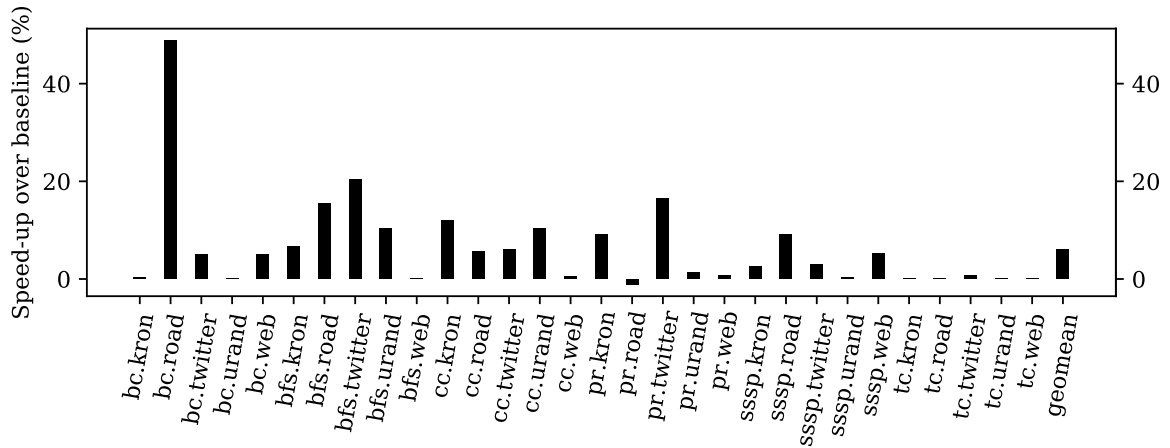
As a result, we see that LLC replacement policies are impractical solutions in this context as these workloads are biased towards a cache-averse behavior, all accesses being cache-averse. Although replacement policies cannot improve performance for these workloads, an excellent way to cope with these workloads would be to increase the capacity of the cache substantially or to incorporate a prefetcher that can predict the access patterns to the memory blocks and bring them to the cache before they are accessed.

To demonstrate the need for increased cache capacity, we increase the size of the LLC from 2 to 16 and present the results in Figure 3.5. Results show a clear benefit from the increased capacity of the LLC. The average MPKI over the whole set of GAP benchmarks drops by 18 %, which delivers a geometric mean speedup of 6.1 % compared with a 2 LLC with an LRU replacement policy. However, some workloads show reduced MPKI while suffering from an IPC slow-down. The reason for this this phenomenon are the DRAM latencies. While increasing the LLC capacity, these workloads have similar statistics in terms of cache accesses and miss rates, but the miss rate in the DRAM row buffers increases. We interpret this behavior as a symptom of workloads with extremely poor temporal and spatial locality.

This clearly shows the need for improved memory allocation policies when it comes to graph processing algorithms. To illustrate this example, we focus on *pr . road*, as Page Rank is an algorithm that should show better locality than the others. When computing the score of an edge of the graph, the algorithm only visits neighbouring edges to that edge. Thus, the



(a) LLC MPKI reduction of 16 LLC over the baseline 2 LLC.



(b) Speed-up of 16 LLC over the baseline 2 LLC.

Figure 3.5 LLC MPKI reduction and speedup of a 16 LLC over the baseline 2 LLC.

only way for the DRAM to exhibit higher row buffer miss rate is that neighbouring edges in graphs are stored in very distant places in main memory. This showcases the need for a memory allocation policy that takes into account the topology of the graph. Another way to improve the performance of these graph processing workloads would be to incorporate a dedicated on-chip storage structure able to serve, at a low cost, not only the required edges but also their neighbours.

To wrap up this discussion about GAP benchmark suite, the presented results show that there is still a good amount of work to be done on the algorithmic and software side of these workloads to make them cache-friendly. In addition, the reported characteristics of this benchmarks are hard to exploit by traditional cache hierarchies and cache replacement policies, so different solutions should be explored on the hardware side to improve the performance of these benchmarks.

3.8 Summary

In this research paper, an extensive examination is conducted to assess the performance of a combination of graph processing, scientific computing, and industrial workloads. The evaluation focuses on the utilization of cutting-edge Last-Level Cache (LLC) replacement policies across a diverse range of benchmark scenarios.

The primary objective of this study is to understand how these replacement policies impact overall system behavior and performance in the context of specific workloads. To achieve this, a variety of benchmarks are employed, each representing a different use case and workload type. This diverse set of benchmarks is essential to capture the nuances of how these replacement policies function across various application domains.

The results obtained from this comprehensive evaluation reveal some noteworthy insights. Firstly, it becomes evident that the replacement policies under scrutiny yield substantial performance enhancements when applied to workloads representative of the SPEC benchmarks. SPEC workloads are typically representative of standard, industry-specific applications and, therefore, have specific cache access patterns. The replacement policies demonstrate their effectiveness in improving cache management for these workloads.

However, the story is different when considering High-Performance Computing (HPC) and big-data workloads. For these types of workloads, the evaluated replacement policies do not exhibit the same level of effectiveness in enhancing system performance. This finding is significant as it underscores the importance of tailoring cache management strategies to the unique characteristics of the workloads being executed.

One noteworthy observation from the benchmark results is the high standard deviation in reused distances. This variation suggests that the existing replacement policies often result in the eviction of blocks that could have been beneficially retained in the cache. This inefficiency underscores the need for more optimized cache management techniques.

In response to these findings, two innovative techniques, DS-MPPPB and MS-MPPPB, are proposed in the paper. These techniques are specifically designed to address the shortcomings of existing approaches for big data and HPC workloads. The results of the evaluation reveal that DS-MPPPB and MS-MPPPB outperform state-of-the-art methods in these specific contexts, delivering superior performance benefits.

In conclusion, this research paper offers a comprehensive evaluation of LLC replacement policies across a wide range of workloads. It highlights the nuanced nature of cache management, where the efficacy of policies depends on the specific characteristics of the workloads they are applied to. Additionally, the introduction of DS-MPPPB and MS-MPPPB presents promising solutions to the challenges posed by big data and HPC workloads, demonstrating the importance of continuous innovation in cache management techniques.

CHAPTER

4

PRACTICALLY TACKLING MEMORY BOTTLENECKS OF GRAPH-PROCESSING WORKLOADS

Lorsque la séparation est inévitable, il faut avoir le courage de laisser partir ceux que l'on aime pour leur permettre de trouver leur propre chemin vers le bonheur.

Honoré de Balzac

4.1 Introduction

IN recent years, graph-processing has become an important class of workloads with applications in a rapidly growing and diverse number of fields (*e.g.*, network analysis [35], bioinformatics [44], path-planning [24], and machine learning [25]). Graph-processing workloads typically use very large input sets, often in multi-gigabyte scale, and data-dependent graph traversal methods [72] that exhibit highly irregular memory access patterns. Processing such massive and irregularly accessed data prevents graph-processing applications from exhibiting good locality, imposing great difficulty on traditional cache hierarchies to efficiently serve memory requests, leading to frequent main memory accesses that incur high latency overheads and compromise application performance. Indeed, recent work [113] demonstrates that, due to the highly irregular memory access patterns of data-depend graph traversals, state-of-the-art graph-processing workloads spend up to 80 % of the total execution time waiting for memory accesses to be served by the DRAM.

Prior work has quantified the overheads of graph-processing workloads [69] and has proposed several approaches to ameliorate their costs. These approaches mainly fall into the following categories: (i) domain-specialized cache management for graph-processing applications [81, 10]; (ii) indirect memory prefetching [110, 4, 13], which aims at improving the throughput of memory-bound graph-processing workloads by fetching data blocks before they are explicitly requested by the application; (iii) graph-processing accelerators [75, 74, 73, 32, 43, 69]; and (iv) pre-processing schemes that improve the locality of graph-processing workloads [102, 31, 8]. Our proposal is in the first category, which applies fine-grained micro-architectural cache management optimizations, avoiding costly pre-processing of the graph data without requiring any changes to the architecture nor to any software layer.

Two key observations drive our proposal: First, conventional cache hierarchies become mostly useless when dealing with graph-processing workloads, since 78.6% of the accesses that miss in the L1 Data Cache (L1D) result in misses in the L2 Cache (L2C) and in the Last Level Cache (LLC), thus requiring a DRAM access. This is caused by the poor locality of part of the data set of graph-processing applications that is accessed with highly irregular memory access patterns. The second observation is that strides between accesses triggered by instructions featuring the same Program Counter (PC) are a good program feature to predict whether a certain memory access will be served by DRAM or not. Our key insight is that bypassing the L2C and the LLC for highly irregular accesses significantly reduces their latency cost and, at the same time, it lowers the pressure on the lower levels of the cache hierarchy. This, in turn, minimizes cache pollution and increases the effective capacity of the L2C and the LLC for the subset of data that exhibits good locality and that is accessed by regular accesses patterns.

To exploit the above explained key insights and overcome the limitations of current cache hierarchies, this work proposes the Large Predictor (LP), a novel and low-cost micro-architectural predictor capable of dynamically identifying regular and irregular access patterns. To do so, LP employs a small prediction table, indexed with the PC, that provides a historic knowledge of the strides of the memory accesses. LP leverages this information to classify a memory access as regular or irregular; those classified as regular are routed to the normal cache hierarchy while the others go through the Side Data Cache (SDC), a new per-core private auxiliary cache. The SDC is a small set-associative cache placed alongside the L1D with the specific purpose of reducing the latency of highly irregular accesses while avoiding polluting the rest of the cache hierarchy with data that exhibits poor locality.

This chapter makes the following contributions:

- We corroborate that memory requests of graph-processing workloads that miss in the L1D also miss in the L2C and the LLC with high probability. The key insight of this study is that bypassing the L2C and LLC for irregular accesses that do not exhibit locality has potential to provide significant benefits.
- We propose and design the Side Data Cache (SDC), a small on-chip cache placed alongside the L1D, and the Large Predictor (LP), a novel micro-architectural predictor that dynamically classifies the memory requests into regular and irregular. The accesses identified by LP as regular and irregular are routed to L1D and to the SDC, respectively. Our proposal improves the performance of graph-processing applications by removing the useless L2C and LLC look-ups for the irregular accesses while also reducing cache pollution and improving the locality in the rest of the cache hierarchy.
- We demonstrate that, across a set of contemporary and diverse graph-processing workloads, our proposal outperforms conventional cache hierarchies as well as the Transpose-based Cache Replacement (T-OPT) [10], the state-of-the-art cache management scheme for graph-processing workloads, achieving respective geometric mean speedups of 20.3% and 10.9% in single-core scenarios, and of 20.2 % and 13.8% on a multi-core setup. Furthermore, T-OPT (and its practical but less performant implementation, P-OPT) requires modifying the original application, whereas our proposal does not. Regarding the hardware budget, SDC coupled with LP requires 10KB of storage per core.

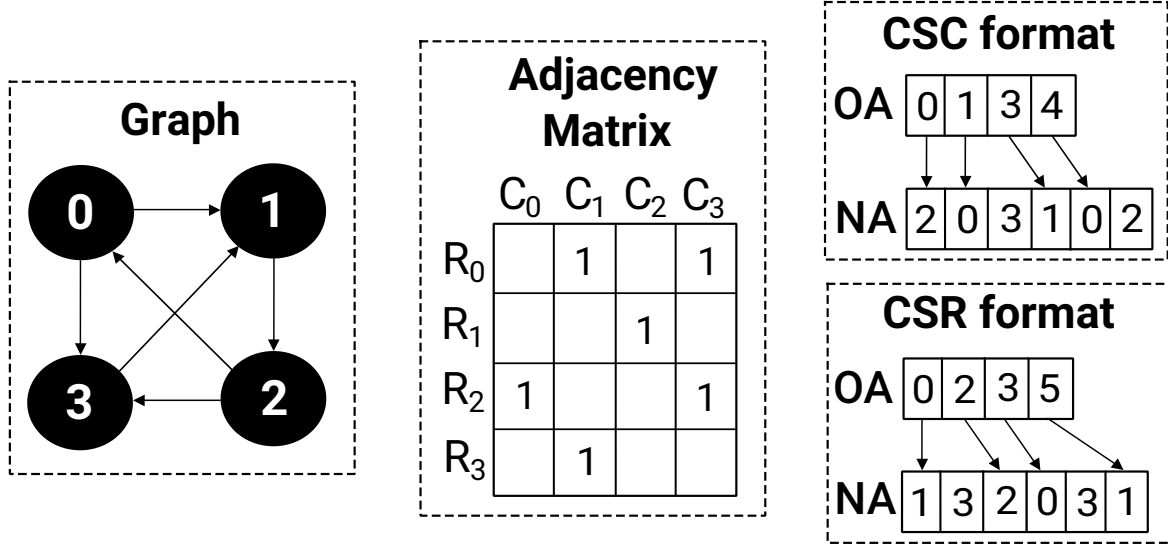


Figure 4.1 Graph representations using the CSR/CSC format.

4.2 Background and Motivation

4.2.1 Graph-Processing Workloads

Graph-processing is becoming a fundamental tool in a wide range of areas including bioinformatics [44], social network analysis [35], and web analytics [24]. Graph-processing workloads typically use sparse data formats like the *Compressed Sparse Row/Column* (CSR/CSC) [42], or even more sophisticated proposals [62], to manage large amounts of graph data. The CSR/CSC format is used to encode the graph adjacency matrix using several data structures. The first one contains the row or the columns indices of the adjacency matrix for the cases of CSR and CSC, respectively, and it is typically called the *Neighbors Array* (NA). The second data structure indexes the beginning of each adjacency matrix row for the case of CSR (column for CSC), and it is denoted as the *Offset Array* (OA). Figure 4.1 represents both the OA and the NA arrays corresponding to the left hand side graph for both the CSR and CSC formats. Finally, there are additional data structures that contain numerical data corresponding to graphs vertices that we call *Property Arrays*. In the context of graph-processing, the CSR format encodes outgoing neighbors while CSC contains incoming ones.

Manipulating these sparse data structures often produces irregular memory access patterns. For example, when computing the Sparse Matrix-Vector (SpMV) multiplication $y = Ax$, accesses to vector x are indexed by the column indices of matrix A , which are non-contiguous and constitute an irregular access stream. Graph-processing workloads also display highly irregular memory access patterns driven by operations like graph traversals that require

visiting all the nodes of graph V , that is, scanning adjacency matrix rows following the graph connectivity. For a certain vertex v_i , its neighbors correspond to the non-zero elements of adjacency matrix row i and define the next rows of the adjacency matrix to be accessed, producing an irregular memory access stream driven by the graph connectivity.

Algorithm 2 shows one of the most important graph-processing workloads, *Page Rank* [24]. Algorithm 2 shows the two main steps of *Page Rank* per each main loop iteration. First, for each vertex v_u , the algorithm updates the content of the *outgoing_contrib* array using the scores obtained in the previous iteration divided by the number of outgoing neighbors of v_u , $d^+(u)$. Then, the algorithm computes the *Page Rank* score of each vertex v_u as the sum of the *outgoing_contrib* and a constant value that depends on the index count, $\#V$, and the damping factor δ . This process is repeated until either convergence or the maximum number of iterations is reached.

The code region that produces irregular accesses is composed of a loop that ranges from line 7 to 15. The purpose of this loop is to traverse the entire graph by iterating over the incoming neighbors of a given vertex v_u , which are located between positions $OA[u]$ and $OA[u + 1] - 1$ of the *NA* array. The algorithm uses these indices to access the *outgoing_contrib* property array, thus, the access pattern to *outgoing_contrib* is irregular and driven by the matrix connectivity.

4.2.2 Characterizing the Memory Hierarchy Behavior of Graph-Processing Workloads

Irregular memory access patterns driven by graph connectivity have a strong impact on the memory hierarchy behavior of graph-processing workloads. Figure 4.2 shows the Misses-Per-Kilo-Instruction (MPKI) rates experienced by graph-processing workloads belonging to the GAP benchmark suite [15]. We show MPKI rates concerning three cache hierarchy levels (L1D, L2C, and LLC). Section 5.5 describes in detail our experimental setup. Figure 4.2 shows that graph-processing workloads suffer from a large number of misses on all the levels of the cache hierarchy. The average MPKI rates of graph workloads for the L1D, the L2C, and the LLC are, respectively, 53.2, 44.5, and 41.8.

Finding 1. *Graph-processing workloads exhibit very high MPKI rates in all cache levels.*

In addition, Figure 4.2 shows that L2C and LLC MPKI rates are just slightly smaller than L1D MPKI, which indicates that just a minor portion of L1D misses are served either by the L2C and LLC. Our analysis indicates that 78.6% of the access that miss in the L1D also miss in the lower cache levels, so they require a DRAM access.

Algorithm 2 Computation of Page Rank scores

Require: $G(V, OA, NA)$ { OA and NA respectively denote the offset array and the neighbors array in the CSC format.}

Require: δ {Damping factor.}

Require: ε {Convergence threshold.}

Require: $iterations$ {Maximum # of iterations to process.}

Ensure: $scores[:]$ {Page Rank scores for all vertices}

```

1:  $scores[:] \leftarrow \frac{1}{\#V}$ 
2: for  $iter \leftarrow [0; iterations - 1]$  do
3:    $error \leftarrow 0$ 
4:   for all vertex  $v_u \in V$  do
5:      $outgoing\_contrib[u] \leftarrow \frac{scores[u]}{d^+(u)}$ 
6:   end for
7:   for all vertex  $v_u \in V$  do
8:      $sum \leftarrow 0$ 
9:     for  $i \leftarrow [OA[u], OA[u+1] - 1]$  do
10:       $sum \leftarrow sum + outgoing\_contrib[NA[i]]$ 
11:    end for
12:     $old\_score \leftarrow scores[u]$ 
13:     $scores[u] \leftarrow \frac{1-\delta}{\#V} + \delta \cdot sum$ 
14:     $error \leftarrow error + |scores[u] - old\_score|$ 
15:  end for
16:  if  $error < \varepsilon$  then
17:    Convergence has been reached.
18:  end if
19: end for

```

Finding 2. A very large portion (78.6%) of the accesses that trigger L1D misses also miss in the lower-levels of the cache hierarchy and require a DRAM access.

Figure 4.3 shows a characterization of the memory accesses triggered by graph-processing workloads in terms of strides between accesses issued by instructions with the same PC. The x-axis displays different intervals of strides. The y-axis shows the percentage of memory accesses that are served at DRAM per each interval. We obtain data on Figure 4.3 by considering the Connected Components application [15] of the GAP suite applied to the Friendster graph[108]. Section 5.5 describes in detail our experimental setup. Figure 4.3 indicates that memory accesses featuring small strides have a much lower probability of accessing DRAM than access displaying large strides. For example, just 11.6% of memory accesses whose strides fall between 2 and 10 (*i. e.* interval $(10^0$ to $10^1]$ of Figure 4.3) access DRAM, while this percentage grows to 97.6% for accesses with strides between $10^5 + 1$ and 10^6 . We analyze all applications of the GAP benchmark suite and observe similar behaviour.

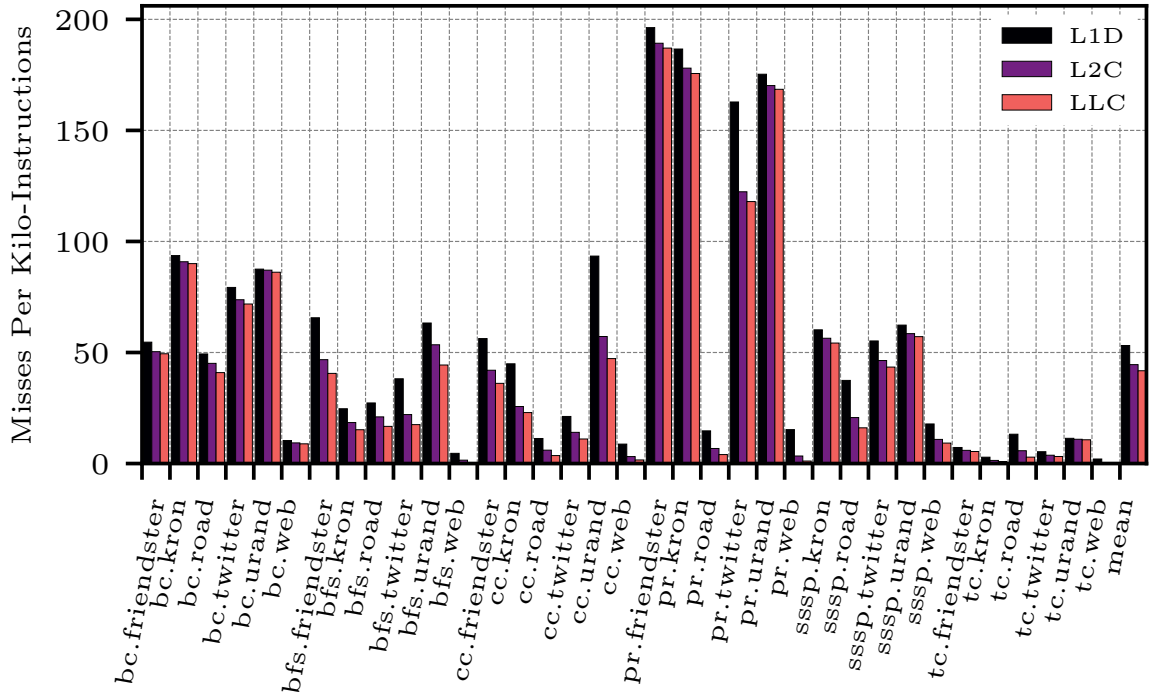


Figure 4.2 Misses-Per-Kilo-Instruction (MPKI) across the different levels of the cache hierarchy triggered by graph-processing workloads.

Finding 3. *Memory accesses featuring large strides have a very high probability of accessing DRAM.*

These three findings imply that memory accesses of graph-processing workloads can be divided into two categories: i) accesses displaying a large stride with respect to the last access triggered by an instruction with the same PC, that have a very high probability of missing on all cache levels and accessing DRAM; and ii) accesses featuring a small stride, that are in general served by the cache hierarchy. It is thus natural to propose hardware mechanisms able to differentiate between these two categories and avoid cache averse accesses to pollute cache content. These mechanisms strongly reduce the latency of both cache averse accesses, because they avoid useless cache look-ups, and cache friendly accesses, because cache lines holding data to serve them are not evicted by insertions of cache averse data.

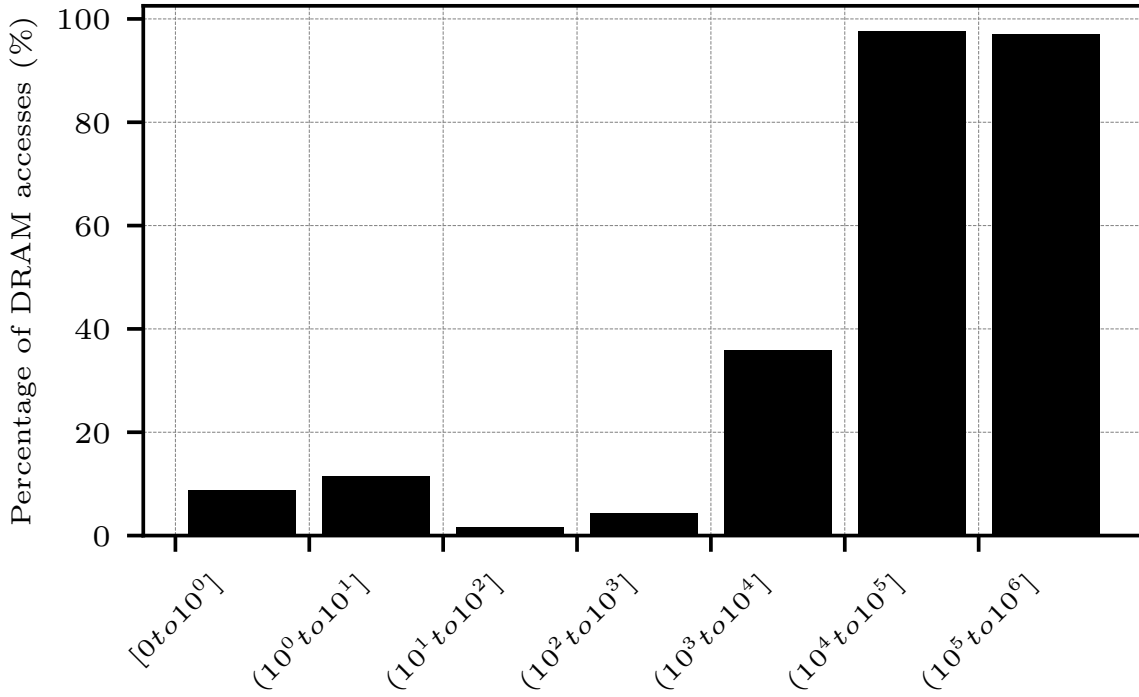


Figure 4.3 Probability of accessing DRAM for accesses exhibiting different strides. Data corresponds to the *cc.friendster* workload.

4.3 Managing Memory Accesses of Graph-Processing Workloads

To tackle the pervasive irregular access patterns experienced by the different elements of the memory sub-system when running graph-processing workloads, we propose two new hardware structures: The Side Data Cache (SDC), a small data cache placed alongside the L1D cache of each core, and the Large Predictor (LP), a low-cost micro-architectural prediction scheme that dynamically classifies memory access as cache friendly or averse. Memory accesses classified as cache averse by the LP are routed to the SDC, which (i) avoids the costly look-ups on the cache hierarchy (L1D, L2C, LLC) for cache averse accesses, thus reducing their latency cost; and (ii) does not pollute the on-chip caches for cache averse accesses, thus improving their cache locality.

4.3.1 Side Data Cache (SDC)

The Side Data Cache (SDC) is a small on-chip cache placed alongside the L1D cache of every core. The goal of the SDC is to provide an alternative and fast data path for cache averse

memory accesses. As Section 5.3 indicates, 78.6% of the accesses that miss in the L1D also miss in the lower levels of the cache hierarchy and end up accessing DRAM, and most of these are caused by cache averse accesses that exhibit large strides. To prevent performance inefficiencies when handling cache averse accesses, these are routed to the SDC instead of to the traditional cache hierarchy. If the look-up in the SDC hits, the requested cache block is returned to the CPU with low latency, given that the SDC is smaller than a conventional L1D. In case the look-up in the SDC misses, a coherence message is sent to the cache directory to ensure coherence (as described later in detail in Section 4.3.3) and, if the data is not present in the rest of the cache hierarchy, the requested block is fetched from DRAM and directly inserted in the SDC, bypassing the L2C and the LLC. This provides a fast path to DRAM, avoiding costly accesses to the lower-level caches (L2C, LLC). In addition, cache pollution is reduced in all the levels of the cache hierarchy (L1D, L2C, LLC), which get cleared from irregular access patterns interfering with regularly accessed data and, thus, improve the cache management for the regularly accessed blocks.

In order to capitalize on the benefits of the SDC, some prediction logic able to categorize memory accesses as cache friendly or cache averse is required.

4.3.2 Large Predictor (LP)

To identify memory accesses that should be directed to the SDC we propose the Large Predictor (LP), a microarchitectural prediction mechanism capable of dynamically distinguishing between regular and irregular access patterns. Before triggering a memory access, the core consults LP to decide whether the access must be directed to the SDC or to the normal cache hierarchy (L1D, L2C, LLC). LP builds its predictions on the historic knowledge of strides between memory accesses triggered by instructions with the same PC. LP classifies memory accesses as either regular or irregular. Section 5.3 indicates that accesses featuring large strides have a high probability of being served at DRAM. This behavior is a distinctive characteristic that allows LP to predict whether a certain memory access benefits from accessing the cache hierarchy or the SDC.

LP practically classifies memory accesses between cache averse and cache friendly by leveraging a PC-indexed prediction table. Each prediction table entry consists of (i) a tag of the PC used to access memory (`entry.tag`); (ii) the block address of the previous access performed by this PC (`entry.addr`); (iii) a field that stores an accumulation of the previous strides between cache blocks accessed by instructions corresponding to this predictor entry (`entry.s_acc`); and (iv) a valid bit (`entry.valid`).

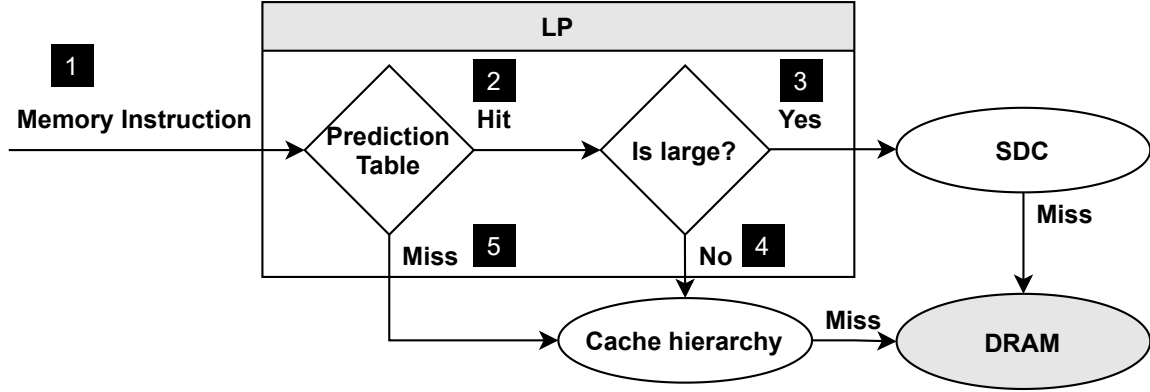


Figure 4.4 Operation of the LP on a prediction event.

4.3.2.1 LP Operation

Figure 4.4 provides a step-by-step illustration of the prediction process. Upon executing a memory instruction and prior to sending the memory request to the memory sub-system **1**, the core consults the LP providing it a tuple $(PC, v@)$, where PC is the instruction's PC and $v@$ is the block address of the fetched data. The PC is hashed into a tag and set index that are both used to look-up the prediction table. The set index is computed as $PC \bmod \#sets$ and the tag as $PC \gg \log_2(\#sets)$. On a prediction table hit **2**, the stride field entry $.s_acc$ of the matching entry is compared to a global threshold τ_{glob} . If the value of the entry $.s_acc$ field satisfies the condition $entry.s_acc \geq \tau_{glob}$, the access is served by the SDC **3**. Conversely, if the entry $.s_acc$ field is lower than the global threshold, the access is routed to the cache hierarchy (L1D, L2C and LLC) **4**. On a prediction table miss **5**, the memory access is also served by the cache hierarchy.

4.3.2.2 LP Update

Figure 4.5 provides a step-by-step illustration of an operational example of a LP update. Upon the trigger of a memory access **1**, the core consults the LP providing it with a tuple containing the instruction PC and the block address: $(PC, v@)$. The PC is hashed into a tag and a set index that are used to look-up the prediction table **2**. On a hit in the prediction table, the block address ($entry.addr$) stored in the prediction table entry is read, and the stride s between the block address ($entry.addr$) and the current block address $v@$ is computed as $s = |v@ - entry.addr|$ **3**. The value stored in the entry $.s_acc$ field of the prediction table entry is accumulated to this stride s , and a right bit shift is performed on the accumulated value **4**.

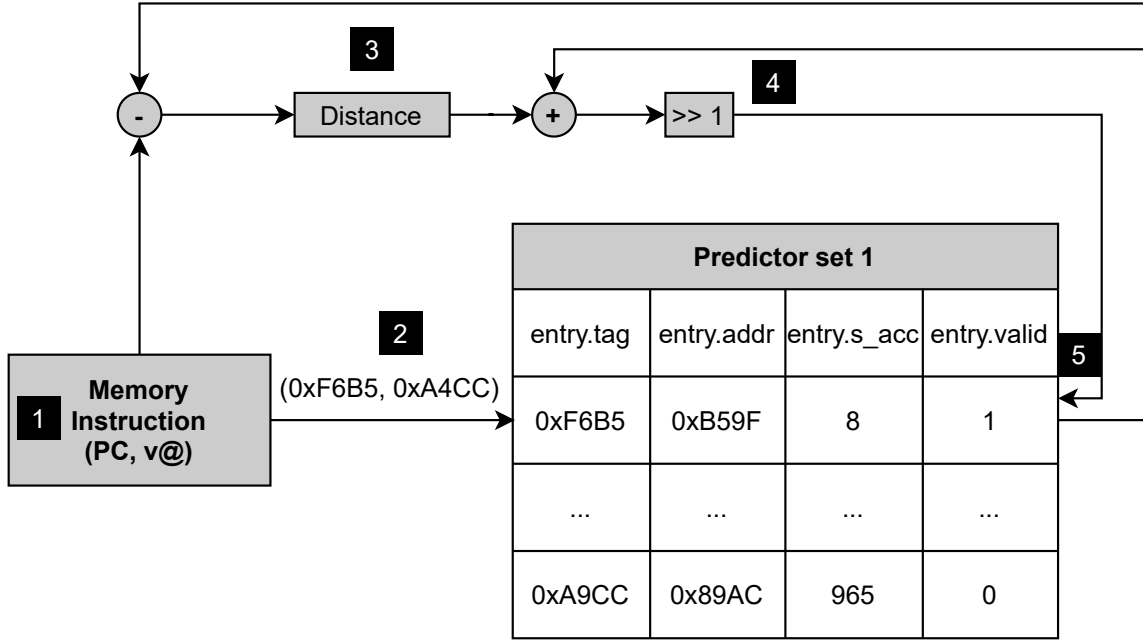


Figure 4.5 Organization of the LP predictor and a functional example of its update sequence.

4.3.2.3 LP Replacement

Upon a prediction table miss, a victim is selected using the LRU replacement policy. The content of the victim entry is then initialized such that the tag is set to $PC \gg \log_2(\#sets)$, the `entry.addr` field is initialized to the block address `v@` attached to the instruction, and the `entry.s_acc` field is initialized to 0.

4.3.3 Coherence

To handle coherence between the SDC and the rest of the cache hierarchy, we extend the cache directory¹ with the *SDCDir*, a directory that tracks the state of the cache blocks stored in the SDCs. Each *SDCDir* entry stores the tag of a cache block, the status bits required by the coherence protocol, and a bit vector of sharer cores. The number of entries and the associativity of the *SDCDir* is configurable. Figure 4.6 shows a scheme of the *SDCDir*.

The *SDCDir* allows extending the coherence protocol and maintaining coherence in the SDCs without major modifications. To do so, coherence requests from the SDCs and the L2s access the cache directory and the *SDCDir* in parallel, and they trigger the same actions as defined in the coherence protocol. For instance, in invalidation-based coherence protocols like MESI [77] and MOESI [93], read requests from the SDCs and the L2s lookup the cache

¹Note that the cache directory can be architected as part of the LLC or as a separate hardware structure. The proposed coherence mechanism works for both design options indistinctly.

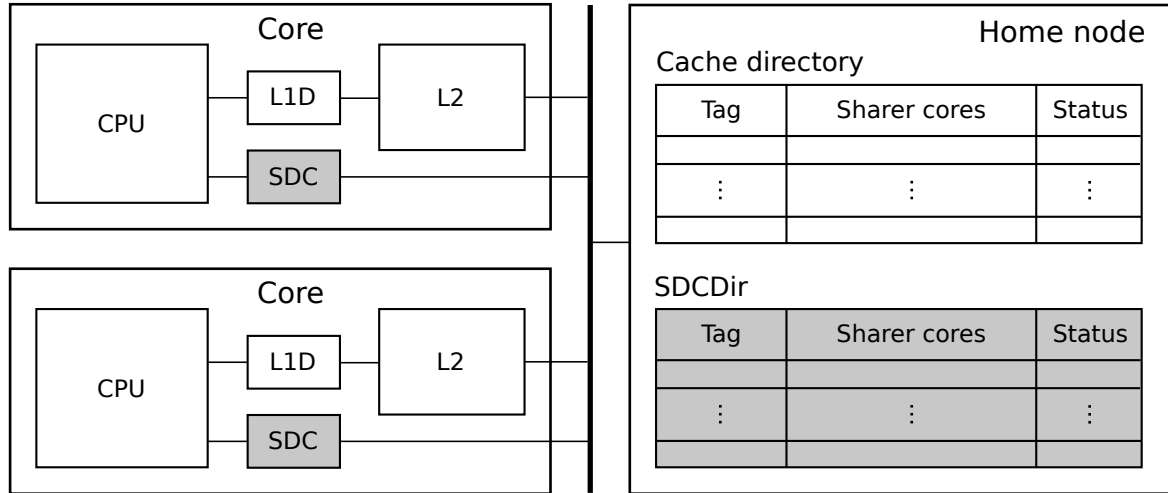


Figure 4.6 Hardware support for SDC coherence.

directory and the SDCDir in parallel. If no valid copy of the cache block is present in any cache nor in any SDC, the request is sent to DRAM. Otherwise, the request is served from the cache or the SDC that holds the valid copy of the cache block. Similarly, write requests from the SDCs and the L2s lookup the cache directory and the SDCDir in parallel and invalidate the cache block in the L1s, L2s and SDCs of the remote cores that keep a valid copy, writing the cache block back to DRAM if it was dirty. This operation ensures that the same cache block can be present at the same time in the cache hierarchy (L1, L2, LLC) and the SDCs only if the block is clean, otherwise only a single valid copy of the cache block can be present either in the cache hierarchy or in the SDCs.

The SDCDir maintains precise information of the data stored in the SDCs. When a SDC request misses in the SDCDir, the request is served by the cache hierarchy or the DRAM and a new SDCDir entry is created for the cache block. Subsequent requests to the same cache block from any SDC update the status bits and the bit vector of sharer cores of the SDCDir entry as defined by the coherence protocol. Upon replacements in the SDCDir, the entry selected as victim is invalidated and all the copies of the corresponding cache block in all the SDCs are also invalidated, writing the block back to DRAM if needed.

4.3.4 Putting It All Together

This section explains the operation when the LP and the SDC are combined (SDC+LP) to accelerate the memory accesses of irregular workloads.

As shown on Figure 4.4, upon a memory access, LP is consulted by the core and answers with a Boolean prediction, saying whether this memory access must be directed to the SDC or to the standard cache hierarchy (L1D, L2C, LLC). If the access is directed to the SDC and

the look-up hits, the data block is returned to the core benefiting the very low latency of the SDC. Otherwise, if the access misses in the SDC, a lightweight coherence message is sent to the cache directory to ensure correctness and the access is served from either a remote cache or the DRAM. In case of accessing the DRAM, the cache block is directly inserted into the SDC, bypassing the L2C and the LLC and thus reducing the latency of the memory operation while avoiding polluting the rest of the cache hierarchy. Conversely, if the LP predicts that the L1D must be accessed, the cache block is requested to the L1D and the operation of the cache hierarchy proceeds as in a regular system without SDCs.

This combination allows feeding the SDC with accesses to irregularly accessed data-structures that, if directed to the L1D, would most likely miss all the way through the cache hierarchy and access the DRAM, leading to poor performance due to the latency costs of useless cache look-ups. In addition, directing irregular memory accesses to the SDC avoids polluting the rest of the cache hierarchy. As a result, the L1D, the L2C, and the LLC experience better locality and can make better cache management decisions for cache friendly memory accesses, without suffering the negative effects of handling cache averse memory accesses. In addition, capitalising on these benefits does not require modifications in any software layer and only introduces minimal storage overheads.

4.3.5 Context Switches

On context switches, LP and SDC must be flushed since the information they contain becomes useless. The performance impact of flushing the SDC upon context switches is the same as that of flushing private caches, that is, negligible since context switches are infrequent events. In addition, since LP requires a very limited number of entries as Section 4.5.2 demonstrates, filling it with meaningful entries requires a minimal number of memory accesses. Therefore, the performance impact of flushing the LP upon context switches is negligible.

4.4 Methodology

4.4.1 Simulation Infrastructure

We evaluate our proposal with ChampSim[29], a detailed simulator that models a 4-wide out-of-order CPU along with its cache hierarchy, prefetching mechanisms, and memory sub-system. Table 4.1 provides our system configuration based on the recent server Intel Cascade Lake micro-architecture [26].

Component	Description
Branch Predictor	hashed perceptron
CPU	2.166 GHz, 4-wide out-of-order processor 6-stage pipeline, 224-entries re-order buffer
L1 ITLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle latency, 8-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle latency, 16-entry MSHR
L1-I Cache	32, 8-way, 4-cycle latency, 10-entry MSHR
L1-D Cache	32, 8-way, 4-cycle latency, 10-entry MSHR LRU replacement, next line prefetcher
SDC	8, 2-way, 1-cycle latency, 10-entry MSHR LRU replacement, next line prefetcher
LP Predictor	552, 32-entries, 8-way, LRU replacement, $\tau_{glob}=8$
L2 Cache	1, 16-way, 10-cycle latency, 16-entry MSHR LRU replacement, SPP prefetcher [61]
LLC	1.375 per core, 11-way, 56-cycle latency, 64-entry MSHR LRU replacement
SDCDir	128 entries per core, 8-way, 1-cycle latency, LRU replacement
DRAM	16 per core, DDR4 SDRAM data-rate: 2.933 GT/s, I/O bus frequency: 1466.5 MHz $t_{RP} = t_{RCD} = t_{CAS} = 24$ cycles

Table 4.1 System configuration

4.4.2 Graph-Processing Applications

We use six graph-processing applications from the GAP benchmark suite [15]. Breadth-First Search (BFS) is a fundamental graph traversal algorithm. Page Rank (PR) iteratively updates per-vertex ranks until convergence. Connected Components (CC) applies the Shiloach-Vishkin [88] algorithm to compute the largest connected components of the graph. Betweenness Centrality (BC) uses the Brandes algorithm [23] to approximate the per-vertex centrality scores. Triangle Count (TC) counts the number of triangles in the graph. Finally, Single-source Shortest Paths (SSSP) uses δ -stepping [66] to return the distance of all vertices of a graph to a given source vertex. Table 5.4 shows the main characteristics of these six applications, including the size of property array elements, and input parameters like the execution style (push or pull), or the use of frontiers.

	BC [15]	BFS [15]	CC [15]	PR [15]	TC [15]	SSSP [15]
irregData ElemSz	8 + 4	4	4	4	4	4
Execution style	Push-Mostly	Push & Pull	Push-Mostly	Pull-Only	Push-Only	Push-Only
Use Frontier	Yes	Yes	No	No	No	Yes

Table 4.2 Graph kernels

	Web [15]	Road [15]	Twitter [15]	Kron [15]	Urand [15]	Friendster [108]
# Vertices (in M)	50.6	23.9	61.6	134.2	134.2	65.6
# Edges (in M)	1,949.4	58.3	1,468.4	2,111.6	2,147.4	3,612.1

Table 4.3 Input Graphs

For each kernel we consider 6 different input graphs from areas like social networks or path-planning. Table 5.5 lists them. These graphs feature different sizes and distributions of node degrees (power-law, normal, etc.). Different degree-distributions produce different memory access patterns. For instance, when node degrees are distributed following a power-law function, there are a few highly connected graph nodes that yield more data reuse opportunities than vertices with a few connections.

4.4.3 Single-Threaded Workloads

Our set of single-thread graph-processing workloads is composed of the 36 combinations of kernels and inputs listed in Tables 5.4 and 5.5, respectively. We use the SimPoint methodology [79] to identify intervals representative of each workload. Each SimPoint is 1 billion instructions long and characterizes a different phase of these workloads. Each SimPoint is executed for 200 million instructions to warm-up the memory hierarchy and other microarchitectural structures, and it is executed for an additional set of 200 million instructions to obtain experimental results.

4.4.4 Multi-Threaded Workloads

We randomly generate 50 distinct 4-thread workloads composed of mixes of all available 36 single-thread workloads. Our performance results on multi-thread workloads report the weighted speed-up normalized to the baseline. This metric is commonly used to evaluate multi-threaded scenarios [55, 87] since it avoids performance overestimation due to high-IPC threads. The metric is computed as follows: for each thread running on the simulated system,

we compute its IPC in a shared environment (IPC_{shared}) and its IPC in isolation on the same system (IPC_{single}). We then compute the weighted IPC of the mix as the weighted sum of $IPC_{shared}/IPC_{single}$ for all the benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC of the baseline design.

4.4.5 Alternative Approaches

We compare our proposal, SDC+LP, against five previously proposed approaches aimed at boosting the performance of graph-processing and irregular workloads: i) The Transpose-based Cache Replacement (T-OPT) [10], the state-of-the-art cache replacement policy for graph data based on the graph adjacency matrix. T-OPT (and its practical but less performant implementation, P-OPT) requires modifying the original application; ii) the Distill Cache [81] technique that retains only the used words within a cache line and evicts the unused ones to optimize the use of cache storage capacity; iii) a scenario where we enhance the L1D with 8KB of storage capacity by increasing its number of ways from 8 to 10. These additional 8KB correspond to the storage budget of the SDC per core. We call this approach L1D 40KB IS0; iv) a scenario where we double the size of the LLC by increasing the number of sets from 2048 to 4096 that we call 2xLLC; and v) an expert-based classification scheme that identifies via judicious source code and performance data analysis which data structures produce memory access patterns that are cache averse and should thus be routed to the SDC. We perform a characterization of the access patterns seen by each individual data structure for all considered workloads. We call this scenario Expert Programmer. Our evaluation also considers a Baseline configuration that corresponds to a system with a standard cache hierarchy. All system configuration parameters are specified in Table 4.1.

4.5 Evaluation

This section evaluates SDC+LP. Section 4.5.1 compares SDC+LP against other hardware proposals in the single-core context, as well as an explanation of the performance delivered by SDC+LP. Section 4.5.2 presents an exhaustive design space exploration to justify SDC+LP design choices. Section 4.5.3 compares the performance achieved by SDC+LP against the Expert Programmer approach. Section 4.5.4 evaluates SDC+LP in the multi-core context. Sections 4.5.5 and 4.5.6 report results on the SPEC benchmark suite and describe the hardware cost of SDC+LP.

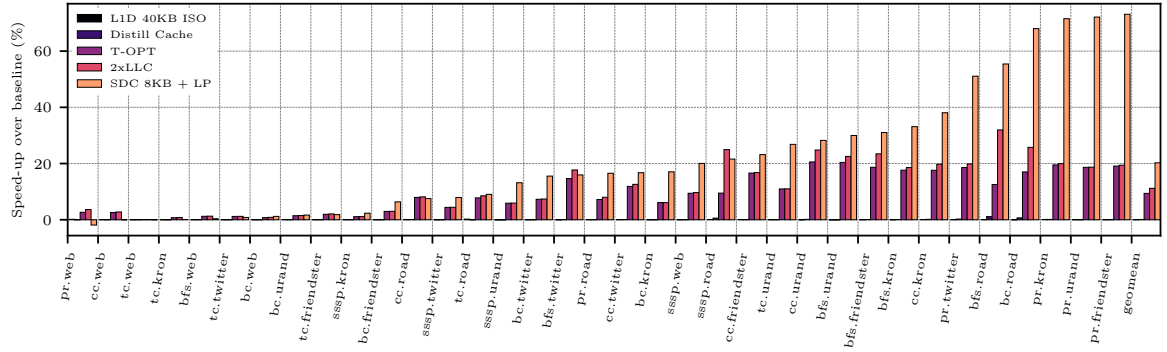


Figure 4.7 Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture.

4.5.1 Single-Core Evaluation

This section compares SDC+LP against four hardware approaches: T-OPT, Distill Cache, L1D 40KB ISO, and 2xLLC. Section 5.5 describes these four hardware approaches. Figure 4.7 shows in the y-axis the performance improvement that SDC+LP and the other hardware approaches achieve with respect to the Baseline architecture with a standard cache hierarchy. The x-axis displays the 36 graph processing workloads we consider.

Both L1D 40KB ISO and the Distill Cache designs fail to provide significant performance improvements, as they only achieve geometric mean speedups of 0.0% and 0.1%, respectively. By contrast, T-OPT can provide a significant geometric mean speedup of 9.4% by making cache replacement decisions based on graph adjacency matrices. The 2xLLC is also able to significantly improve performance, achieving a geometric mean speedup of 11.2% over the baseline and slightly outperforming T-OPT. Finally, LP+SDC doubles the performance improvements of both T-OPT and 2xLLC by providing a geometric mean speedup of 20.3% over the baseline.

We conduct a performance analysis considering MPKI rates of both Baseline and SDC+LP to explain the large performance improvements of SDC+LP. Figure 4.8 displays L2C and LLC MPKI rates considering both the Baseline and the SDC+LP approaches for each workload. Two stacked bars per workload are depicted: the left-hand side bar represents MPKI rates of Baseline, and the right-hand side bar represents MPKI rates of SDC+LP. For each bar, the lower stack refers to the LLC MPKI, and the total height refers to the L2C MPKI. Therefore, the difference between the the lower stack and the total bar height comes from accesses that miss in the L2C and hit in the LLC.

Similarly, Figure 4.9 shows MPKI rates in the first-level caches of Baseline and SDC+LP. The leftmost bar of each workload shows the L1D MPKI of Baseline, while the rightmost bar of each workload presents the accumulated MPKI of the L1D and the SDC of SDC+LP.

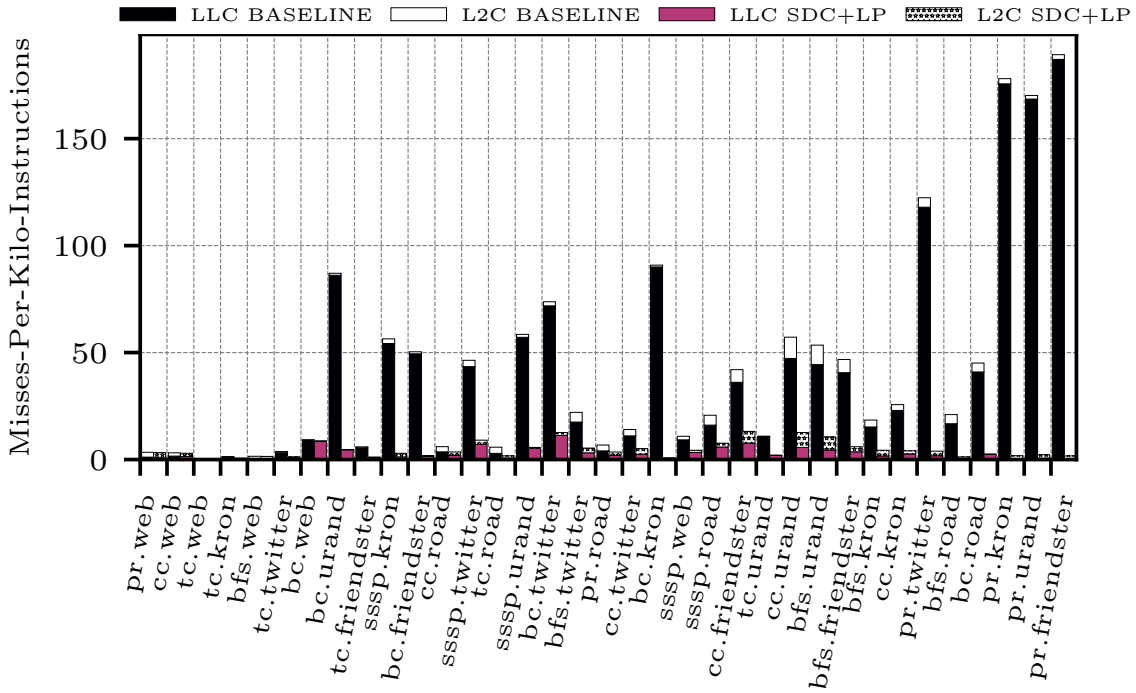


Figure 4.8 L2C and LLC MPKI ratios of SDC+LP and Baseline.

These two figures present the workloads sorted in ascending order in terms of the speed-up provided by the SDC+LP design as shown on Figure 4.7.

Figure 4.8 indicates that SDC+LP significantly reduces the pressure on both L2C and LLC with respect to Baseline. Indeed, average MPKI rates drop from 44.5 and 41.8 (Baseline) to 4.4 and 2.8 (SDC+LP) for L2C and LLC, respectively. Figure 4.9 shows that SDC handles the vast majority of the misses experienced by the L1D in the Baseline configuration. The Average MPKI rate of L1D decreases from 53.2 (Baseline) to 7.4 (SDC+LP) while the SDC experiences an average MPKI rate of 48.3. These measurements illustrate how the LP predictor successfully identifies cache averse accesses and redirects them to the SDC, avoiding pollution of the contents of L1D, L2C and LLC caches, and eliminates useless cache lookups for cache averse accesses. These two effects decrease the latencies of cache averse accesses, since they do not waste time in useless cache look-ups, and cache friendly accesses, since cache lines that serve them do not suffer from evictions due to insertions of cache averse data. These two effects combined explain the large benefits of SDC+LP.

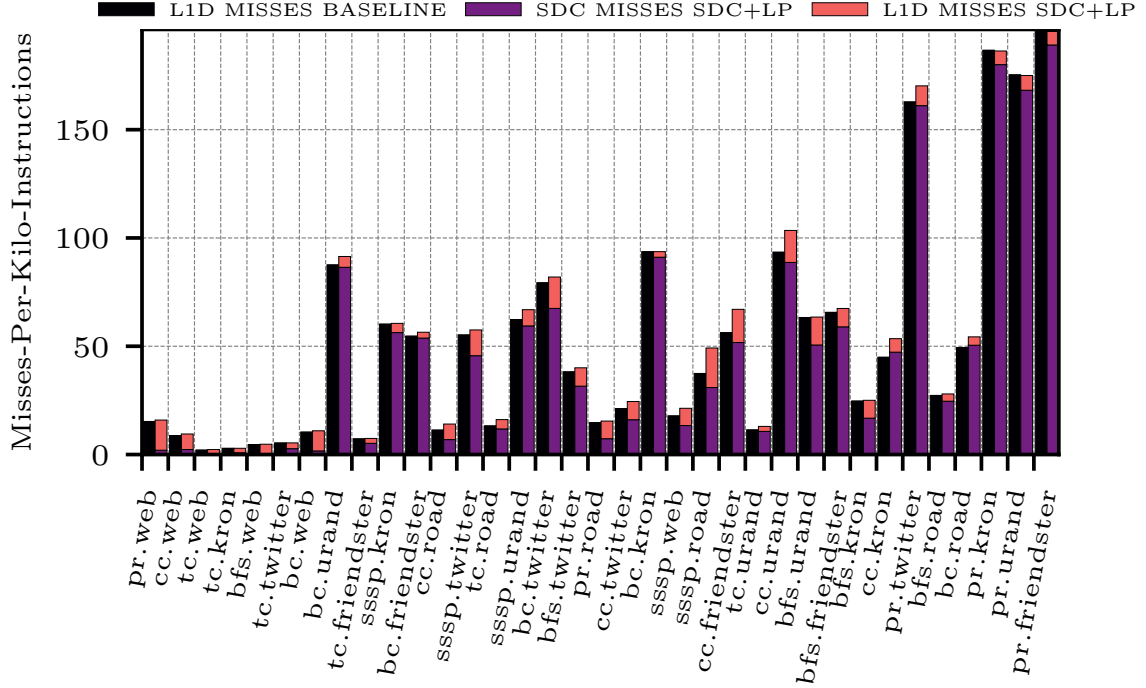
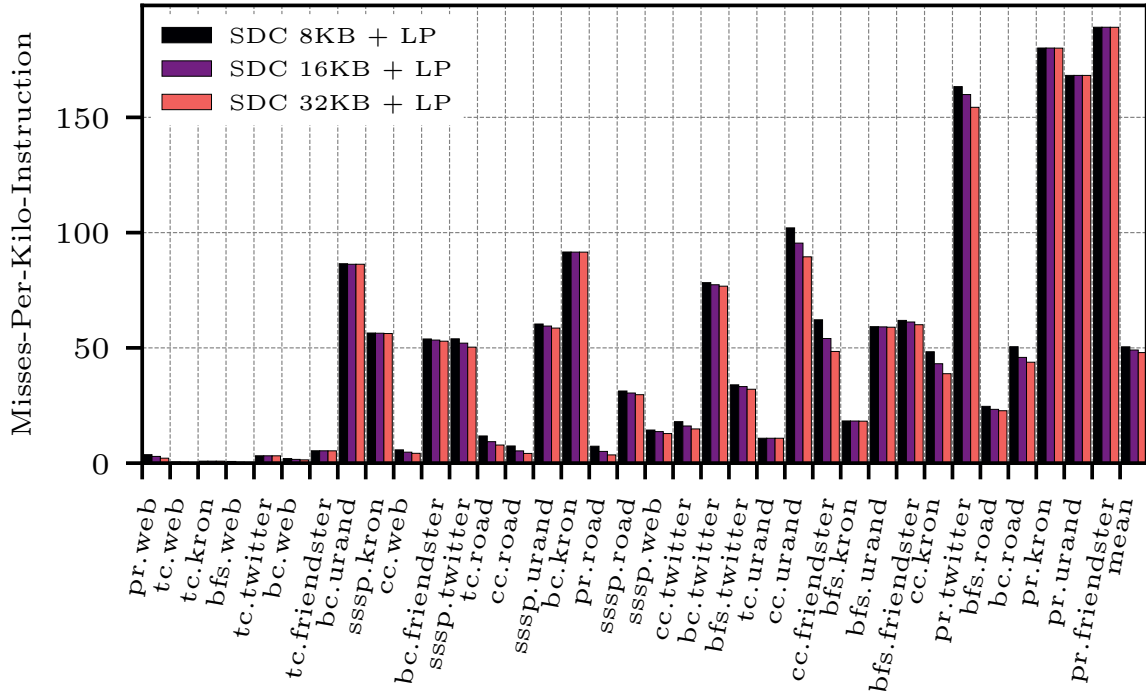


Figure 4.9 L1D and SDC MPKI ratios of SDC+LP and Baseline.

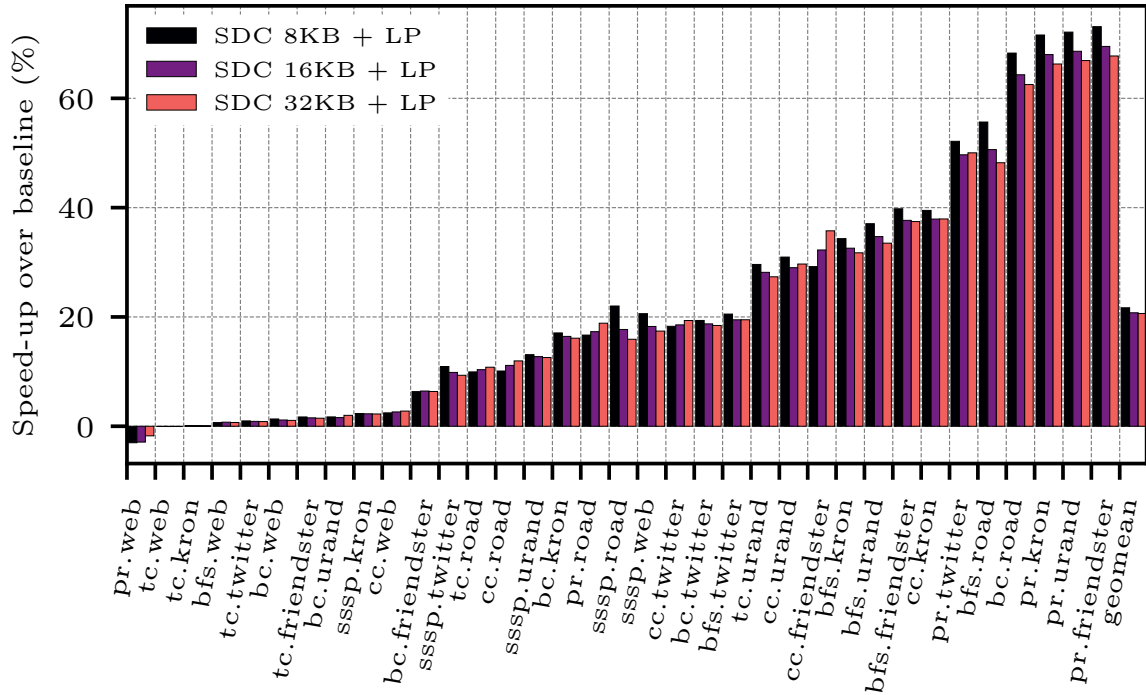
4.5.2 Design Space Exploration

4.5.2.1 SDC Size

We consider different SDC sizes and compare the performance they obtain. We consider SDC sizes of 8, 16, and 32KB. We adapt the associativity and latency of the SDC in the following way: The 8KB SDC is 2-way set associative and has a 1-cycle latency. The 16KB and 32KB SDCs are 4-way and 8-way set associative, and have 3-cycle and 4-cycle latencies, respectively. We use the LP parameters specified in Table 4.1. Figure 4.10a reveals that SDC MPKI experiences very small reductions as its size increases. We observe 50.5, 49.1, and 48.0 MPKI rates for 8KB, 16KB, and 32KB SDCs, respectively. Figure 4.10b indicates that even if a 32KB SDC and 16KB SDC experience fewer misses than their 8KB counterpart, they provide slightly smaller performance benefits. This small decrease of performance while increasing SDC size comes from the larger latencies of the 16KB and 32KB SDCs. By contrast, the 8KB SDC allows very fast cache look-ups while the MPKI only slightly increases.



(a) SDC MPKI ratios considering 8KB, 16KB, and 32KB of storage.



(b) Performance improvement of SDC+LP for 8KB, 16KB and 32KB SDCs.

Figure 4.10 Exploration on the size of the SDC.

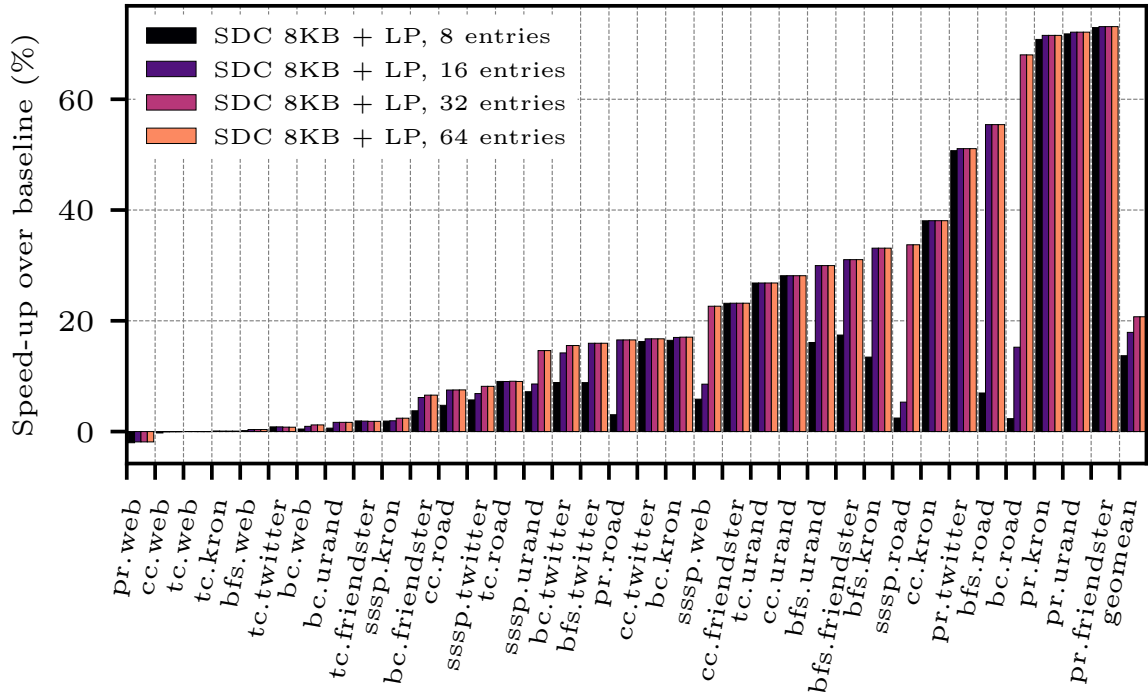


Figure 4.11 Speed-up of SDC+LP considering a fully-associative LP with different entry counts.

4.5.2.2 LP Organization

We evaluate the performance impact of the two most important LP configuration parameters: (i) number of entries of the LP prediction table; and (ii) associativity of the table.

Figure 4.11 considers a fully-associative LP prediction table and reports performance improvements of 13.7%, 17.9%, 20.7% and 20.7% SDC+LP against Baseline when using 8-, 16-, 32- and 64-entry prediction tables. The SDC parameters are specified in Table 4.1. This evaluation indicates that a 32-entry prediction table maximizes the benefits of SDC+PP as a 64-entry table achieves very similar performance.

Figure 4.12 considers a 32-entry LP prediction table and varies its associativity, ranging from a fully-associative organization to a direct-mapped structure. Similarly as before, the SDC configuration is specified in Table 4.1. We observe performance improvements of 17.0%, 20.3%, 20.7%, and 20.7% for the direct-mapped, 8-way, 16-way, and fully-associative prediction tables over Baseline, respectively. The 8-way design provides close to optimal results.

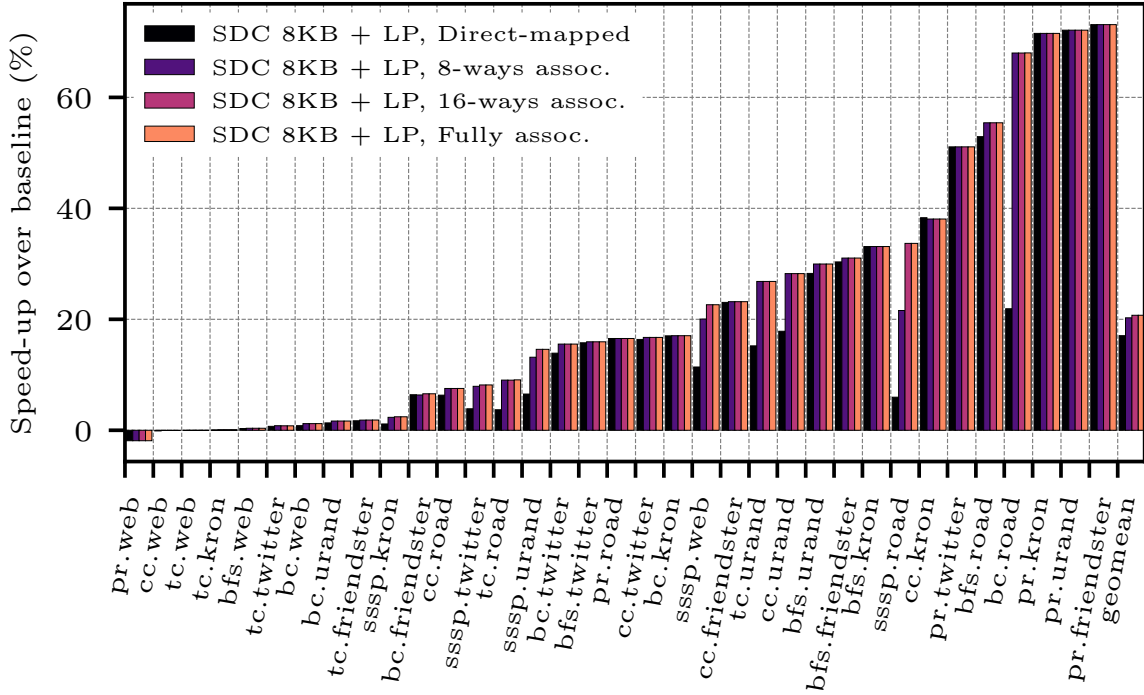


Figure 4.12 Speed-up of SDC+LP considering different LP associativities.

4.5.2.3 Global Threshold

We evaluate the performance impact of the LP global threshold, τ_{glob} . We consider values ranging from 0 to 2^{16} . Setting τ_{glob} to 0 means that all memory accesses are routed to the SDC. Conversely, setting τ_{glob} to large values like 2^{16} implies that all memory accesses are routed to the L1D, a scenario equivalent to Baseline. We perform an extensive analysis considering not only the GAP benchmarks but also the SPEC 2006 [91] and SPEC 2017 [92] benchmark suites to make sure that SDC+LP does not hurt the performance of general-purpose workloads. We find that setting τ_{glob} to 8 provides significant performance improvements for graph-processing workloads like GAP benchmarks (20.3% improvement over Baseline) while not harming the performance of general-purpose workloads like SPEC (0.5% improvement over Baseline). Section 4.5.5 describes the performance of SDC+LP on SPEC 2006 and SPEC 2017 benchmarks.

4.5.3 Comparison with An Expert Programmer

We compare SDC+LP with Expert Programmer, an approach that relies on a judicious analysis of performance data to identify data structures that display either cache averse or cache friendly memory access patterns. The SDC+LP configuration parameters are described

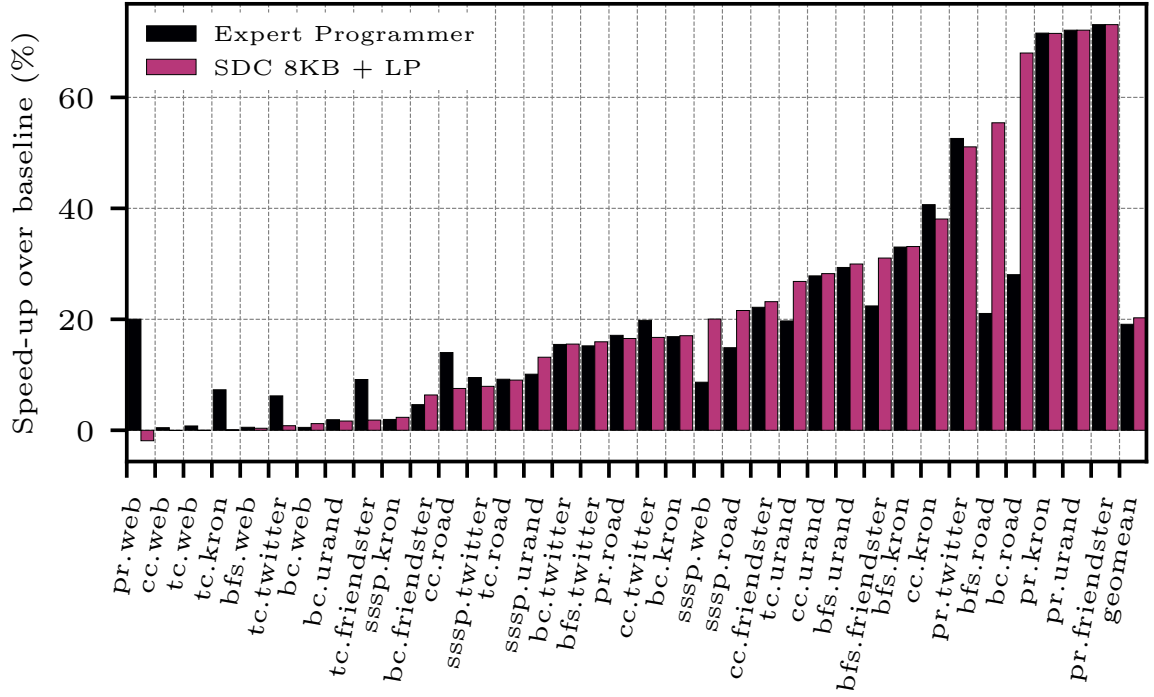


Figure 4.13 Speed-up over Baseline of SDC+LP and the Expert Programmer approach, which manually categorises cache averse and cache friendly accesses.

in Table 4.1. Expert Programmer uses an 8KB SDC configured as Table 4.1 describes. Figure 4.13 shows how Expert Programmer achieves a 19.1% performance improvement over Baseline, while SDC+LP achieves 20.3%. These results confirm that the LP predictor is able to distinguish between cache averse and cache friendly memory accesses and achieve very similar performance as an expert-driven approach where cache averse accesses are identified via analyzing performance data. The performance improvement of SDC+LP over Baseline is very close to the one achieved by Expert Programmer for a large portion of the 36 considered workloads. However, SDC+LP outperforms Expert Programmer on scenarios where graph connectivity is very heterogeneous and accesses to graph data are sometimes cache averse and sometimes cache friendly, like *bc.road*. Alternatively, Expert Programmer outperforms SDC+LP in scenarios where setting the τ_{glob} to 8 is not adequate, like *pr.web*.

4.5.4 Multi-Core Evaluation

This section evaluates in the multi-core context the performance of SDC+LP and four additional hardware approaches: T-OPT, Distill Cache, L1D 40KB ISO, and 2xLLC. Each core contains its own private SDC+LP. Figure 4.14 shows the performance of all evaluated designs

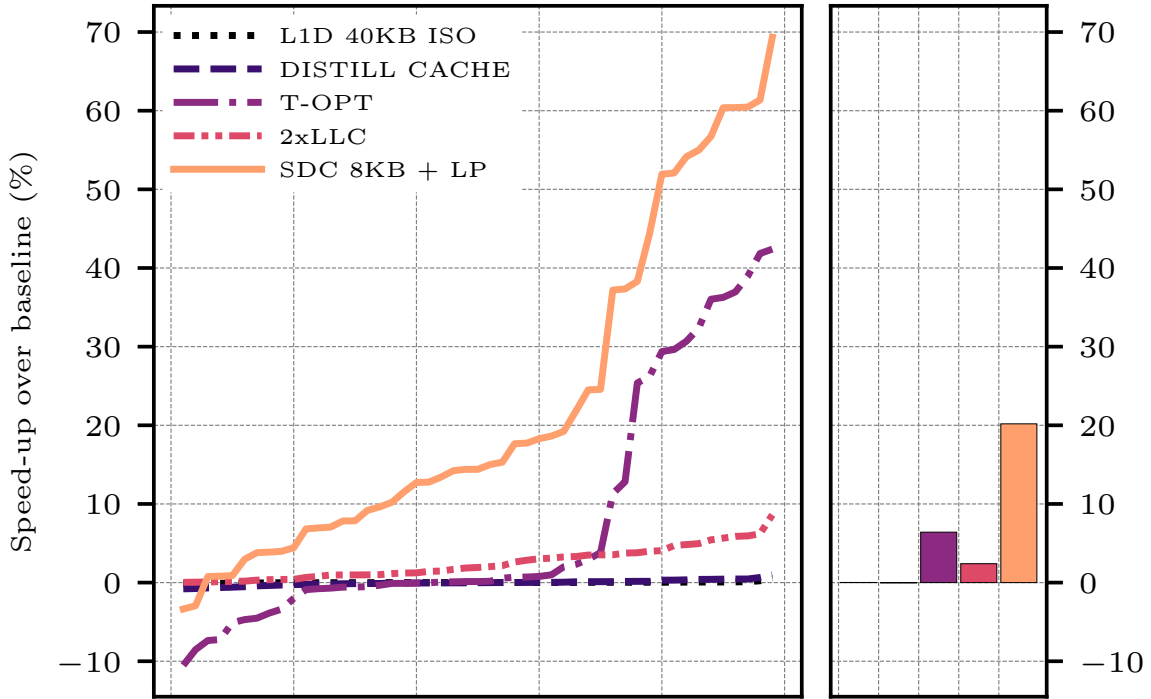


Figure 4.14 Performance improvement of SDC+LP and other considered scenarios with respect to the Baseline architecture in a multi-core scenario.

across a set of the 50 multi-thread graph-processing workloads described in Section 4.4.4. The y-axis shows the performance improvement over Baseline, while the x-axis shows the 50 considered workloads sorted in terms of the improvement achieved by SDC+LP over Baseline. Figure 4.14 also displays the geometric mean performance improvement of all considered scenarios in terms of bars. Section 4.4.4 describes how we compute performance in multi-core scenarios.

Figure 4.14 reveals that the L1D 40KB ISO and Distill Cache approaches achieve very similar performance to Baseline. Their geometric mean speed-ups are 0.02% and -0.04%, respectively. T-OPT achieves remarkable performance gains, particularly for one third of the workloads. T-OPT reaches 6.4% geometric mean improvement over the baseline. The 2xLLC scenario provides improvements above 5% for 10 workloads and achieves a 2.4% geometric mean speed-up over the Baseline. SDC+LP outperforms all considered scenarios as it provides a 20.2% improvement over Baseline. SDC+LP achieves very remarkable performance improvements above 20% for one third of the multi-thread workloads, and reaches a maximum speed-up of 69.3%.

	Entries	Bits per entry	Total KB
SDC	128	512 data + 42 tag + 1 valid + 1 dirty	8.69
LP	32	65 tag + 58 address + 14 stride + 1 valid	0.54
SDCDir	128	42 tag + 6 state + 1 sharer per core	0.77

Table 4.4 Hardware budget per core

4.5.5 Evaluation on SPEC Benchmarks

We evaluate the performance of SDC+LP on the SPEC 2006 [91] and SPEC 2017 [92] benchmarks to assess its benefits on desktop and high-performance computing workloads. SDC+LP achieves 0.5% geometric mean speed-up with respect to Baseline when considering all workloads of the SPEC 2006 and SPEC 2017 benchmark suites. Our evaluation indicates that SDC+LP does not incur any performance degradation when dealing with workloads outside the graph-processing area and, even more, we observe remarkable improvements on SPEC workloads displaying high MPKI rates. For example, SDC+LP achieves 16.3%, 5.4%, and 4.1% speed-ups over Baseline for the *429.mcf*, *450.soplex*, and *403.gcc* workloads, respectively.

4.5.6 Hardware Cost

Table 4.4 details the hardware budget per core of each element of the SDC+LP proposal, assuming an architecture with 48-bit physical addresses. The total hardware budget of SDC+LP is 10KB per core. The most expensive structure in terms of area is the SDC, with a total hardware budget of 8.69 KB per core. The other two structures, LP and SDCDir, have very small area requirements of only 0.54 KB and 0.77 KB per core, respectively.

4.6 Related work

In the recent years, numerous designs have been proposed to accelerate the irregular accesses of graph-processing applications. In Section 4.5 we compared our proposal to the most relevant published works available to date. This section comments on additional works on graph-processing workloads.

4.6.1 Replacement Policies

Numerous complex cache replacement policies have been proposed in recent literature [47, 106, 52, 87]. These works provide significant performance benefits when applied to general-purpose computing applications, not necessarily graph-processing workloads. Indeed, as we saw in Chapter 3 these replacement policies are ineffective over graph kernels using reasonably sized graph inputs due to the irregularity of the memory access patterns.

Specifically, to overcome the challenges of cache management in the context of, researchers have turned their attention to techniques such as cache-bypassing and reuse prediction. The premise of cache bypassing is to judiciously prevent some memory blocks to be inserted in the caches, thus evicting useful data. Reuse prediction aims at prioritizing cache blocks for eviction when they are predicted to be reused in the distant future. Such techniques have proven to leverage significant performance improvement.

More ample information about cache replacement policies can be found in Section 2.2.

4.6.2 Hardware Prefetching

Stream and strided cache prefetchers are unable to effectively prefetch for the indirect memory access patterns of graph-processing workloads [13, 10]. Yu et al. [110] propose IMP, a microarchitectural prefetching module that identifies and prefetches indirect memory access patterns without requiring any application nor software information. Ainsworth et al. [4] propose a prefetcher that leverages application-level information to capture indirect memory access patterns. Basak et al. [13] propose DROPLET, a physically decoupled prefetcher that takes into account the reuse distances when applying prefetching for different graph types. Although effective, these hardware prefetchers can easily saturate memory bandwidth and become bandwidth-bound. Contrarily, our proposal reduces the latency cost of graph memory accesses by routing these accesses on the most appropriate memory path.

More ample information about hardware prefetching mechanisms can be found in Section 2.3.

4.6.3 Memory System Optimizations for Graph-Processing

Recent research shows the advantages of optimizing the memory hierarchy for graph-processing workloads. Ozdal et al. [75] leverage scratchpads for storing vertex and edge data, while Gonzalez et al. [40] employ a large eDRAM scratchpad to fit larger graph data volumes compared to traditional SRAMs. Prior studies [111, 89, 70, 36, 2] reduce latency in graph memory accesses through near-memory computing, executing operations close to

DRAM to mitigate latency costs. Our approach complements these efforts by enhancing memory access management within the cache hierarchy.

The Distill Cache [81] reserves a section of the L2C cache for storing used words of an evicted cache line. This enhances cache storage utilization by evicting only unused words within each line. Our design addresses irregular access patterns by categorizing memory accesses as regular or irregular. By labeling non-cache-friendly accesses, we mitigate cache pollution and unnecessary cache look-ups.

Selective Cache [39] allows memory accesses lacking temporal or spatial locality to bypass the L1D. It employs a predictor to classify accesses as temporal, spatial, or irregular. This approach avoids L1D impact from irregular patterns. In contrast, our work leverages observations on graph-processing memory patterns to predict whether DRAM will serve memory accesses. By sidestepping lower-level cache pollution, our method allows the cache hierarchy to handle regular graph-processing accesses while avoiding futile cache look-ups for non-regular accesses.

Victim Cache [56], a small fully-associative cache found on the LLC refill path in modern processors, diminishes conflict misses by containing cache eviction victims. On an LLC miss, both the LLC and Victim Cache are consulted. If the requested block resides in the Victim Cache, the LLC victim and Victim Cache entry are swapped, significantly reducing miss latency. A Victim Cache miss fetches the block from DRAM, with the LLC victim entering the Victim Cache. Although the Victim Cache enhances performance for workloads like SPEC 2006 [91] and SPEC 2017 [92], it relies on spatial locality for cache victim insertion. In contrast, our approach, devoid of locality assumptions, bypasses the cache hierarchy when inefficiency is anticipated.

4.6.4 Pre-processing Algorithms

Several pre-processing algorithms have been proposed [102, 31, 8] to improve the locality of graph-processing workloads. Although effective, these pre-processing works are orders of magnitude more expensive compared to the runtime of a single traversal [69]. Our work aims at reducing the latency cost of single traversals by routing each memory access to the most appropriate memory path.

4.6.5 Graph-Processing Accelerators

Numerous previous works design and propose graph-processing accelerators [75, 74, 73, 32, 43] to attenuate the memory bottleneck of graph-processing workloads. These accelerators introduce specialized logic and memory system specializations to provide significant perfor-

mance enhancements. However, these works rely on pre-processing to improve performance, a very expensive technique, as explained in 4.6.4. Recently, Mukkara et al. [69] propose a hardware-accelerated traversal scheduler that replaces pre-processing with a simple online locality-aware scheduler. Our work is orthogonal to graph-processing acceleration.

4.7 Summary

This study uncovered a significant issue affecting graph-processing workloads: the under-utilization of conventional cache hierarchies. A substantial portion, approximately 78.6 %, of Level 1 Data (L1D) cache misses propagates to higher-level caches, such as the Level 2 Cache (L2C) and the Last-Level Cache (LLC), leading to costly accesses to Dynamic Random-Access Memory (DRAM). Additionally, we've observed that memory accesses served from DRAM often exhibit long strides compared to previous accesses with the same program counter (PC).

To tackle these challenges, we introduce SDC+LP, a novel solution comprising two core components:

Side Data Cache (SDC): SDC is an auxiliary set-associative cache located alongside the L1D cache on a per-core basis. It is an intelligent buffer to capture and optimize memory access patterns specific to graph-processing workloads.

Large Predictor (LP): LP is a cost-effective micro-architectural predictor that plays a central role in the SDC+LP framework. LP intelligently distinguishes between regular and irregular memory accesses by utilizing historical stride information. This differentiation reduces cache pollution, ultimately leading to more efficient cache utilization.

The integration of SDC with LP within the SDC+LP framework has yielded impressive results. In the context of single-core processing, our solution achieved a remarkable 20.3 % geometric mean speed-up over the baseline configuration. Furthermore, compared to T-OPT, a state-of-the-art cache management scheme tailored for graph-processing workloads, SDC+LP demonstrated an additional 10.9 % geometric mean speed-up.

In multi-core scenarios, SDC+LP continued to excel, achieving a substantial 20.2 % speed-up over the baseline configuration and further improving performance by an additional 13.8 % compared to T-OPT. These achievements highlight the versatility and effectiveness of our proposed solution, which optimizes memory access efficiency across diverse computing environments.

Our study addresses a pressing challenge in cache hierarchy optimization and sheds light on the broader significance of hardware solutions capable of distinguishing between regular

and irregular memory accesses. Such solutions offer significant performance enhancements, particularly for workloads with large memory footprints and data-driven memory access patterns. Our findings and the SDC+LP framework provide valuable insights into cache hierarchy optimization, specifically tailored for the unique demands of graph-processing workloads, with implications extending to the broader domain of memory-intensive applications.

CHAPTER

5

A TWO LEVEL NEURAL APPROACH COMBINING OFF-CHIP PREDICTION WITH ADAPTIVE PREFETCH FILTERING

La vie ne suit pas toujours le même
rythme pour chacun. Certains doivent
avancer plus vite que les autres pour
accomplir leur destinée.

Gustave Flaubert

5.1 Introduction

EMERGING workloads from various domains [44, 13, 25, 24, 35] have very large data footprints that are orders of magnitude larger than the capacity of current cache hierarchies [21]. These workloads frequently trigger DRAM accesses, spending a large portion of their execution time waiting for data transfers to and from DRAM to complete with a detrimental effect on performance and the energy consumption [114, 12, 11, 4, 14, 102].

Prior work has proposed several techniques to mitigate the performance and energy overheads of these applications. These techniques can be broadly classified into four categories: (i) off-chip prediction schemes that predict whether a memory access will result in a DRAM access or hit in the cache hierarchy [48, 84, 65, 109, 19], (ii) aggressive data prefetching with adaptive filters to ensure that only correct prefetches will be issued, (iii) cache bypassing that avoids caching blocks that will not be referenced in the near future [47, 59, 106, 97, 55, 87], and (iv) disruptive cache designs and optimizations for specific workload types [81, 75, 40, 111, 89, 70, 36, 2]. This work focuses on the first two categories and aims at combining their benefits in a cost-effective manner.

Despite their potential for determining the location of requested data in the memory hierarchy, previously proposed off-chip predictors [19, 48] have important drawbacks that undermine their potential for boosting the performance of the memory subsystem while hindering their implementation in real-world designs. For example, the state-of-the-art off-chip predictor [19] triggers two memory accesses, one to DRAM and a second regular request to the cache hierarchy, when it predicts that the corresponding load access will be served from DRAM. While this approach can potentially reduce the latency of a load request that ends up being served from DRAM, it may also significantly increase the number of DRAM transactions. This work shows that, although effective, the state-of-the-art off-chip predictor significantly increases the number of DRAM transactions, which is a critical aspect in bandwidth-constrained scenarios. In addition, our analysis indicates that a large fraction of the inaccurate off-chip predictions is actually served by the first-level data cache (L1D). Therefore, a microarchitectural scheme that selectively delays the off-chip predictions with modest confidence until the L1D lookup is resolved has potential to significantly reduce the number of useless DRAM transactions and deliver higher performance.

Previous approaches have successfully applied prefetch filtering at the lower level caches [85, 116, 86, 20]. However, these approaches are not agile since they are typically optimized on top of specific prefetch engines, incur significant area overheads, and are not exposed to program features that are very valuable to produce accurate predictions (*e.g.*, a complete sequence of accessed virtual addresses). This work argues that the concept of

off-chip prediction can be leveraged to form effective prefetch filters for L1D. Specifically, our analysis demonstrates that the vast majority of the L1D prefetch requests served from DRAM are inaccurate.

To address our findings and improve the performance of memory-intensive workloads, we propose the *Two Level Perceptron (TLP)* predictor. TLP constitutes the first hardware proposal targeting both off-chip prediction and prefetch filtering using a multi-level perceptron hardware approach. TLP is composed of two connected microarchitectural perceptron predictors: the *First Level Predictor (FLP)* and the *Second Level Predictor (SLP)*. FLP is a perceptron hardware predictor located near the core that employs a novel mechanism to reduce the number of DRAM accesses by selectively delaying off-chip predictions when needed. SLP is a perceptron predictor located alongside the L1D. The novelty of SLP relies on leveraging off-chip prediction to drive L1D prefetch filtering using physical addresses as well as the FLP prediction as features. Our evaluation illustrates that TLP yields significantly higher performance than the state-of-the-art off-chip predictor [19] and prefetch filtering scheme [20] across a large set of single-core and multi-core workloads.

This paper makes the following contributions:

- We design and propose *Two Level Perceptron (TLP)* predictor, a scheme composed of two connected perceptron predictors: FLP and SLP. FLP reduces the pressure on the memory subsystem using a novel selective delay mechanism. SLP leverages off-chip prediction to guide prefetch filtering in the L1 data cache. TLP is the first hardware proposal targeting both off-chip prediction and prefetch filtering. TLP only requires 7KB of storage.
- We compare TLP with the state-of-the-art off-chip predictor, Hermes [19], the state-of-the-art prefetch filter, PPF [20], and a combination of both. Our evaluation considers 55 single-core and 200 multi-core workloads. When considering a system that uses IPCP [76] as L1D prefetcher, TLP reduces the average number of DRAM transactions by 30.7% and 17.7%, as compared to a baseline that uses IPCP as L1D prefetcher but no off-chip prediction mechanism, across the single-core and multi-core workloads, respectively, while state-of-the-art approaches significantly increase DRAM transactions. As a result, TLP achieves geometric mean performance speedups of 6.2% and 11.8% across single-core and multi-core workloads, respectively. When considering a scenario with the Berti [71] L1D prefetcher, TLP also outperforms Hermes, PPF, and a combination of them in both single-core and multi-core contexts since it significantly reduces DRAM accesses.

5.2 Background

5.2.1 Off-Chip Prediction

Emerging workloads spanning various domains [25, 44, 13, 24, 35], have a key property in common: massive working set sizes that do not fit in the existing cache hierarchies [21], making cache management a major performance bottleneck for processor design. Indeed, recent work [114, 12, 11, 5, 14, 103] shows that these workloads spend up to 80% of their total execution time waiting for DRAM.

To address the high-latency load requests of these emerging applications, prior work [48, 84, 65, 109, 19] has introduced the concept of *off-chip prediction*. The core idea behind off-chip prediction is to predict whether a memory access will eventually result in a DRAM access or in a hit in the cache hierarchy (L1D, L2C, LLC). Prior work in the domain can be classified in two categories depending on their prediction strategy: i) predict which cache level (L1D, L2C, LLC) will provide a hit, if any [84, 48], and ii) predict whether the cache hierarchy as a whole will provide a hit or not [19]. A representative work from the first category is Level Prediction (LP) [48], a scheme that dynamically predicts where in the memory hierarchy a demanded memory block is most likely to be found. A representative scheme of the second category is Hermes [19], an adaptive perceptron-based off-chip predictor that routes demand load requests directly to DRAM when it is confident that the load will miss in all cache levels.

Hermes [19] is the state-of-the-art microarchitectural off-chip prediction scheme. At the core of Hermes, there is a perceptron predictor composed of several prediction tables, one per selected program feature, similar to prior work on perceptron-based microarchitectural prediction: from branch prediction [37, 49, 50] to cache replacement policies [97, 53] and other intelligent modules [20].

Before a demand load request is issued to the cache hierarchy, *i.e.*, before accessing the L1D, Hermes is consulted to provide a prediction. If the prediction is positive, *i.e.*, the demand load request is predicted to go off-chip, the core issues two requests: one regular request to the cache hierarchy, that might go down to DRAM, and another speculative request that fetches the cache line from DRAM in an attempt to hide the latency cost of accessing the caches. When a demand request eventually returns to the core to be consumed, the training logic of Hermes compares the original prediction with the actual outcome and accordingly updates the weights in the prediction tables.

5.2.2 Prefetch Filtering

Hardware prefetching is a technique that proactively fetches blocks in the cache hierarchy before they are explicitly requested by a core. Hardware prefetchers need to deal with two metrics that are at odds with one another: miss coverage and prefetching accuracy. Aggressive prefetchers typically have high coverage but low accuracy while conservative prefetchers tend to have low coverage and high accuracy.

To gracefully handle the coverage-accuracy tradeoff, smart prefetch filters and throttling schemes have been proposed [85, 116, 86, 20] that are capable of accurately identifying useless prefetch requests and discarding them. An effective prefetch filter would increase the accuracy of a hardware prefetcher without harming its coverage, resulting in higher performance by enabling better cache management.

The state-of-the-art prefetch filter is the Perceptron-based Prefetch Filter (PPF) [20], a perceptron predictor that uses several program features to filter out inaccurate prefetch requests, increasing the accuracy of the underlying prefetcher. Although effective, PPF has two limitations. First, PPF is built and optimized on top of a specific prior prefetcher [61], thus it requires significant engineering effort as well as feature exploration and tuning to make it filter effectively the requests of other prefetchers. Second, PPF requires around 40KB of storage that hinders its implementation in real-world designs.

5.3 Motivation

This section motivates the need for better off-chip predictors and highlights the potential of leveraging the concept of off-chip prediction to apply effective prefetch filtering for the L1D cache. Section 5.3.1 characterizes the cache behavior of contemporary applications, showing that a large fraction of the memory accesses that miss in the L1D result in a DRAM access. Section 5.3.2 analyzes the behavior of Hermes [19], the state-of-the-art off-chip predictor presented in Section 5.2.1, in both single-core and multi-core contexts. Our analysis indicates that Hermes significantly increases DRAM bandwidth consumption, especially in multi-core contexts, opening a path for potential improvement by reducing the number of additional DRAM transactions triggered by Hermes. We demonstrate that selectively delaying the Hermes prediction until the L1D access completion can provide great benefits by lowering the inaccurate off-chip predictions of Hermes. Section 5.3.3 focuses on L1D cache prefetching and characterizes the inaccurate prefetches issued by two state-of-the-art L1D prefetchers depending on the cache level where they were prefetched from and reveals that leveraging the properties of off-chip prediction can be very useful at designing an effective

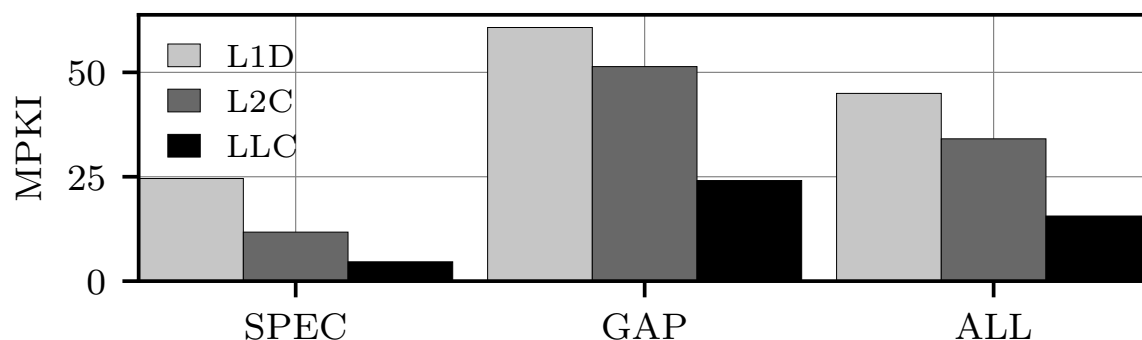


Figure 5.1 MPKI of all caches (L1D, L2C, LLC) across the SPEC (SPEC CPU 2006 and SPEC CPU 2017) and GAP workloads.

prefetch filter for L1D. Section 5.5 presents in detail our simulation infrastructure and all the considered workloads.

5.3.1 Cache Behavior of Modern Workloads

Prior work discussed in Section 5.2.1 shows that the majority of demand load requests of applications featuring huge data working sets miss in all levels of the cache hierarchy, triggering many DRAM accesses. This section analyzes the cache behavior of our complete set of single-core workloads presented in Section 5.5.

Figure 5.1 shows the Misses per 1000 Instructions (MPKI) rates in all cache levels for the considered workloads. When considering all workloads, the MPKIs of L1D, L2C, and LLC are 45.0, 34.1, and 15.6, respectively. On average, 34.7% of L1D misses eventually require a DRAM access. Remarkably, emerging workloads from domains such as graph processing put more pressure on the cache hierarchy, resulting in more frequent DRAM accesses. Indeed, Figure 5.1 reveals that, on average, the graph-processing GAP workloads trigger a DRAM access for 39.7% of the L1D misses.

Finding 1. A large fraction of the demand load requests triggered by applications with large working set sizes miss in all cache levels.

5.3.2 Impact of Hermes

This section quantifies the impact of Hermes on the number of DRAM transactions processed by the main memory in both single-core and multi-core contexts and identifies features that can potentially increase Hermes' efficiency and performance. This analysis is conducted using the methodology and the set of workloads presented in Section 5.5.

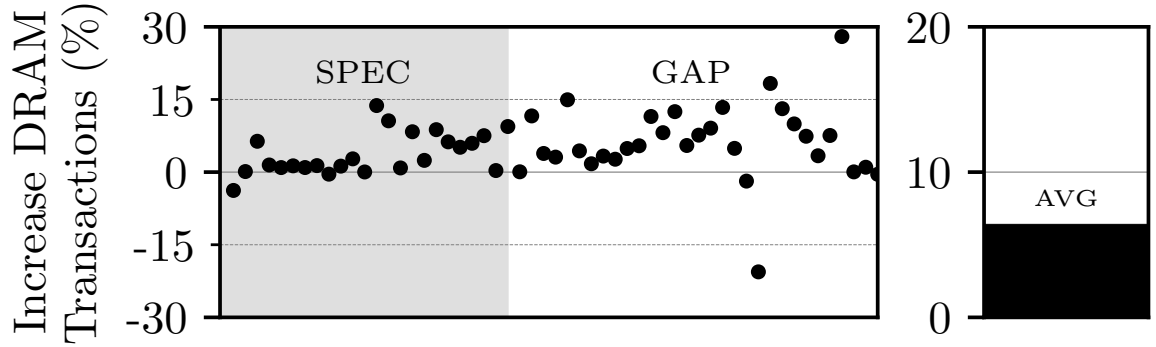


Figure 5.2 Increase in DRAM transactions due to Hermes off-chip predictions relative to a baseline without off-chip prediction mechanism. X-axis ticks represent different SPEC and GAP workloads. Both SPEC and GAP workloads are separately sorted based on LLC MPKI. Lower is better.

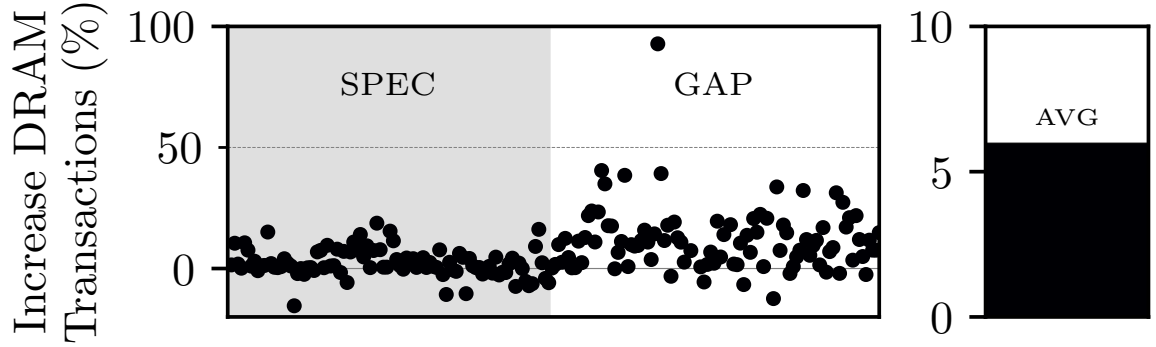


Figure 5.3 Increase in DRAM transactions due to Hermes off-chip predictions relative to a baseline without off-chip prediction mechanism in the 4-core context. X-axis ticks represent 200 different 4-core workload mixes of SPEC and GAP workloads. Both SPEC and GAP workload mixes are separately sorted based on LLC MPKI. Lower is better.

5.3.2.1 DRAM Transactions

Figures 5.2 and 5.3 illustrate the impact of Hermes on the number of DRAM transactions in single-core and multi-core contexts, respectively. These figures show results in the form of a scatter plot for all considered SPEC and GAP workloads on the left-hand side, and average results on the right-hand side. Finally, we compute the above presented metric for Hermes over a baseline that considers the same system but without any off-chip predictor.

Figures 5.2 and 5.3 indicate that Hermes places high pressure on DRAM, especially in the multi-core scenario, due to issuing many speculative DRAM requests. Regarding the single-core evaluation, Hermes increases the number of DRAM transactions by 5.2%, 6.6%, and 6.4% over the baseline system that does not use any off-chip predictor for the SPEC, GAP, and all workloads combined, respectively. Figure 5.3, which presents the impact of

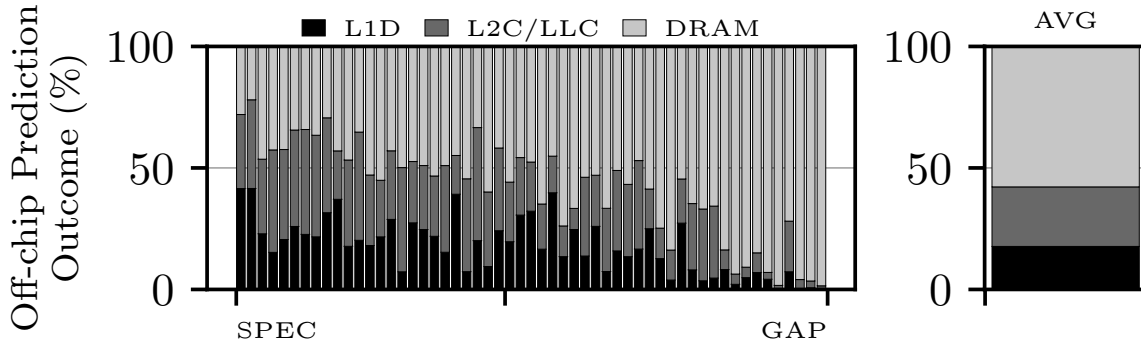


Figure 5.4 Location of a block upon a Hermes off-chip prediction. A large fraction of the inaccurate off-chip predictions could have been avoided if Hermes would wait to issue a speculative DRAM access until the L1D lookup is resolved.

Hermes on DRAM transactions in a multi-core context, shows that Hermes significantly increases DRAM transactions. Specifically, Hermes increases the average number of DRAM transactions by 2.2%, 9.6%, and 6.0% over the multi-core baseline for the SPEC mixes, GAP mixes, and all mixes, respectively. Notably, the increase in DRAM transactions for the GAP workloads is significantly higher than the increase for the SPEC workloads; this happens because the GAP suite is made of graph-processing applications that have much larger data working sets than the general-purpose SPEC CPU workloads.

Finding 2. *Hermes significantly increases the number of DRAM accesses in both single-core and multi-core contexts, especially for graph-processing applications.*

5.3.2.2 Analysis of Hermes Predictions

This section characterizes the off-chip predictions of Hermes, *i.e.*, cases where Hermes triggered a speculative DRAM request, and motivates potential design and functionality enhancements. To do so, we categorize the off-chip predictions of Hermes depending on where the corresponding block is located in the memory hierarchy (L1D, L2C, LLC, DRAM). Specifically, we consider the following categories: (i) block resides in L1D, (ii) block resides in L2C, (iii) block resides in LLC, and (iv) block resides in DRAM. Predictions belonging to categories (i), (ii), and (iii) correspond to inaccurate off-chip predictions since the block is located in the cache hierarchy while category (iv) represents accurate off-chip predictions since the block is not present in the caches. Figure 5.4 presents this breakdown for both SPEC and GAP single-core workloads, using the same methodology as Section 5.5. Note that both SPEC and GAP workloads are separately sorted based on LLC MPKI, similar to Figure 5.2.

Figure 5.4 shows that 42.2% of the total off-chip predictions are inaccurate since the corresponding blocks reside in the cache hierarchy (L1D, L2C, or LLC). Notably, a large fraction of the load requests corresponding to an inaccurate off-chip prediction are served by the L1D cache. Specifically, 17.7% of the total off-chip predictions are useless since their corresponding block resides in the L1D. In other words, delaying Hermes to issue an off-chip prediction after the L1D lookup completion would significantly reduce DRAM transactions. However, constantly delaying the off-chip predictions of Hermes until the L1D lookup is completed would result in suboptimal performance gains since more than 50% (57.8% on average in Figure 5.4) of the Hermes off-chip predictions are accurate. In these cases, issuing the DRAM access before the L1D access is resolved provides latency benefits. Thus, a mechanism to decide whether or not an off-chip prediction of Hermes should be issued before or after the L1D access completion has the potential to significantly reduce the number of useless DRAM accesses triggered by Hermes.

Finding 3. *Selectively delaying Hermes off-chip predictions until the L1D lookup is resolved has the potential to significantly reduce the number of useless DRAM transactions and deliver higher performance.*

5.3.3 Off-Chip Prediction for L1D Prefetch Filtering

This section characterizes the inaccurate prefetches issued by L1D prefetchers across the considered single-core SPEC and GAP workloads. To do so, we consider two state-of-the-art L1D prefetchers: (i) the Instruction Pointer Classification Prefetcher (IPCP) [76], and (ii) the Berti prefetcher [71].

Figure 5.5a presents the breakdown of the inaccurate L1D prefetches issued by IPCP depending on where in the memory hierarchy (L2C, LLC, DRAM) the corresponding prefetch request is served. To do so, we use the Prefetches Per 1000 Instructions (PPKI) metric. Overall, 18.2%, 3.8%, and 78% of the total inaccurate prefetch requests are served by L2C, LLC, and DRAM, respectively. We observe that the majority of the inaccurate prefetch requests are the ones that were served from DRAM. This behavior is more prevalent for the GAP workloads since these workloads have more complex patterns than SPEC. In addition, we compare the accurate L1D prefetches of IPCP that were served from DRAM with the inaccurate ones. Our analysis indicates that, on average, 95.2% of the prefetches that were served from DRAM are inaccurate (the rest 4.8% is accurate prefetches) for our set of workloads; the ratio is higher for the GAP benchmarks (96.7%) than for the SPEC (82%) since the former exhibit more complex memory access patterns. Figure 5.6 indicates how the overall number of accurate prefetchers served from DRAM (1.7 and 0.5 PPKI for IPCP and

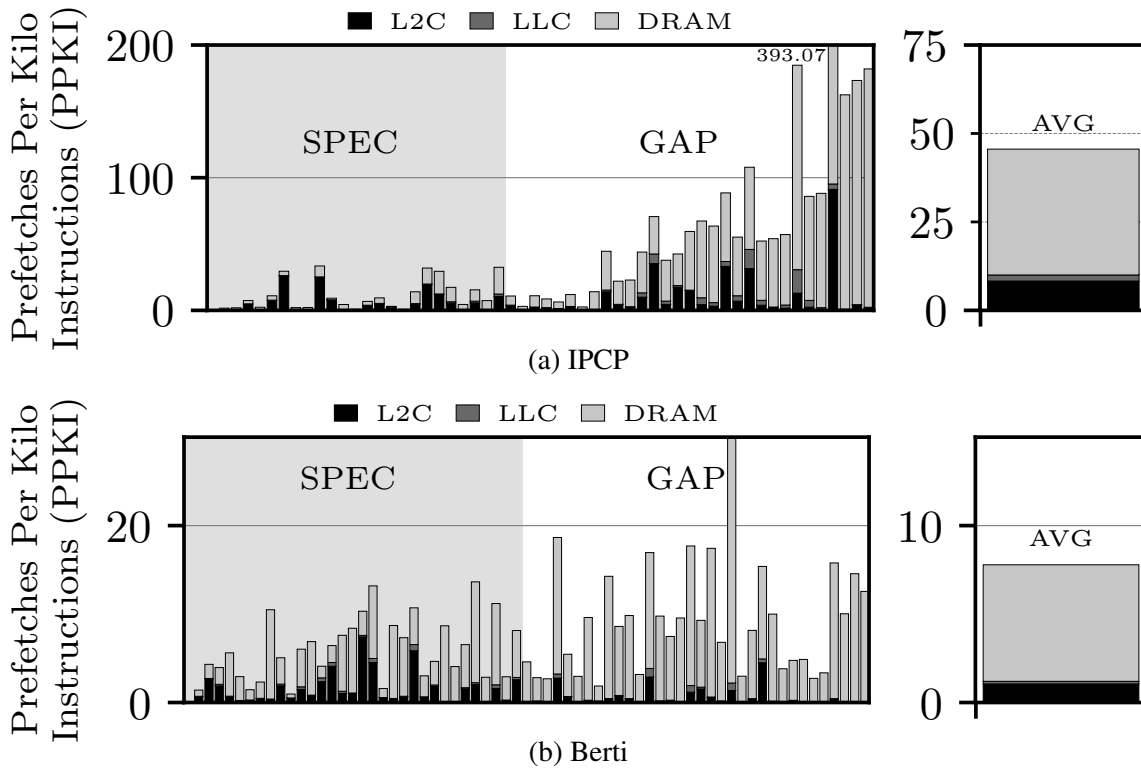


Figure 5.5 Location where the **inaccurate** L1D prefetch requests are served across two state-of-the-art L1D prefetchers. Both SPEC and GAP workloads are separately sorted based on LLC MPKI, similar to Figure 5.2.

Berti) is much smaller than the inaccurate prefetchers served from DRAM (35.6 and 6.6 for IPCP and Berti). Thus, we conclude that accurately predicting whether a prefetch will be served from DRAM can provide a useful hint regarding the usefulness of the corresponding prefetch. Consequently, an accurate off-chip predictor can be leveraged as an L1D prefetcher filter. Finally, we observe similar behavior in the multi-core context.

Figure 5.5b presents the breakdown of Berti’s inaccurate prefetches depending on where in the memory hierarchy the corresponding request is served, similar to Figure 5.5a. Overall, we observe the same behavior. The vast majority of Berti’s useless prefetch requests are served from DRAM and there is high probability for a prefetch that goes all the way to DRAM to fetch a block to be inaccurate, making a strong case for exploiting the off-chip prediction technique to design an L1D prefetch filter.

Finding 4. *Off-chip prediction can be leveraged to design an effective prefetch filtering scheme for L1D.*

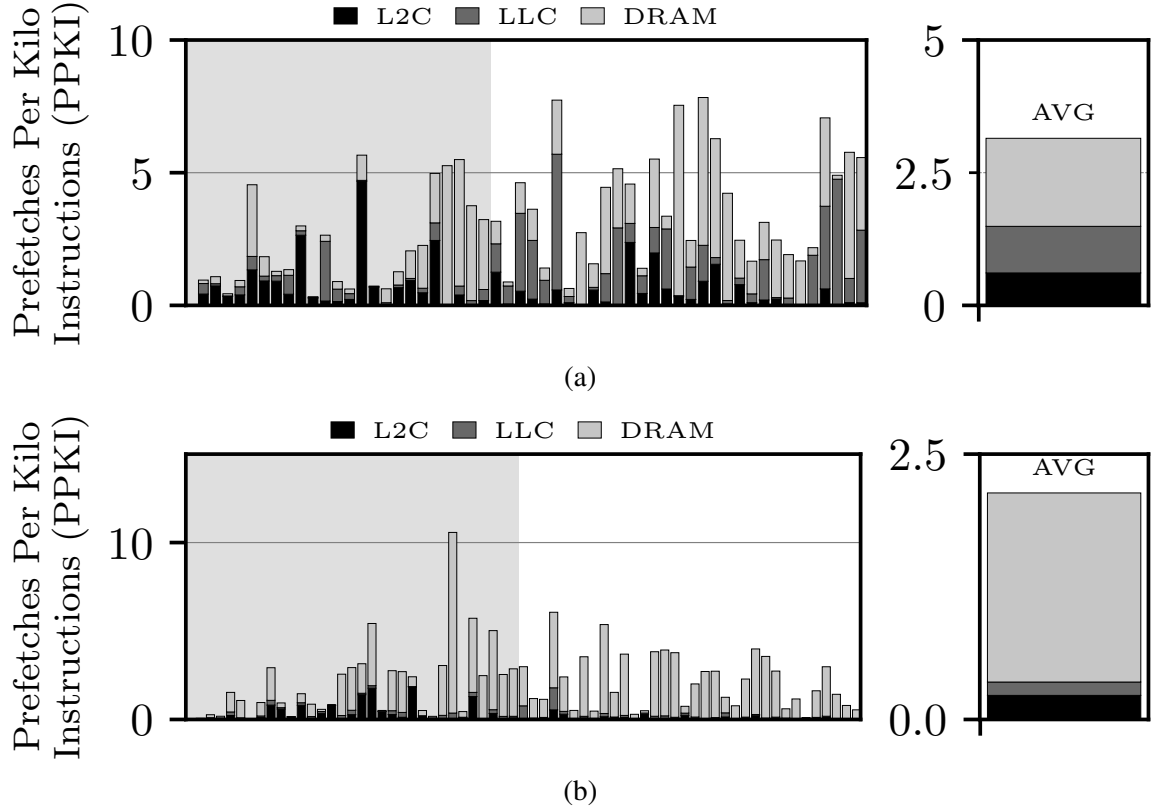


Figure 5.6 Location where the accurate L1D prefetch requests are served across two state-of-the-art L1D prefetchers. Both SPEC and GAP workloads are separately sorted based on LLC MPKI, similar to Figure 5.2.

These four findings demonstrate that the state-of-the-art approach for off-chip prediction incurs a significant overhead in terms of additional DRAM transactions, and that there are opportunities to eliminate this overhead and boost performance by unifying off-chip prediction and prefetch filtering. Section 5.4 presents a novel approach that unifies these two techniques in a single method.

5.4 Two Level Perceptron Prediction

This paper proposes the *Two Level Perceptron (TLP)* predictor, a two level cooperative prediction scheme that leverages neural methods to perform cost-effective off-chip prediction for demand load requests combined with adaptive L1D prefetch filtering. TLP is composed of two microarchitectural perceptron predictors named First Level Perceptron (FLP) predictor and Second Level Perceptron (SLP), respectively. TLP is motivated by the four findings of Section 5.2.

	• $PC \oplus$ cacheline offset • $PC \oplus$ byte offset
<i>Legacy Hermes features</i>	• $PC +$ first access • Cacheline offset + first access • Last-4 load PCs
<i>Leveling feature</i>	• FLP prediction + cache-line offset

Table 5.1 List of features used by the FLP and the SLP.

Findings 1 and 2 demonstrate that Hermes exacerbates the pressure on the memory subsystem that modern workloads inject, particularly for memory intensive workloads from domains like graph-processing. In addition, Finding 3 indicates that a selective delay mechanism can potentially mitigate this pressure. The FLP design, described in Section 5.4.1, is motivated by these three findings. FLP includes a novel selective delay mechanism to only trigger speculative requests to DRAM for highly confident off-chip predictions. Finding 4 indicates the potential of guiding prefetch filtering via off-chip prediction. The SLP design, described in Section 5.4.2, exploits this potential and incorporates a novel feature based on FLP output. Section 5.4.3 presents our complete proposal, TLP, a multi-Level perceptron combining FLP and SLP. TLP is novel in three ways: i) it incorporates a new selective delay mechanism to reduce pressure on the memory subsystem; ii) it leverages off-chip prediction to guide prefetch filtering; and iii) it constitutes the first hardware proposal targeting both off-chip prediction and prefetch filtering. In addition, TLP is the first multi-level perceptron hardware approach that can be effectively applied due to its low area requirements. Finally, Section 5.4.4 details the hardware requirements of TLP.

5.4.1 First Level Perceptron (FLP) Predictor

FLP is an off-chip predictor based on a microarchitectural hashed perceptron predictor that dynamically decides whether to consume the off-chip prediction in the core, *i.e.*, in parallel with the L1D lookup since L1D caches are typically implemented as VIPT structures, or upon an L1D miss. This delayed decision mechanism is driven by two threshold values: τ_{high}

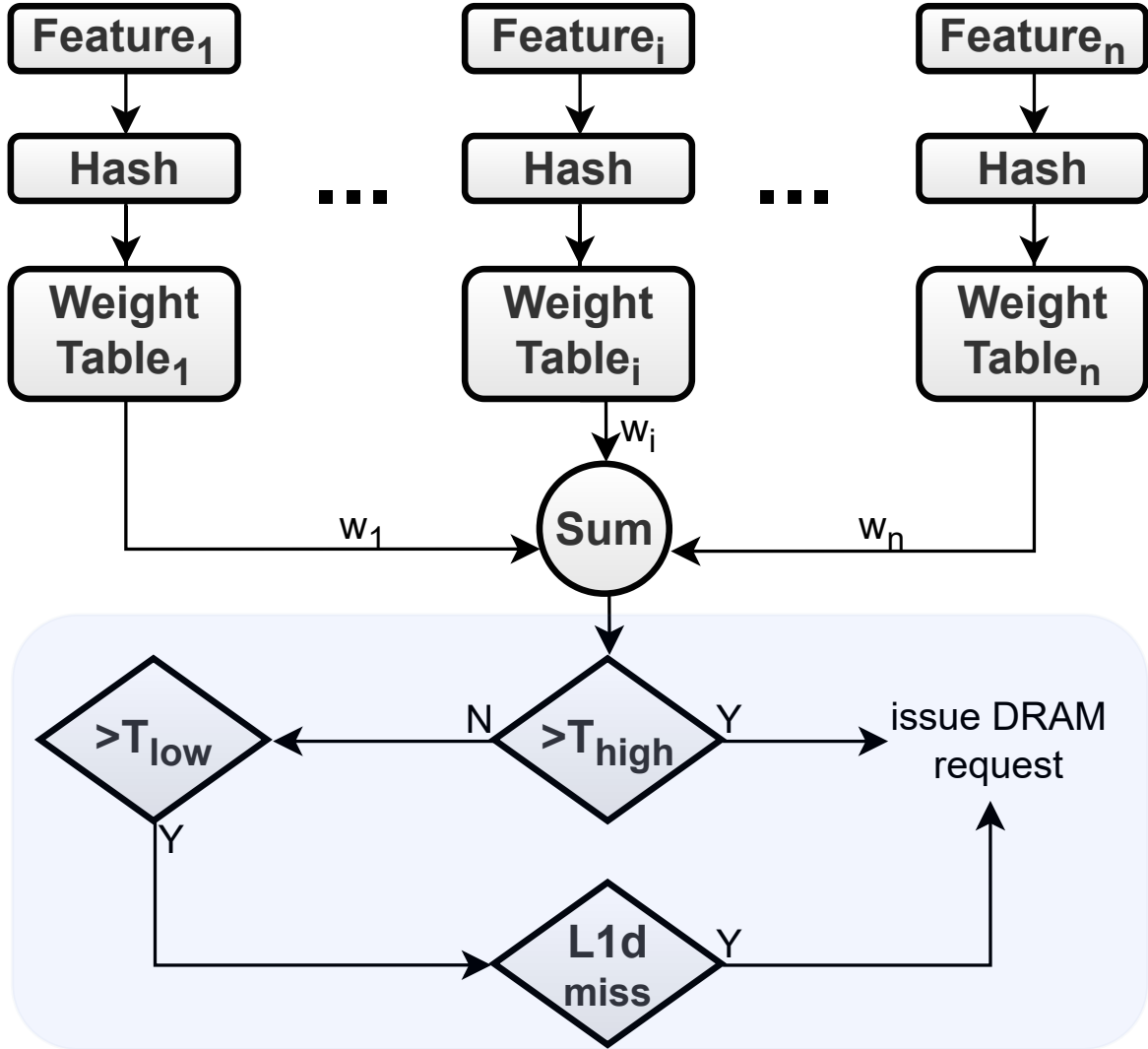


Figure 5.7 Flowchart of FLP. Diamonds indicate decision points.

and τ_{low} . Perceptron confidence values greater than τ_{high} indicate a high probability for the corresponding load request to miss in all cache levels, values lower than τ_{low} indicate the opposite, and intermediate values indicate the need for delaying the decision upon an L1D miss. FLP takes into account several program features to predict whether a demand load request will miss in the cache hierarchy or not. Our exploration indicates that the features used in the original Hermes [19] work provide good predictions and that adding more features provides marginal benefits. Thus, FLP uses the same set of features as the original Hermes prediction, presented in Table 5.1 (*c.f.*: Legacy Hermes features). The selected features correlate the probability of a demand load request going off-chip with a history of PCs and accessed memory regions. Each FLP feature is associated with a weight table which is composed of confidence counters.

Figure 5.7 presents a flowchart of FLP's operation and illustrates how the confidence value produced by FLP is used to drive the off-chip prediction mechanism. Upon a demand load request, FLP is consulted by the core. FLP uses the selected program features to index its weight tables, then reads out and sums the corresponding weights to produce a confidence value. Then, the confidence value is compared to the τ_{high} threshold. A confidence value greater than τ_{high} indicates a high probability for the corresponding load request to miss in all caches. In this case, FLP issues a speculative DRAM request from the core in parallel with the L1D lookup as L1D caches are typically implemented as VIPT structures. However, if the confidence value does not exceed τ_{high} but does exceed the τ_{low} threshold, the probability of the load demand request to miss in all cache levels is not considered high enough to benefit from a speculative DRAM request. Thus, the request is flagged as predicted off-chip and is sent to the L1D cache waiting for a hint that this request might miss in all cache levels. In Section 5.3.1, we observed that the probability of a load demand requiring an access to the DRAM tends to rise with each successive cache level traversed. Thus, if this request results in a miss in the L1D, the flag is read, and a speculative DRAM request is issued from the L1D. Thus, FLP addresses our third analysis finding and avoids sending useless DRAM requests for loads that might hit in the on-chip caches. Finally, if the confidence value exceeds none of the two thresholds, the demand load request continues like a normal request without triggering speculative DRAM access.

The FLP is trained upon completing a memory access, *i.e.*, when the memory block is returned to the core from the cache hierarchy. When the request comes back to the core, the FLP checks if the request was a true off-chip load request, *i.e.*, if this request required a DRAM access. If the request was a true off-chip load request, the predictor's corresponding weights are trained positively. Conversely, if the request was not a true off-chip load request, the predictor's corresponding weights are trained negatively.

5.4.2 Second Level Perceptron (SLP) Predictor

The SLP is a perceptron-based off-chip predictor conceived to be used in the context of L1D prefetch filtering. The SLP design is motivated by the observation that off-chip prediction can be leveraged to design effective L1D prefetch filters. Section 5.3.3 justifies this observation. SLP can be used to improve the performance of any generic L1D prefetcher since it makes no assumption regarding the L1D prefetcher design.

SLP uses several program features to perform effective prefetch filtering at L1D. Our feature exploration indicates that FLP's features can be also used in the context of L1D prefetch filtering. Therefore, SLP uses the same as FLP, presented in Table 5.1, but these features are adapted to use physical addresses in place of virtual addresses as SLP is placed

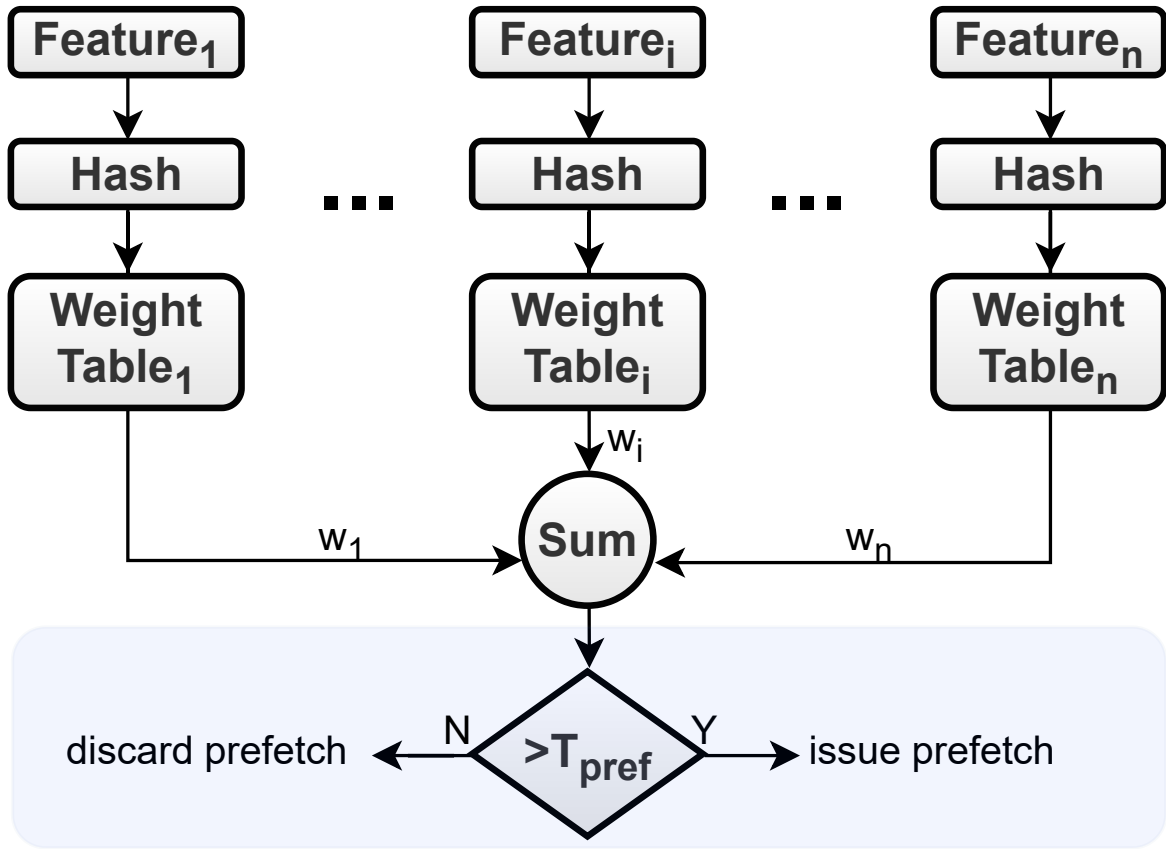


Figure 5.8 Flowchart of SLP. Diamonds indicate decision points.

after the L1D cache. Additionally, SLP makes use of a new feature denoted as FLP prediction + offset in Table 5.1. This feature combines the FLP output bit of the cache block from which the prefetch request originated with the offset of the prefetched cache block in its physical memory page. The rationale of this feature is to correlate the probability of an L1D prefetch request going off-chip when a certain cache line offset is touched with the off-chip prediction decision related to the block that triggered the prefetch request. The SLP produces a binary off-chip prediction when an L1D prefetch request is issued.

Figure 5.8 presents a flowchart of the SLP operation. SLP is consulted when the L1D prefetcher issues a prefetch request. The confidence value is built similarly to the FLP. The output value is compared to the τ_{pref} threshold. If it exceeds τ_{pref} , the prefetch is considered as eventually requiring a DRAM access and, therefore, likely useless. In this situation, the prefetch request is discarded. Conversely, if the confidence value does not exceed τ_{pref} , the prefetch request is processed as usual by the cache hierarchy.

SLP is trained in a similar way as FLP (cf. Section 5.4.1). Upon the completion of an L1D prefetch request, the predictor's weights are trained positively or negatively depending on whether or not the prefetch request was served off-chip.

5.4.3 Building a Multi-Level Perceptron Predictor

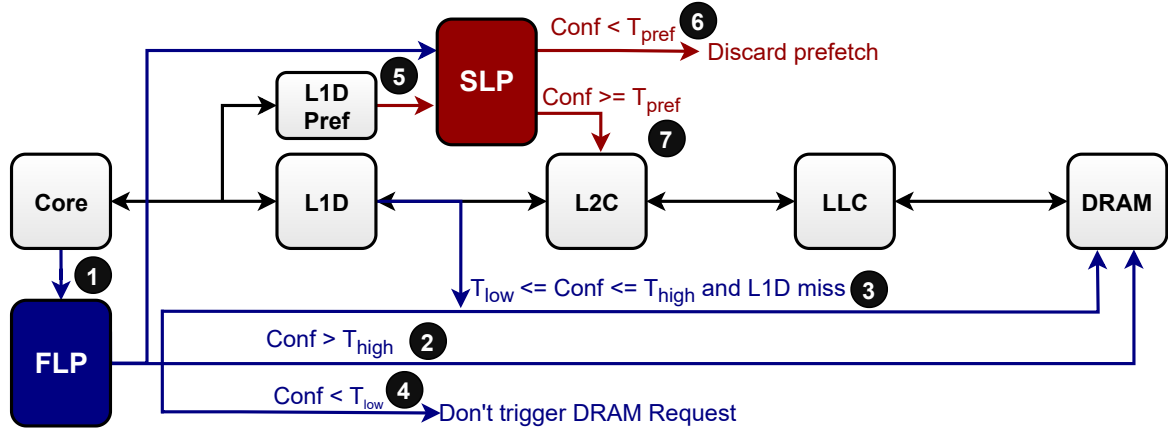


Figure 5.9 Organization and operation of the Two Level Perceptron (TLP) prediction mechanism.

This section presents our complete proposal, Two Level Perceptron (TLP) predictor, a hierarchical neural prediction scheme that combines FLP and SLP predictors, presented in Sections 5.4.1 and 5.4.2, respectively.

Figure 5.9 shows the design and the operation of TLP. Upon a load demand access, the core consults FLP to obtain a confidence value $Conf$ driving the off-chip prediction ①. This prediction can give one of the three following outcomes: i) the load request is predicted to be off-chip with high confidence ($Conf > \tau_{high}$), thus a speculative DRAM request is thrown from the core ② besides the regular load demand access; ii) the load request is predicted to be off-chip with low confidence ($\tau_{low} \leq Conf \leq \tau_{high}$), thus the speculative DRAM request will be thrown only if the load misses in the L1D ③; iii) the load request is predicted to be on-chip; therefore no additional action is taken ④ besides triggering the regular demand access. Metadata relative to the prediction (hashed PC, history of last load PCs, and perceptron confidence value) are stored in the matching Load Queue entry for later training and an off-chip prediction tag is set in the load request thrown to the cache hierarchy depending on the FLP prediction.

SLP is consulted upon L1D prefetch requests ⑤. To make a prediction, SLP takes as input the metadata attached to the prefetch request and the off-chip prediction tag attached to the demand load request from which the prefetch request originates. This information is used to produce an off-chip $Conf$ prediction specific to L1D prefetch request. This prediction can result in two possible outcomes: i) the prefetch request is predicted to be off-chip ($Conf \leq \tau_{pref}$) and the prefetch request is discarded ⑥, and ii) the prefetch request is predicted to be on-chip ($Conf > \tau_{pref}$) and the prefetch request is processed as usual by

Component	Description	Size
<i>FLP</i>	• Perceptron weight tables: 2.58KB • Page buffer: 0.63KB	3.21KB
<i>SLP</i>	• Perceptron weight tables: 2.66KB • Page buffer: 0.63KB	3.29KB
<i>Load Queue metadata</i>	Hashed PC: 32b; Last-4 PC: 10b; First access: 1b; perceptron confidence value: 5b	0.42KB
<i>L1D MSHR metadata</i>	Hashed PC: 32b; Last-4 PC: 10b; First access: 1b; perceptron confidence value: 5b; prediction: 1b	0.06KB
<i>Total</i>		6.98KB

Table 5.2 Storage overhead of TLP.

the cache hierarchy ⑦. Similarly to FLP, SLP stores metadata relative to its prediction in the L1D MSR entries for later training.

The training routines of the FLP and the SLP are triggered upon completion of the corresponding requests, *i.e.* for FLP when the load request returns to the core and SLP when the prefetch request is served, as Sections 5.4.1 and 5.4.2 explain.

5.4.4 TLP Hardware Requirements and Latency

Table 5.2 breaks down the hardware requirements of TLP into its various components. TLP only requires 6.98KB of additional storage per core. Similar to Hermes [19], FLP requires 3.21KB of storage for its prediction tables and 0.42KB of storage for the metadata in the Load Queue entries for training purposes. SLP requires 3.29KB of storage for its prediction tables as we make the addition of a new feature, and 0.06KB of additional storage in the L1D MSHR entries for training purposes. In total, TLP requires only 6.99KB of extra storage, making it a low overhead design combining off-chip prediction and prefetch filtering. Similarly to previous work [19], we consider a 6-cycles latency when either FLP or SLP trigger a speculative DRAM access.

5.5 Experimental Methodology

5.5.1 Simulation Methodology

We evaluate our proposal using ChampSim [38], a detailed trace-based simulator that models a 4-wide out-of-order CPU. We consider a baseline system similar to the Intel Cascade Lake microarchitecture [26]. Table 5.3 presents the specific configuration details of the baseline system. Regarding hardware prefetching, we use state-of-the-art prefetchers in both the L1D and the L2C. At the L1D level we consider both the Instruction Pointer Classification Prefetcher (IPCP) prefetcher [76] and the Berti prefetcher [71]. At the L2 level we use the SPP prefetcher [61], which brings prefetched blocks into either the L2C or the LLC depending on its internal prefetch logic.

5.5.2 Workloads

Our evaluation considers a large set of applications spanning different benchmark suites. Specifically, we consider workloads from SPEC CPU 2006 [91] and SPEC CPU 2017 [92] benchmark suites. In addition, we consider graph-processing applications included in the GAP benchmark suite [15]. Specifically, we use six graph-processing kernels from GAP: Breadth-First Search (BFS) is a fundamental graph traversal algorithm; Page Rank (PR) iteratively updates per-vertex ranks until convergence; Connected Components (CC) applies the Shiloach-Vishkin [88] algorithm to compute the largest connected components of the graph; Betweenness Centrality (BC) uses the Brandes algorithm [23] to approximate the per-vertex centrality scores; Triangle Count (TC) counts the number of triangles in the graph; and, finally, Single-source Shortest Paths (SSSP) uses δ -stepping [66] to return the distance of all vertices of a graph to a given source vertex. Table 5.4 shows the main characteristics of these six applications, including the size of property array elements, and input parameters such as the execution style (push or pull), or the use of frontiers.

For each graph-processing kernel, we consider 6 different input graphs that feature different sizes and distributions of node degrees (*e.g.*, power-law, normal, etc.). Different degree distributions produce different memory access patterns. For instance, when node degrees are distributed following a power-law function, there are a few highly connected graph nodes that yield more data reuse opportunities than vertices with a few connections. Table 5.5 lists all considered input graphs.

In addition, we only consider workloads for which the baseline system shows LLC MPKI greater than 1. This filters out workloads and leaves us with 31 GAP workloads and 24 SPEC workloads.

Component	Description
Branch Predictor	hashed-perceptron [51]
CPU	3.8 GHz, 4-wide out-of-order processor 6-stage pipeline, 224-entries re-order buffer
L1 ITLB	64-entry, 4-way, 1cc, 8-entry MSHR, LRU
L1 DTLB	64-entry, 4-way, 1cc, 8-entry MSHR, LRU
L2 TLB	1536-entry, 12-way, 8cc, 16-entry MSHR, LRU
L1I Cache	32 k, 8-way, 4cc, 10-entry MSHR, LRU
L1D Cache	32 k, 8-way, 4cc, 10-entry MSHR, LRU, IPCP[76] or Berti[71]
L2 Cache	1 M, 16-way, 10cc, 16-entry MSHR, LRU, SPP [61]
LLC	1.375 M per core, 11-way, 36/56cc, 64-entry MSHR, LRU
DRAM	16 G, DDR4 SDRAM single-core data-rate: 12.8 GB/s per core multi-core data-rate: 3.2 GB/s per core $t_{RP} = t_{RCD} = t_{CAS} = 24$ cycles

Table 5.3 System configuration.

All workload traces have been obtained using the SimPoint methodology [79] to identify at least one SimPoint representative of each workload. Each SimPoint is 1 billion instructions long and characterizes a different phase of these workloads, similar to prior work [55, 87, 59, 100].

Section 5.6 refers to SPEC 2006 and SPEC 2017 workloads as SPEC and to GAP workloads as GAP.

	BC [15]	BFS [15]	CC [15]	PR [15]	TC [15]	SSSP [15]
irregData ElemSz	8 + 4	4	4	4	4	4
Execution style	Push-Mostly	Push & Pull	Push-Mostly	Pull-Only	Push-Only	Push-Only
Use Frontier	Yes	Yes	No	No	No	Yes

Table 5.4 Graph kernels

5.5.3 Single-Core Evaluation

Our set of single-core workloads contains 55 distinct workloads: 31 possible combinations of graph-processing kernels and input graphs, described in Section 5.5.2, and 24 SPEC CPU

	Web[15]	Road[15]	Twitter[15]	Kron[15]	Urand[15]	Friendster[108]
# Vertices (in M)	50.6	23.9	61.6	134.2	134.2	65.6
# Edges (in M)	1,949.4	58.3	1,468.4	2,111.6	2,147.4	3,612.1

Table 5.5 Input Graphs

2006 [91] and SPEC CPU 2017 [92] benchmarks. All considered workloads experience at least 1 Miss per Kilo Instructions (MPKI) in the baseline system that Table 5.3 describes. Each workload is executed for 100 million instructions to warm up the memory hierarchy and the other microarchitectural structures, and it is executed for an additional set of 100 million instructions to obtain performance data. We run experiments evaluating the impact of using larger numbers of instructions (500 million warmup instructions, 1 billion simulation instructions), and observe identical trends with negligible differences in terms of IPC.

5.5.4 Multi-Core Evaluation

We generate multi-core workload mixes using the same methodology as previous work [19]. We consider either single-core GAP workloads or single-core SPEC workloads to create both homogeneous and heterogeneous multi-core workload mixes. To generate the homogeneous ones, we randomly select 50 single-core workloads and run four instances of each workload, one per core. For the heterogeneous mixes, we randomly select 50 combinations of four single-core workloads. In total, we consider 50 homogeneous and 50 heterogeneous four-core workloads. We do this process for both SPEC and GAP benchmark suites, meaning that our multi-core evaluation campaign is composed of 200 workloads. Finally, our multi-core experiments use the same number of warmup and simulation instructions as the single-core scenario.

Our performance results concerning multi-core workloads report the weighted speedup normalized to the baseline. This metric is commonly used to evaluate multi-core workloads [55, 87, 101] since it avoids performance overestimation due to high-IPC threads. The metric is computed as follows: for each single-core workload, we compute its IPC in a multi-core scenario shared with the other co-running single-core workloads (IPC_{shared}), and its IPC running in isolation on the same system (IPC_{single}). We then compute the weighted IPC of the mix as the weighted sum of $IPC_{shared}/IPC_{single}$ for all the benchmarks in the mix, and we normalize this weighted IPC with the weighted IPC of the baseline design.

5.5.5 Alternative Techniques

Besides TLP, we consider the following techniques in our evaluation: i) the Perceptron-based Prefetch Filtering (PPF) [20], a perceptron-based predictor that filters inaccurate prefetch requests, thus increasing the accuracy of the underlying prefetcher. PPF is located at the L2C since it is built on top of the SPP prefetcher. When using PPF, we configure SPP as previous work indicates [20] to fully exploit the advantages of PPF. ii) Hermes [19], the state-of-the-art off-chip predictor that removes long-lasting load requests from the critical path by issuing speculative requests to the DRAM controller. iii) Hermes+PPF, a scheme that uses both Hermes as off-chip predictor and PPF as a prefetch filter.

5.6 Evaluation

5.6.1 Single-Core Evaluation

This section evaluates TLP in the single-core context following the methodology that Section 5.5 presents. Figure 5.10 shows the performance gains provided by PPF, Hermes, Hermes+PPF, and TLP in the single-core context over a baseline described in Section 5.5, which has no off-chip predictor neither L1D prefetch filter. Specifically, Figures 5.10a and 5.10b present performance results when using IPCP and Berti as L1D prefetchers, respectively. Both figures display speedup with respect to the baseline system in the y-axis. The x-axis shows the selected SPEC and GAP workloads. For each benchmark suite we sort workloads in increasing order of MPKI in the baseline system. This evaluation indicates that TLP significantly outperforms state-of-the-art approaches for off-chip prediction (Hermes), prefetch filtering (PPF), and a combination of them (Hermes+PPF). In the scenario considering IPCP as L1D prefetcher, TLP yields 6.2% geometric mean speedup with respect to the baseline system while PPF, Hermes, and Hermes+PPF bring -0.2%, 5.2%, and 4.7% geometric mean speedups, respectively. When considering the Berti prefetcher, TLP yields 8.1% geometric mean speedup as compared to 1.7%, 4.8%, and 6.1% for PPF, Hermes, and Hermes+PPF, respectively. TLP achieves larger performance gains for GAP than SPEC. Since GAP workloads are strongly memory bound, the reductions in terms of DRAM transactions that TLP achieves compared to Hermes particularly benefit GAP.

To identify the source of TLP performance improvements we quantify the impact of PPF, Hermes, Hermes+PPF, and TLP in terms of number of DRAM accesses. Figure 5.11 shows this evaluation. Specifically, Figure 5.11a presents results obtained using IPCP, while Figure 5.11b considers Berti. The y-axis displays the increase in DRAM transactions processed by the memory controller in the single-core context while the x-axis shows the

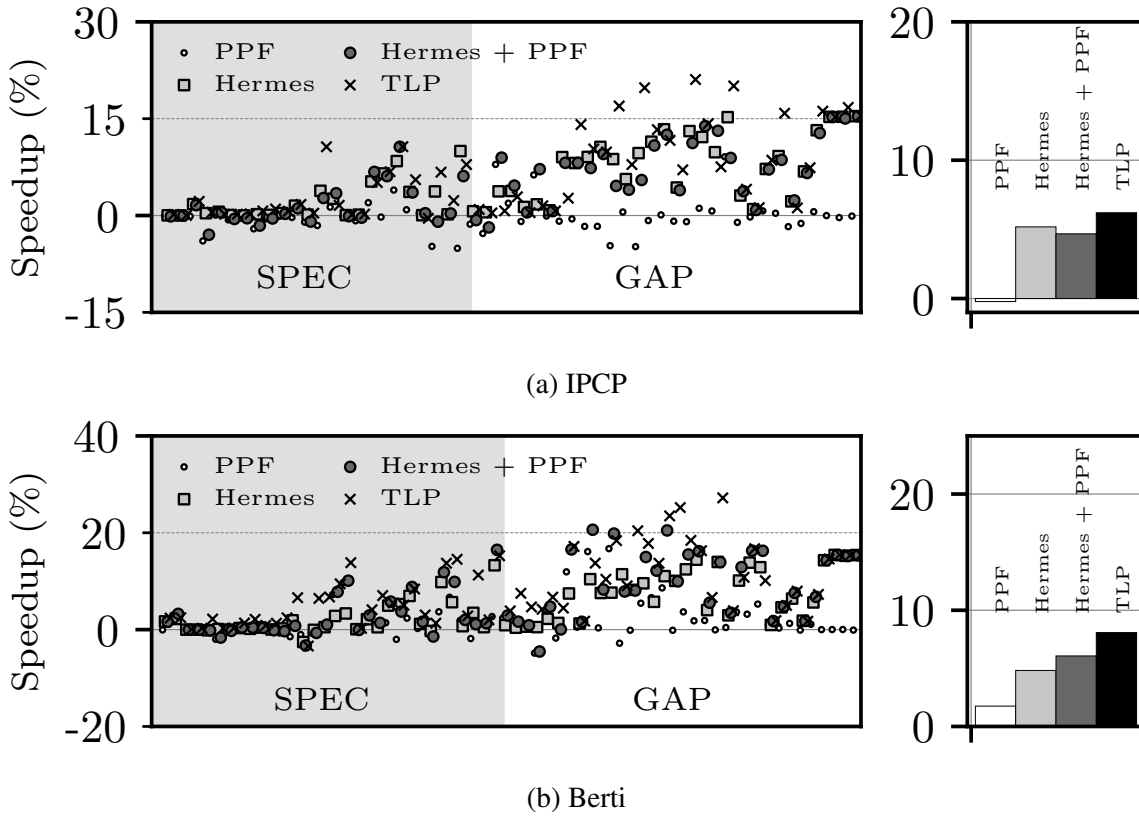


Figure 5.10 Performance evaluation in the single-core scenario.

SPEC and GAP workloads sorted in terms of MPKI. When using IPCP, TLP reduces DRAM transactions by an average of 30.7% over the baseline while PPF, Hermes, and Hermes+PPF increase average DRAM transactions by 7.7%, 5.2%, and 13.3%, respectively. When considering Berti, TLP reduces the number of DRAM transactions by an average of 14.2% over the baseline, while PPF, Hermes, and Hermes+PPF trigger of 8.8%, 9.6%, and 16.9% additional DRAM transactions, respectively.

To further explain the performance gain obtained by TLP, we evaluate the accuracy of the considered L1D prefetchers (IPCP and Berti) when PPF, Hermes, Hermes+PPF, and TLP operate in the system. Figure 5.12 presents the results. Specifically, Figures 5.12a and 5.12b present the accuracy of the IPCP and Berti prefetchers, respectively. The key takeaway of this comparison is that TLP increases the accuracy of the L1D prefetchers. Across all SPEC and GAP workloads, IPCP experiences an average accuracy of 20.6%, 20.6%, 20.3%, and 38.0% when PPF, Hermes, Hermes+PPF, and TLP operate in the system, respectively. Finally, we observe similar behavior for the Berti prefetcher; Figure 5.12a reveals that Berti experiences the highest accuracy with TLP.

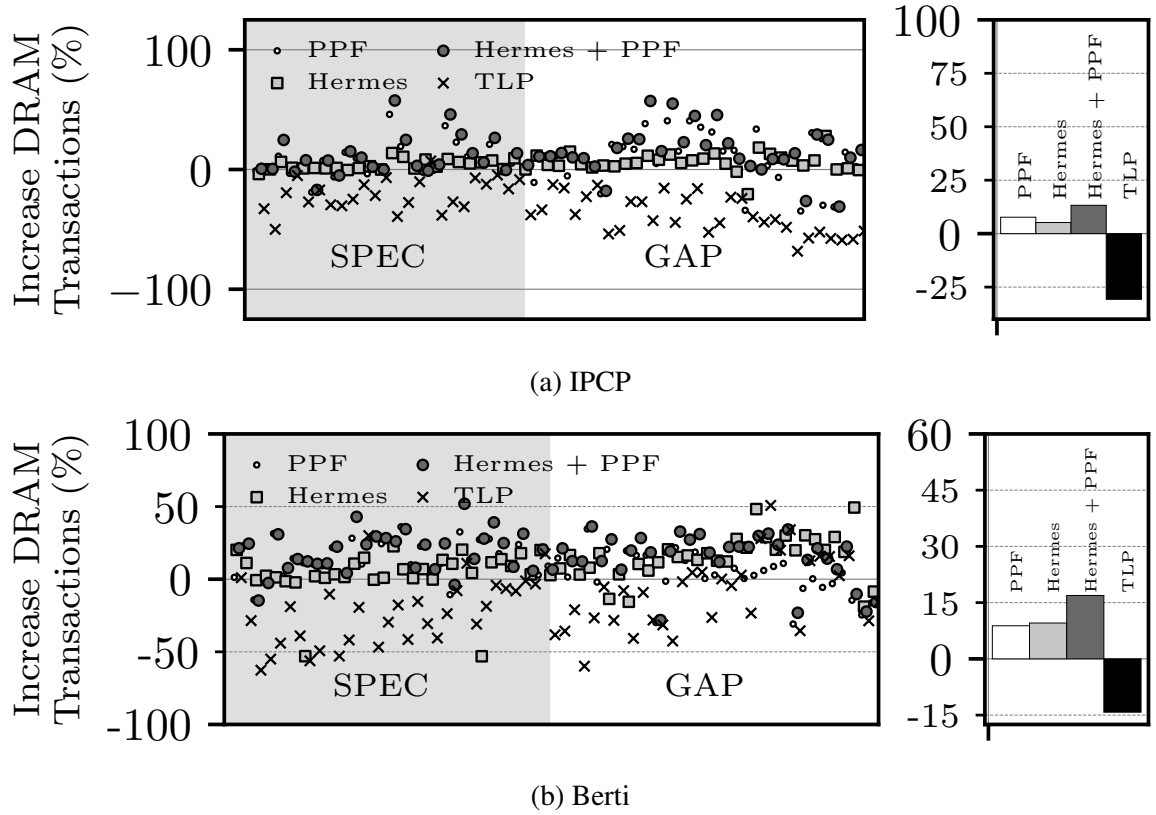


Figure 5.11 Increase in DRAM transactions in the single-core scenario. Lower is better.

Data in Figures 5.11 and 5.12 indicate that TLP successfully reduces the number of DRAM transactions that state-of-the-art off-chip prediction and prefetch filtering approaches trigger.

5.6.2 Multi-Core Evaluation

This section evaluates the performance of TLP in the multi-core scenario following the methodology that Section 5.5 presents. In addition, this section indicates the contribution of each specific TLP component to final performance (Section 5.6.2.1), and evaluates TLP considering different DRAM bandwidth scenarios (Section 5.6.2.2).

Figure 5.13 shows the performance gains provided by PPF, Hermes, Hermes+PPF, and TLP in the multi-core context over the baseline system that Section 5.5 describes. Specifically, Figures 5.13a and 5.13b present performance results considering IPCP and Berti, respectively. Both figures display speedup over the baseline system in the y-axis. The x-axis shows the multi-core SPEC and GAP workloads sorted in increasing order in terms of MPKI. The sorting is done independently within each benchmark suite. Considering the IPCP prefetcher,

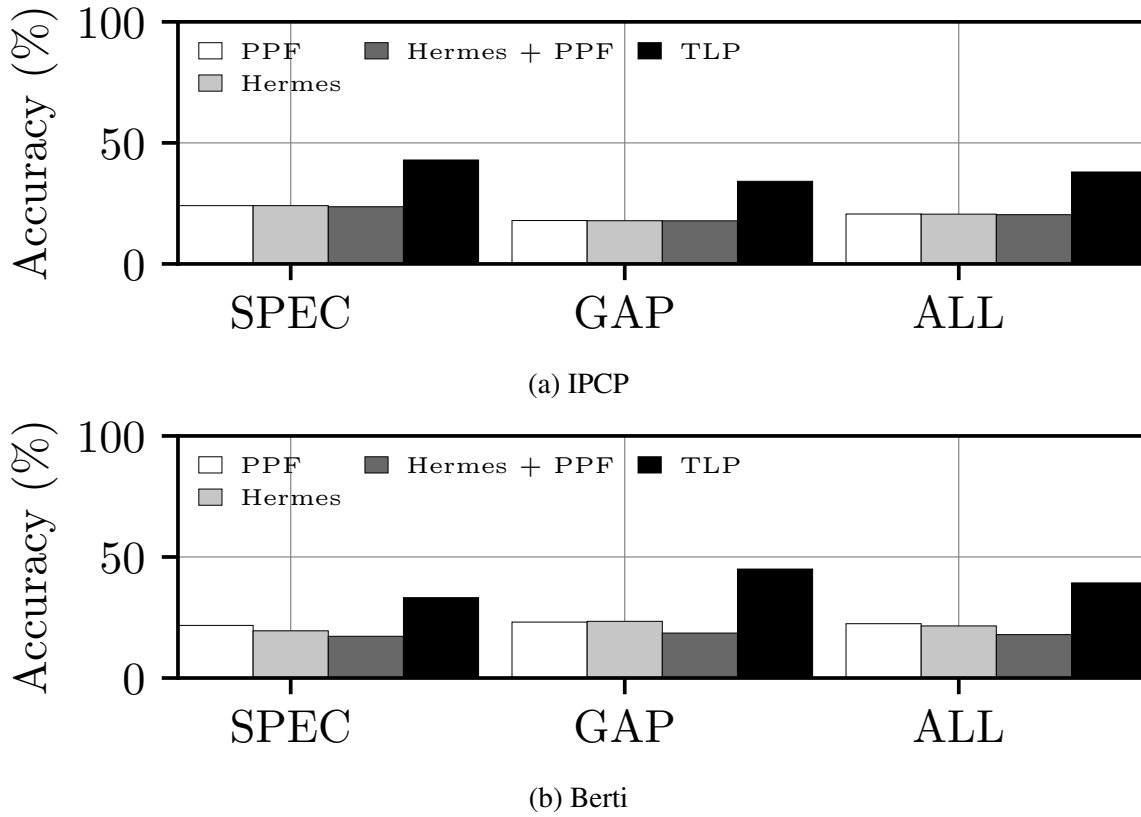


Figure 5.12 Accuracy of the L1D prefetchers.

TLP improves geometric mean performance by 11.5% as compared to -3.3%, 3.0%, and -0.5% for PPF, Hermes, and Hermes+PPF, respectively. When considering Berti as L1D prefetcher we observe similar trends. Specifically, TLP yields a 11.8% geometric mean speedup over the baseline as compared to -1.5%, 1.0%, and -0.3% for PPF, Hermes, and Hermes+PPF, respectively. The main takeaway of this experiment is that TLP provides significantly higher multi-core performance than all considered prior proposals.

To explain the source of TLP performance improvements in the multi-core context, we quantify the impact of PPF, Hermes, Hermes+PPF, and TLP on the number of DRAM accesses. Figure 5.14 shows the increase in terms of DRAM transactions over the baseline system that Section 5.5 describes. Figures 5.14a and Figure 5.14b shows the impact on DRAM transactions when IPCP and Berti operate at L1D, respectively. Considering the IPCP prefetcher, TLP reduces the number of DRAM transactions by an average of 17.7% over the baseline while PPF, Hermes, and Hermes+PPF increase DRAM transactions by 6.5%, 6.0%, and 13.4%, respectively. We observe a similar behavior when Berti is used as L1D prefetcher: TLP reduces the average DRAM transactions by 6.3% while PPF, Hermes, and Hermes+PPF increase DRAM transactions by 9.8%, 1.4%, and 7.8%, respectively. The main takeaway

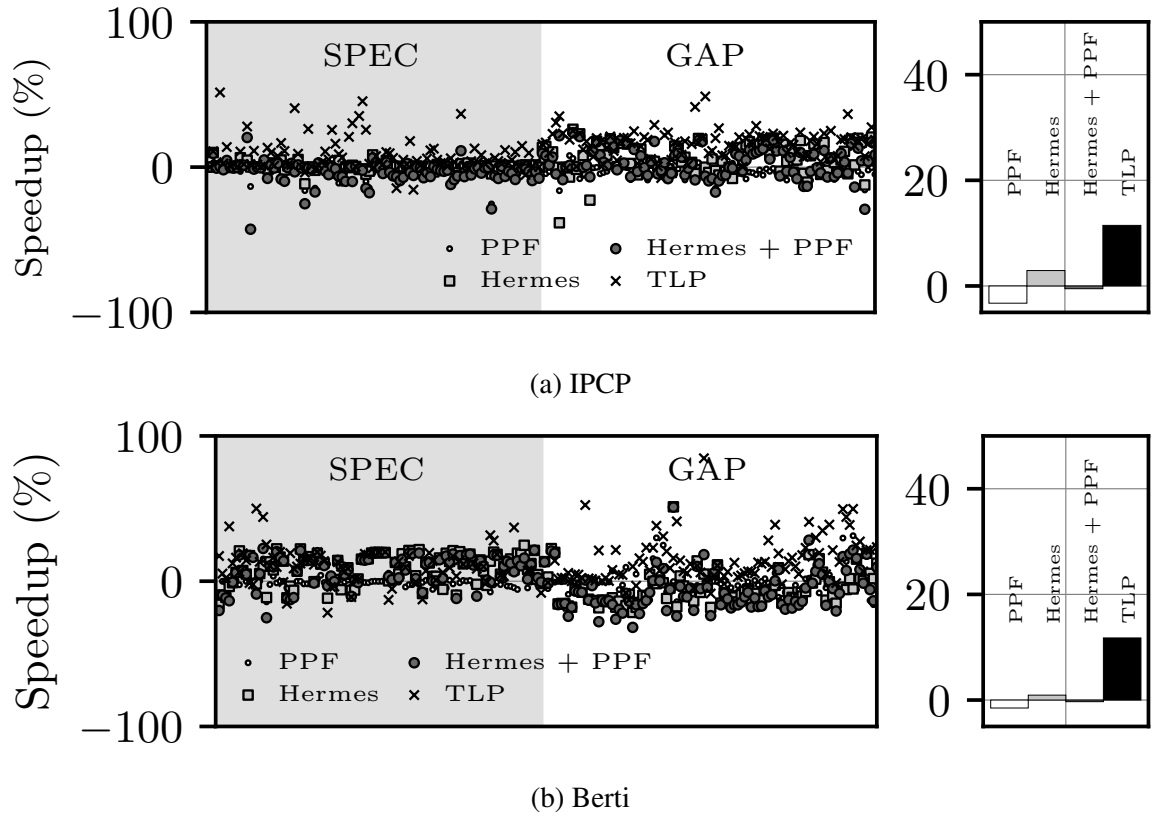


Figure 5.13 Performance improvement in the multi-core scenario.

is that TLP outperforms prior approaches for off-chip prediction and prefetch filtering in multi-core contexts since it significantly reduces the DRAM pressure.

5.6.2.1 Performance Contribution of Each TLP Component

This section evaluates the contribution of each specific TLP component to the final performance. To do so, we consider five different scenarios besides TLP: i) FLP, which consists of just the FLP predictor without the selective delay mechanism. ii) SLP, which consists of just the SLP predictor. iii) Two-Step Predictor (TSP), which consists of FLP without the selective delay mechanism, and SLP without the feature based on FLP output. TSP consumes FLP predictions before the completion of L1D accesses, as Hermes does. Therefore, the difference between TSP and Hermes is the use of SLP. iv) Delayed TSP, a technique similar to TSP with the exception that always delays the consumption of FLP predictions upon L1D misses. v) Selective TSP, an evolution of Delayed TSP that uses selective delay. Finally, we consider TLP. The difference between TLP and Selective TSP is that TLP uses a feature based on the output of FLP to drive the predictions of SLP.

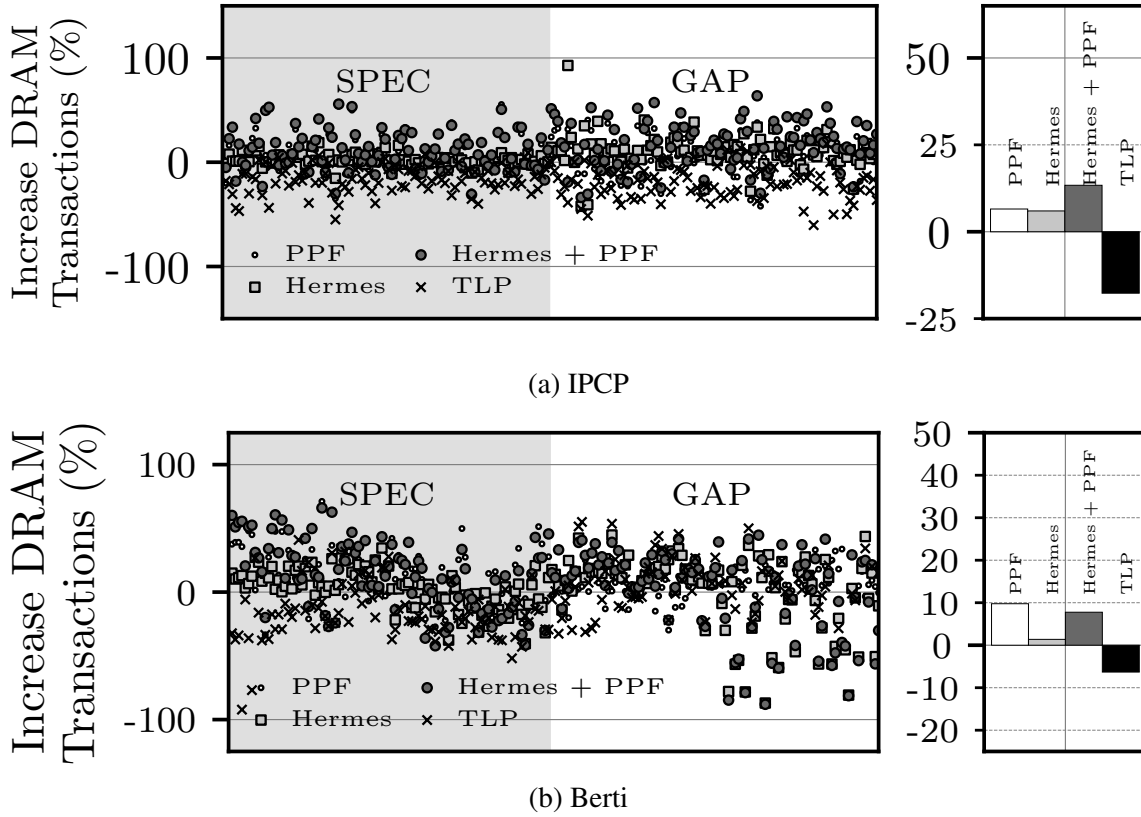


Figure 5.14 Increase in DRAM transactions in the multi-core scenario. Lower is better.

Figure 5.15 shows the performance of these six approaches when IPCP operates at L1D. We observe that by incrementally adding parts of our design to form our final proposal, TLP, we can compound performance as FLP, SLP, TSP, Delayed TSP, and Selective TSP yield respectively 2.9%, 6.9%, 8.4%, 10.2%, and 11.4% geometric mean speedups over the baseline. Our final proposal, TLP provides a 11.5% speedup over the baseline, justifying our design choices. Although the difference between TLP and Selective TSP in terms of geometric mean speed-ups is rather small, it becomes larger for workloads with high correlation between off-chip load demand requests and off-chip L1D prefetch requests like `bc.road`. We observe a very similar behavior when we consider Berti as L1D prefetcher.

5.6.2.2 Sensitivity Analysis on DRAM Bandwidth

This section evaluates TLP, PPF, Hermes, and Hermes+PPF on scenarios with 1.6 GB/s per core up to 25.6 GB/s per core. Figures 5.16a and 5.16b show the geometric mean performance and the impact on DRAM accesses of TLP of these four approaches in the multi-core context, respectively.

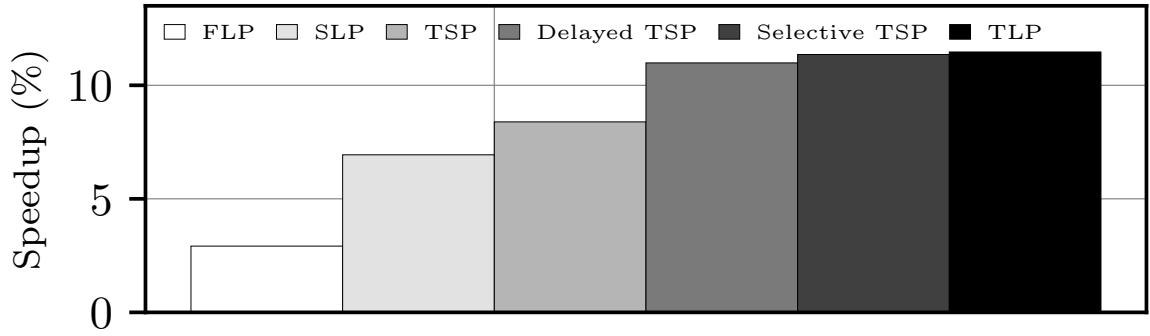


Figure 5.15 Performance contribution of each TLP component.

Figure 5.16a indicates that TLP outperforms the other approaches under memory bandwidth regimes between 1.6 and 6.4 GB/s per core. The improvement achieved by TLP in the 1.6 GB/s scenario is 21.2%, while TLP obtains a 6.9% geometric mean speedup over the baseline when there are 25.6 GB/s per core available. Even in scenarios with unrealistically large memory bandwidth per core ratios (*e.g.* 12.8 or 25.8 GB/s per core), TLP outperforms Hermes and PPF since it avoids cache pollution due to inaccurate prefetching. In these scenarios, the unrealistic abundance of memory bandwidth allows Hermes+TLP to deliver larger performance than TLP.

Figure 5.16b shows the impact in terms of DRAM transactions for all considered approaches on five memory bandwidth per core scenarios. TLP achieves a remarkable reduction in terms of DRAM transactions over the baseline in all scenarios. Specifically, TLP decreases DRAM transactions from 24.8% (1.6 GB/s core scenario) to 17.6% (25.6 GB/s per core scenario) as compared to the baseline.

5.6.3 Designs Enhanced with TLP’s Storage Budget

In addition to the results provided in the previous sections, we also evaluate other designs leveraging 7KB of extra storage over IPCP, Berti, and Hermes. We compare them to TLP. Figure 5.17 evaluates these designs, in both single-core and multi-core contexts. In the single-core context, adding 7KB of extra storage to IPCP and Berti does not leverage any performance benefits over the baseline. When Hermes is enhanced with 7KB of extra storage it provides performance improvement similar to a classic Hermes design, about 5.2% and 4.8% geometric mean speedup. In comparison TLP leverages 6.2% and 8.1% geometric mean speedup for respectively IPCP and Berti. In the multi-core context, we observe a similar behavior where adding extra storage to the prefetchers does not leverage performance

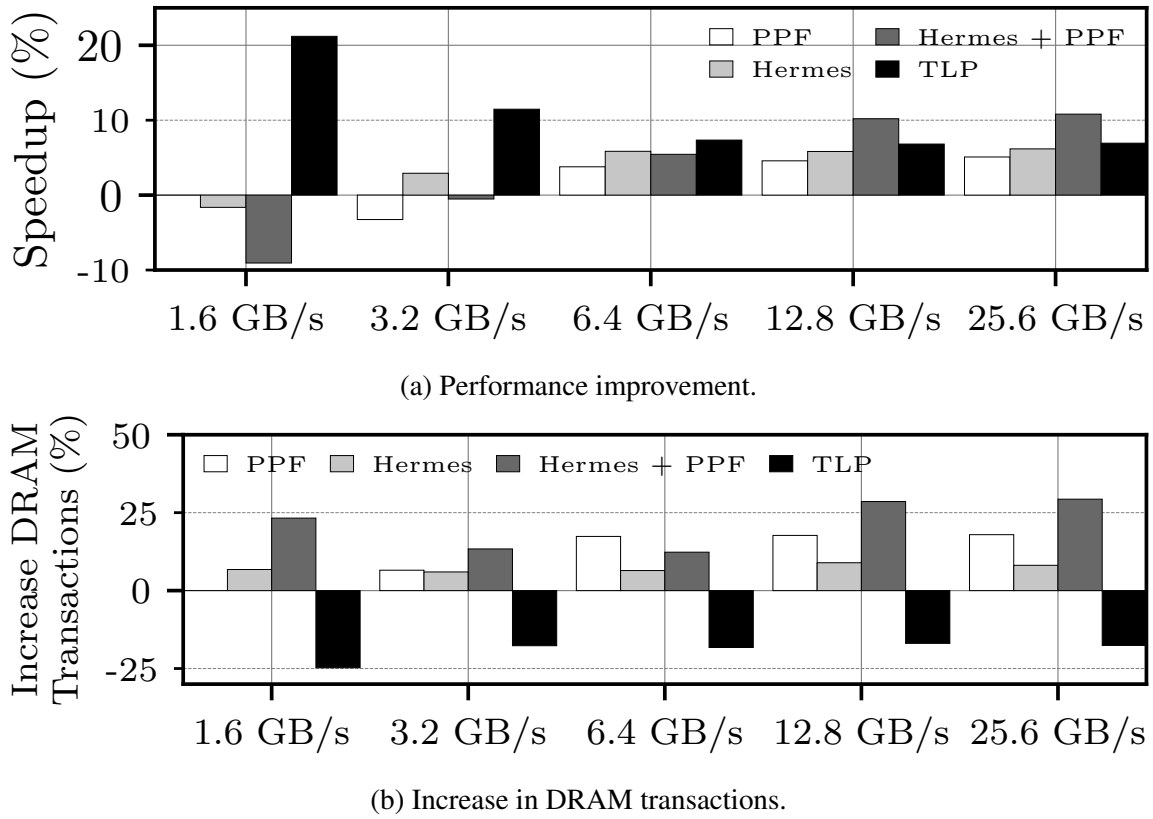


Figure 5.16 Impact of DRAM bandwidth on geometric mean performance and average number of DRAM transactions in the multi-core context.

improvements. Finally, We observe that Hermes shows a similar behavior as its counterpart using no extra storage.

5.7 Related Work

To the best of our knowledge, this is the first work to provide a cooperative solution for off-chip prediction and adaptive prefetch filtering using neural methods. Sections 5.2, 5.3.2, and 5.6 describe, analyze and compare our proposal against Hermes [19], the state-of-the-art off-chip predictor, respectively. Sections 5.2 and 5.6 compare our proposal against PPF [20], the state-of-the-art prefetch filter. This section focuses on other related work targeting memory hierarchy optimizations.

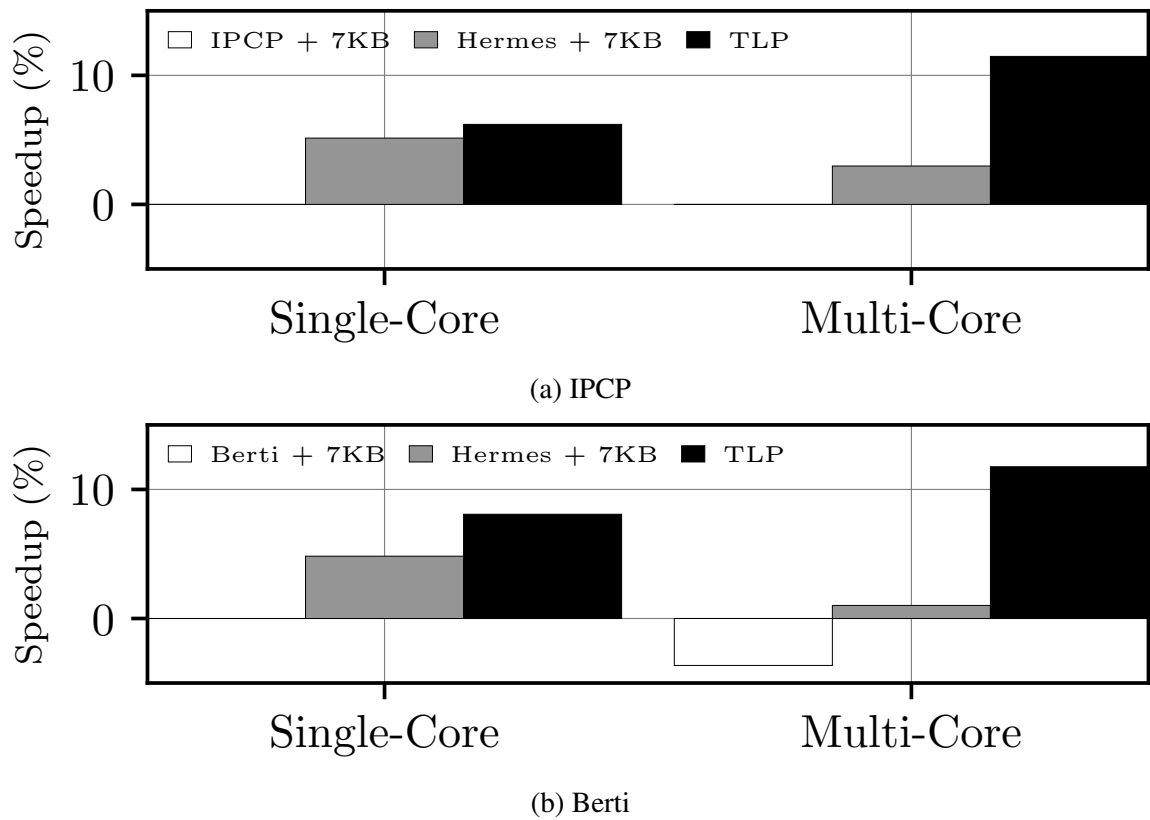


Figure 5.17 Performance improvement of designs enhanced with TLP's storage budget.

5.7.1 Hit/Miss Prediction

Jalili and Erez [48] proposed *LP*, a scheme that uses a flat-array to track the residency of cache lines in the cache hierarchy. This flat-array is stored in a reserved section of DRAM, and a small cache keeps recently used entries of the flat-array for future predictions. *LP* presents several challenges. First, it can have a high false-positive prediction rate. Second, the size of *LP* can grow very large, leading to significant latency and storage overheads. Third, *LP* does not address the large bandwidth consumption of cache prefetchers. In contrast, TLP requires only 4KB per core of storage overhead, and 56 additional bits per prefetch request, while producing accurate off-chip predictions for both load and prefetch requests. With its more streamlined approach, TLP offers a promising solution to the challenges presented by other prefetching methods, paving the way for improved performance and efficiency in modern computing systems.

5.7.2 Data Prefetching

Stream and strided cache prefetchers are unable to effectively prefetch for the indirect memory access patterns of graph-processing workloads [13, 10]. Yu et al. [110] propose a microarchitectural prefetcher that identifies and prefetches indirect memory access patterns without requiring any application nor software information. Ainsworth et al. [4] propose a prefetcher that leverages application-level information to capture indirect memory access patterns. Basak et al. [13] propose DROPLET, a physically decoupled prefetcher that takes into account the reuse distances when applying prefetching for different graph types. Although effective, these hardware prefetchers increase memory bandwidth consumption. In contrast, our proposal reduces the cost of hardware prefetching while keeping its advantages, as Section 5.6 shows.

More ample information about hardware prefetching mechanisms can be found in Section 2.3.

5.7.3 Cache Bypassing

Recent research has proposed several complex cache replacement and bypassing policies [47, 106, 52, 87], that have demonstrated significant performance gains in general-purpose computing applications. However, as we saw in Chapter 3 these policies are ineffective when applied to workloads managing irregular and sparse structures like graphs due to the irregularity of the memory access patterns that these workloads display.

5.7.4 Memory Optimizations for Graph-Processing Applications

Recent work demonstrates the benefits of optimizing the memory hierarchy for graph applications. Ozdal et al. [75] use scratchpads to store vertex and edge data of graph-processing applications, while Gonzalez et al. [40] employ a large eDRAM scratchpad to accommodate larger volumes of graph data than the conventional SRAMs. Several prior works [111, 89, 70, 36, 2] reduce the latency cost of graph memory accesses by executing graph-processing operations close to DRAM, partially hiding the latency cost of the corresponding memory accesses. TLP complements these works since it improves the cache management of a wide range of applications.

5.7.5 Redesigning the Cache Hierarchy

The Distill Cache [81] approach reserves a section of the L2C to place the used words of a cache line when that line is elected for eviction. This design improves the use of cache storage

capacity by evicting just the unused words of each cache line. In contrast, our proposal manages the pervasiveness of highly irregular access patterns and dynamically classifies memory accesses as either regular or irregular. By labeling some memory accesses as not cache-friendly, we avoid cache pollution and useless cache look-ups. The Victim Cache [56] proposal is a small fully-associative cache, found on the refill path of the LLC. It contains eviction victims of the cache to which it is attached and tries to decrease conflict misses. On LLC misses, both the LLC and the Victim Cache are looked-up; if the requested cache block is found in the Victim Cache, the LLC victim and the Victim Cache entry are swapped, thus lowering the miss latency. On a Victim Cache miss, the block is fetched from DRAM and the LLC victim is inserted in the Victim Cache. While the Victim Cache has been proven effective at improving the performance of SPEC workload [91, 92], it relies heavily on spatial locality as it inserts caches victims. Our proposal does not rely on locality assumptions and shortcuts the cache hierarchy when it is predicted to be inefficient.

5.8 Summary

In this research endeavor, we present a groundbreaking innovation: the Two Level Perceptron (TLP) predictor, a novel neural approach meticulously designed to enhance memory access efficiency. TLP harnesses the power of two perceptron predictors, strategically employed to provide predictions for both demand and prefetch requests. This dual approach serves a dual purpose, notably mitigating memory bandwidth wastage due to inaccurate prefetching and off-chip predictions while simultaneously minimizing cache pollution resulting from prefetch operations.

To assess the prowess of TLP in optimizing memory access efficiency, we embarked on a comprehensive evaluation, juxtaposing it against existing approaches, including PPF and Hermes. Our evaluation was wide-ranging, encompassing a diverse array of workloads spanning 55 single-core scenarios, 200 multi-core workloads, and involving two state-of-the-art L1D prefetchers, namely IPCP and Berti.

The results emanating from our study stand as a testament to the substantial performance gains realized through the employment of TLP. In the context of multi-core systems coexisting with the IPCP L1D prefetcher, TLP achieved a remarkable 11.5 % geometric mean speedup. This achievement surpasses the performance of the prior best-performing approach, Hermes, which delivered a respectable 3.0 % improvement. In the context of the Berti prefetcher, where Hermes still exhibited a modest 0.9 % geometric mean speedup, TLP surged ahead, achieving an impressive 11.8 % improvement.

Furthermore, our evaluation revealed a remarkable reduction in DRAM bandwidth transaction overhead across all assessed scenarios when TLP was employed. These findings underscore the significant contribution of TLP in the ongoing quest to optimize memory access efficiency and overall system performance in the dynamic landscape of diverse computing environments.

In conclusion, the introduction of the Two Level Perceptron (TLP) predictor represents a pivotal advancement in the field of memory access optimization. Its dual neural approach not only outperforms existing methods but also significantly reduces DRAM bandwidth transaction overhead. These results highlight the transformative potential of TLP, as it continues to reshape the landscape of memory access efficiency, thereby enhancing performance across a wide spectrum of computing scenarios.

CHAPTER

6

CONCLUSION

La victoire appartient au plus
persévérant.

Napoléon I^{er} Bonaparte

THROUGHOUT Throughout this Ph.D. thesis, a deep and comprehensive exploration into the intricacies of memory access efficiency and cache management strategies across a spectrum of workloads has been meticulously undertaken. This undertaking has led to the developing of three distinct contributions, each of which has contributed insights to this field, shedding light on techniques and approaches geared towards optimizing memory access and cache utilization through AI-inspired techniques and designs.

The first contribution conducted a profound investigation into evaluating Last-Level Cache (LLC) replacement policies within a diverse range of workloads. The findings of this endeavor uncovered the subtle complexities inherent in cache management, emphatically underlining the imperative need to customize cache policies to the unique characteristics of specific workloads. Furthermore, this research introduced pioneering techniques, DS-MPPPB and MS-MPPPB, which offer highly promising solutions tailored to enhance cache management in the challenging domains of big data and High-Performance Computing (HPC) scenarios.

In the second contribution, the spotlight shifted to the underutilization of cache hierarchies when confronted with the intricacies of graph-processing workloads. A novel solution named SDC+LP, designed to overcome these limitations effectively, was presented here. This solution demonstrated remarkable improvements in cache utilization and overall performance when applied to graph-processing workloads. It was a compelling testament to the significance of hardware-based solutions distinguishing between regular and irregular memory accesses, particularly in memory-intensive applications.

The third contribution ushered in the era of neural-inspired approaches by introducing the Two Level Perceptron (TLP) predictor. TLP was meticulously designed to enhance memory access efficiency significantly. Its innovative architecture leveraged neural principles and offered substantial performance improvements, particularly in multi-core systems. TLP's capacity to outperform existing approaches while concurrently reducing DRAM bandwidth transaction overhead across a diverse array of computing scenarios signaled a paradigm shift in memory access optimization.

These contributions serve as beacons illuminating the paramount importance of memory access optimization and cache management in contemporary computing environments. They decisively establish the need for the nuanced tailoring of cache management strategies to suit the peculiarities of specific workloads. Simultaneously, they usher in an era of innovation by introducing pioneering hardware and neural-inspired solutions, thereby enhancing memory access efficiency.

These accomplishments extend beyond the confines of cache management and memory access optimization, offering profound implications for a broader spectrum of memory-intensive applications. As technology continues its relentless march forward, the pursuit of efficient memory access and cache utilization remains an enduring challenge. The contributions of this Ph.D. thesis stand as a testament to the relentless commitment to addressing these challenges and provide an essential foundation upon which to build contemporary computing systems that are both efficient and responsive to the evolving needs of our digital world.

6.1 Future Work

The work presented in this thesis has provided insights into the limitations of state-of-the-art cache replacement policies when applied to High-Performance Computing (HPC) and big data workloads. While these policies have shown significant performance improvements for SPEC 2006 and SPEC 2017 workloads, adapting them to the irregularities of modern workloads, such as graph processing, presents challenges.

6.1.1 Cache Hierarchy and Cache Level Prediction

One avenue for future research lies in the extension of cache hierarchies within modern processors. Current cache designs are well-suited for codes showcasing high levels of predictability and locality, but the deviation of contemporary workloads from these characteristics impacts performance. Researchers are actively exploring innovative solutions to tailor cache hierarchies for the unique demands of graph processing and similar irregular workloads.

Cache Level Prediction (CLP) is another category of approaches that addresses the challenges posed by irregular workloads. Techniques like the Load Predictor (LP) and Hermes leverage CLP to forecast the cache's location within the memory subsystem, aiding in making informed decisions about block fetching from DRAM. Integrating CLP into cache management strategies can help conceal portions of the cache latency associated with irregular access patterns.

6.1.2 Integration with Advanced Prefetching Techniques

The integration of Side Data Cache (SDC) with Cache Level Prediction opens a promising avenue for further exploration. Evaluating the compatibility of state-of-the-art hardware prefetching mechanisms when applied to the SDC and developing novel hardware prefetchers tailored for irregular access patterns can significantly advance memory access optimization.

6.1.3 Scalability, Energy Efficiency, and DRAM Bandwidth Considerations

Assessing the scalability of SDC+LP in heterogeneous computing environments is crucial for broader applicability. Ensuring adaptability to diverse core architectures and memory hierarchies is essential for achieving performance benefits. Investigating the energy-efficiency implications of SDC+LP, especially in power-constrained devices, will be vital for addressing the growing demand for energy-efficient computing solutions. Additionally, exploring DRAM bandwidth implications, particularly in multi-core scenarios, can provide insights into optimizing SDC+LP for improved performance and energy efficiency.

6.1.4 Enhancing Neural Predictors and Integration with Emerging Memory Technologies

Future research efforts can focus on enhancing neural predictors, such as the Two Level Perceptron (TLP). Strategies to boost prediction accuracy, adaptability, and scalability to support increasingly complex memory hierarchies are avenues worth exploring. Integration with emerging memory technologies, including Non-Volatile Memory (NVM) and High-Bandwidth Memory (HBM), presents a compelling opportunity for future research. Investigating innovative memory hierarchy designs that combine the potential of neural prediction and emerging memory technologies can shape the future landscape of memory optimization.

6.1.5 Integration within GPUs

The integration of TLP into Graphics Processing Units (GPUs) holds substantial promise for the future of parallel computing. TLP's demonstrated ability to enhance memory access efficiency, reduce cache pollution, and optimize DRAM bandwidth usage positions it as a valuable candidate for future GPU designs. Seamless integration of TLP into GPU memory hierarchies has the potential to deliver significant performance gains across a range of applications, from high-end gaming to data-intensive scientific simulations and artificial intelligence tasks.

6.1.6 Final Words

Above, we outlined promising avenues for future research corresponding to each contribution in this dissertation. In addition to these, an exciting research prospect involves amalgamating the insights from the last two contributions (*cf.*: Chapters 4 and 5). This research direction would explore the potential of utilizing Hermes as an alternative to the predictor presented in Chapter 4.

In conclusion, these focused research areas serve as beacons guiding the way forward, building upon the foundational contributions of TLP and driving innovation in memory access optimization to address the evolving demands of computing systems and workloads.

Acknowledgments and Funding Supports

This work has been made possible through diverse and essential sources of support. We acknowledge the contributions of the European HiPEAC Network of Excellence, the Spanish

Ministry of Science and Innovation (contracts PID2019-107255GB-C21 and PID2019-105660RB-C22), and the Generalitat de Catalunya (contract 2021-SGR-00763).

The National Science Foundation and Intel have played pivotal roles, providing financial support through grant CCF-1912617 and generous gifts, respectively.

Marc Casas expresses gratitude for the support from the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and ESF Investing in your future.

The research group “Performance understanding, analysis, and simulation/emulation of novel architectures,” supported by the Departament de Recerca i Universitats de la Generalitat de Catalunya (Codi: 2021 SGR 00865), has been instrumental in facilitating this work.

European High-Performance Computing Joint Undertaking, under Framework Partnership Agreement No 800928 (European Processor Initiative), and Specific Grant Agreement No 101036168 (EPI SGA2), has provided significant funding. This collaboration involves the European Union’s Horizon 2020 research and innovation program, along with support from countries including Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project, PCI2022-132935, is co-funded by MCIN/AEI/10.13039/501100011033 and by the UE NextGenerationEU/PRTR.

Lluc Alvarez’s work has been supported by the Spanish Ministry of Economy, Industry, and Competitiveness under the Juan de la Cierva Formación fellowship number FJCI-2016-30984.

Additionally, the European Union’s Horizon 2020 research and innovation program has provided funding under the Mont-Blanc 2020 project (grant agreement 779877).

Finally, the ongoing contributions of Marc Casas, supported by the Spanish Ministry of Economy, Industry, and Competitiveness under Ramon y Cajal fellowship number RYC-2017-23269, and the support from National Science Foundation grant CCF-1912617, Semiconductor Research Corporation project 2936.001, and generous gifts from Intel Labs have been crucial in the continued progress of this work.

CHAPTER

7

PUBLICATIONS

THE research findings presented in this dissertation have resulted in the following conference publications, in which the author assumes the role of the main researcher.

- Alexandre Valentin Jamet, Lluc Alvarez, Daniel A. Jiménez, and Marc Casas, **Characterizing the impact of last-level cache replacement policies on big-data workloads**, in 2020 IEEE International Symposium on Workload Characterization (IISWC), Oct. 2020, pp. 134–144. doi: 10.1109/IISWC50251.2020.00022.
- Alexandre Valentin Jamet, Lluc Alvarez, Daniel A. Jiménez, and Marc Casas, **Practically Tackling Memory Bottlenecks of Graph-Processing Workloads**, accepted for publication at the 38th IEEE International Parallel & Distributed Processing Symposium (IPDPS’24).
- Alexandre Valentin Jamet, Georgios Vavouliotis, Lluc Alvarez, Daniel A. Jiménez, and Marc Casas, **A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering**, accepted for publication at the 30th IEEE International Symposium on High-Performance Computer Architecture (HPCA’30).

- Alexandre Valentin Jamet, Lluc Alvarez, and Marc Casas, **Characterizing the impact of last-level cache replacement policies on big-data workloads.** arXiv, May 11, 2023. Accessed: Sep. 24, 2023. [Online]. Available: <http://arxiv.org/abs/2305.06696>

REFERENCES

- [1] J. Abella, A. González, X. Vera, and M. F. P. O’Boyle, “Iatac: A smart predictor to turn-off l2 cache lines,” *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, 55–77, Mar. 2005, ISSN: 1544-3566. DOI: 10.1145/1061267.1061271. [Online]. Available: <https://doi.org/10.1145/1061267.1061271> (cit. on p. 59).
- [2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117. DOI: 10.1145/2749469.2750386 (cit. on pp. 94, 100, 128).
- [3] S. Ainsworth and T. M. Jones, “An event-triggered programmable prefetcher for irregular workloads,” *ACM Sigplan Notices*, vol. 53, no. 2, pp. 578–592, 2018 (cit. on p. 38).
- [4] S. Ainsworth and T. M. Jones, “Graph Prefetching Using Data Structure Knowledge,” in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS ’16, Istanbul, Turkey: Association for Computing Machinery, 2016, ISBN: 9781450343619. DOI: 10.1145/2925426.2926254. [Online]. Available: <https://doi.org/10.1145/2925426.2926254> (cit. on pp. 70, 94, 100, 128).
- [5] S. Ainsworth and T. M. Jones, “Graph prefetching using data structure knowledge,” in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–11 (cit. on p. 102).
- [6] An-Chow Lai, C. Fide, and B. Falsafi, “Dead-block prediction dead-block correlating prefetchers,” pp. 144–154, 2001, ISSN: 1063-6897. DOI: 10.1109/ISCA.2001.937443 (cit. on p. 57).
- [7] (). Apple unveils m3, m3 pro, and m3 max, the most advanced chips for a personal computer, Apple Newsroom (cit. on p. 2).

- [8] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, “Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31. DOI: 10.1109/IPDPS.2016.110 (cit. on pp. 70, 95).
- [9] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 399–411 (cit. on p. 35).
- [10] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, “P-OPT: Practical Optimal Cache Replacement for Graph Analytics,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 668–681 (cit. on pp. 12, 70, 71, 84, 94, 128).
- [11] V. Balaji and B. Lucia, “Combining data duplication and graph reordering to accelerate parallel graph processing,” in *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*, 2019, pp. 133–144 (cit. on pp. 100, 102).
- [12] —, “When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2018, pp. 203–214 (cit. on pp. 100, 102).
- [13] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, “Analysis and optimization of the memory hierarchy for graph processing workloads,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 373–386. DOI: 10.1109/HPCA.2019.00051 (cit. on pp. 31, 39, 70, 94, 100, 102, 128).
- [14] S. Beamer, K. Asanović, and D. Patterson, “Reducing pagerank communication via propagation blocking,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2017, pp. 820–831 (cit. on pp. 100, 102).
- [15] —, “The gap benchmark suite,” *arXiv preprint arXiv:1508.03619*, 2015 (cit. on pp. 9, 10, 12, 73, 74, 82, 83, 116–118, 152, 153).
- [16] S. Beamer, K. Asanović, and D. Patterson, “The GAP benchmark suite,” *arXiv:1508.03619 [cs]*, May 16, 2017. arXiv: 1508.03619. [Online]. Available: <http://arxiv.org/abs/1508.03619> (cit. on pp. 35–37, 39, 40, 46, 48).
- [17] N. Beckmann and D. Sanchez, “Maximizing cache performance under uncertainty,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX: IEEE, Feb. 2017, pp. 109–120, ISBN: 978-1-5090-4985-1. DOI: 10.1109/HPCA.2017.43. [Online]. Available: <http://ieeexplore.ieee.org/document/7920818/> (cit. on p. 59).
- [18] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems Journal*, pp. 78–101, Jun. 1966, ISSN: 0018-8670. [Online]. Available: <https://doi.org/10.1147/sj.52.0078> (cit. on pp. 19, 26, 53).
- [19] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, “Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1–18. DOI: 10.1109/MICRO56248.2022.00015 (cit. on pp. 3, 12, 40, 42, 100–103, 111, 115, 118, 119, 126).

- [20] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, “Perceptron-based prefetch filtering,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 1–13 (cit. on pp. 100–103, 119, 126).
- [21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, 72–81, ISBN: 9781605582825. DOI: 10.1145/1454115.1454128. [Online]. Available: <https://doi.org/10.1145/1454115.1454128> (cit. on pp. 100, 102).
- [22] M. Bohr, “A 30 year retrospective on dennard’s mosfet scaling paper,” *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007 (cit. on p. 2).
- [23] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001 (cit. on pp. 82, 116).
- [24] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998 (cit. on pp. 70, 72, 73, 100, 102).
- [25] T. D. Bui, S. Ravi, and V. Ramavajjala, “Neural graph learning: Training neural networks using graphs,” in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM ’18, Marina Del Rey, CA, USA: Association for Computing Machinery, 2018, 64–71, ISBN: 9781450355810. DOI: 10.1145/3159652.3159731. [Online]. Available: <https://doi.org/10.1145/3159652.3159731> (cit. on pp. 70, 100, 102).
- [26] (). Cascade lake - microarchitectures - intel - WikiChip, [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake#Memory_Hierarchy (cit. on pp. 18, 81, 116).
- [27] (). Championship value prediction (CVP), [Online]. Available: <https://www.microarch.org/cvp1/> (cit. on p. 10).
- [28] (). Championship value prediction (CVP), [Online]. Available: <https://www.microarch.org/cvp1/> (cit. on p. 46).
- [29] *ChampSim*, <https://crc2.ece.tamu.edu/>, [Online], 2021 (cit. on pp. 81, 152).
- [30] ChampSim, *ChampSim/ChampSim*, original-date: 2017-06-30T05:41:50Z, Jun. 15, 2020. [Online]. Available: <https://github.com/ChampSim/ChampSim> (cit. on p. 59).
- [31] E. Cuthill and J. McKee, “Reducing the Bandwidth of Sparse Symmetric Matrices,” in *Proceedings of the 1969 24th National Conference*, ser. ACM ’69, New York, NY, USA: Association for Computing Machinery, 1969, 157–172, ISBN: 9781450374934. DOI: 10.1145/800195.805928. [Online]. Available: <https://doi.org/10.1145/800195.805928> (cit. on pp. 70, 95).
- [32] G. Dai, Y. Chi, Y. Wang, and H. Yang, “Fpgp: Graph processing framework on fpga a case study of breadth-first search,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16, Monterey, California, USA: Association for Computing Machinery, 2016, 105–110, ISBN: 9781450338561. DOI: 10.1145/2847263.2847339. [Online]. Available: <https://doi.org/10.1145/2847263.2847339> (cit. on pp. 70, 95).

- [33] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, “Improving cache management policies using dynamic reuse distances,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 389–400. DOI: 10.1109/MICRO.2012.43 (cit. on p. 59).
- [34] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, “Simultaneous multithreading: A platform for next-generation processors,” *IEEE micro*, vol. 17, no. 5, pp. 12–19, 1997 (cit. on p. 2).
- [35] O. Evelien and R. Rousseau, “Social network analysis: A powerful strategy, also for the information sciences,” *Journal of Information Science*, vol. 28, no. 6, 441–453, 2002 (cit. on pp. 70, 72, 100, 102).
- [36] M. Gao, G. Ayers, and C. Kozyrakis, “Practical near-data processing for in-memory analytics frameworks,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 113–124. DOI: 10.1109/PACT.2015.22 (cit. on pp. 94, 100, 128).
- [37] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, “Bit-level perceptron prediction for indirect branches,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 27–38 (cit. on p. 102).
- [38] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, *The championship simulator: Architectural simulation for education and competition*, 2022. arXiv: 2210.14324 [cs.AR] (cit. on p. 116).
- [39] A. González, C. Aliagas, and M. Valero, “A data cache with multiple caching strategies tuned to different types of locality,” in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 1995, pp. 217–226 (cit. on p. 95).
- [40] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “GraphX: Graph Processing in a Distributed Dataflow Framework,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14, Broomfield, CO: USENIX Association, 2014, 599?613, ISBN: 9781931971164 (cit. on pp. 94, 100, 128).
- [41] P. V. Gratz, J. Kim, and G. Chacon. (Jun. 2017). THE 2nd CACHE REPLACEMENT CHAMPIONSHIP – co-located with ISCA june 2017, <https://crc2.ece.tamu.edu/>, [Online]. Available: <https://crc2.ece.tamu.edu/> (cit. on p. 60).
- [42] J. L. Greathouse and M. Daga, “Efficient sparse matrix-vector multiplication on gpus using the csr storage format,” in *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 769–780. DOI: 10.1109/SC.2014.68 (cit. on p. 72).
- [43] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783759 (cit. on pp. 70, 95).
- [44] C. T. Have and L. J. Jensen, “Are graph databases ready for bioinformatics?” *Bioinformatics*, vol. 29, no. 24, pp. 3107–3108, Oct. 2013, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt549. eprint: <https://academic.oup.com/bioinformatics/article-pdf/29/24/3107/25024470/btt549.pdf>. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btt549> (cit. on pp. 70, 72, 100, 102).

-
- [45] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea: IEEE, Jun. 18, 2016, pp. 78–89, ISBN: 978-1-4673-8947-1. DOI: <https://doi.org/10.1109/ISCA.2016.17>. [Online]. Available: <https://dl.acm.org/citation.cfm?doid=3007787.3001146> (cit. on pp. 3, 10, 26, 46, 53).
 - [46] A. Jaleel, “Memory characterization of workloads using instrumentation-driven simulation,” *Web Copy*: <http://www.glue.umd.edu/ajaleel/workload>, 2010 (cit. on pp. 46, 48).
 - [47] A. Jaleel, K. B. Theobald, S. C. S. Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (RRIP),” in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA ’10, vol. ISCA’10, Saint-Malo, France: IEEE, Jun. 19, 2010, pp. 60–71, ISBN: 978-1-4503-0053-7. DOI: <https://doi.org/10.1145/1815961.1815971>. [Online]. Available: <https://dl.acm.org/citation.cfm?doid=1815961.1815971> (cit. on pp. 10, 21, 22, 46, 51, 52, 59, 94, 100, 128).
 - [48] M. Jalili and M. Erez, “Reducing load latency with cache level prediction,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 648–661. DOI: 10.1109/HPCA53966.2022.00054 (cit. on pp. 12, 40, 100, 102, 127).
 - [49] D. Jimenez, “Piecewise linear branch prediction,” in *32nd International Symposium on Computer Architecture (ISCA’05)*, 2005, pp. 382–393. DOI: 10.1109/ISCA.2005.40 (cit. on p. 102).
 - [50] D. Jiménez and C. Lin, “Dynamic branch prediction with perceptrons,” in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA ’01, 2001, pp. 197–206. DOI: 10.1109/HPCA.2001.903263 (cit. on pp. 2, 4, 102).
 - [51] D. A. Jiménez, “Multiperspective perceptron predictor,” in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016 (cit. on pp. 3, 10, 25, 117).
 - [52] D. A. Jiménez and E. Teran, “Multiperspective reuse prediction,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2017, pp. 436–448 (cit. on pp. 94, 128).
 - [53] D. A. Jiménez and E. Teran, “Multiperspective reuse prediction,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17, Cambridge, Massachusetts: Association for Computing Machinery, 2017, 436–448, ISBN: 9781450349529. DOI: 10.1145/3123939.3123942. [Online]. Available: <https://doi.org/10.1145/3123939.3123942> (cit. on p. 102).
 - [54] D. A. Jiménez, “Insertion and promotion for tree-based PseudoLRU last-level caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, Davis, California: ACM Press, 2013, pp. 284–296, ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540733. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2540708.2540733> (cit. on pp. 57, 59).

- [55] D. A. Jiménez and E. Teran, “Multiperspective reuse prediction,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*, ser. MICRO '17, IEEE, Cambridge, Massachusetts: IEEE, 2017, pp. 436–448, ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123942. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3123939.3123942> (cit. on pp. 23, 46, 52, 56, 57, 83, 100, 117, 118).
- [56] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990 (cit. on pp. 95, 129).
- [57] S. Khan and D. A. Jimenez, “Insertion policy selection using decision tree analysis,” in *2010 IEEE International Conference on Computer Design*, Amsterdam, Netherlands: IEEE, Oct. 2010, pp. 106–111, ISBN: 978-1-4244-8936-7. DOI: 10.1109/ICCD.2010.5647608. [Online]. Available: <http://ieeexplore.ieee.org/document/5647608/> (cit. on p. 56).
- [58] S. M. Khan, Y. Tian, and D. A. Jimenez, “Sampling dead block prediction for last-level caches,” in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA: IEEE, Dec. 2010, pp. 175–186, ISBN: 978-1-4244-9071-4. DOI: 10.1109/MICRO.2010.24. [Online]. Available: <http://ieeexplore.ieee.org/document/5695535/> (cit. on pp. 22, 23, 25, 52, 57, 59).
- [59] S. M. Khan, Y. Tian, and D. A. Jiménez, “Sampling dead block prediction for last-level caches,” in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 175–186. DOI: 10.1109/MICRO.2010.24 (cit. on pp. 22, 100, 117).
- [60] M. Kharbutli and Y. Solihin, “Counter-based cache replacement and bypassing algorithms,” *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008, ISSN: 1557-9956. DOI: 10.1109/TC.2007.70816 (cit. on p. 59).
- [61] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, “Path confidence based lookahead prefetching,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12 (cit. on pp. 33, 35, 82, 103, 116, 117).
- [62] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, “A unified sparse matrix data format for modern processors with wide SIMD units,” *CoRR*, vol. abs/1307.6209, 2013. arXiv: 1307.6209. [Online]. Available: <http://arxiv.org/abs/1307.6209> (cit. on p. 72).
- [63] H. Liu, M. Ferdman, J. Huh, and D. Burger, “Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency,” in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 222–233. DOI: 10.1109/MICRO.2008.4771793 (cit. on p. 57).
- [64] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943 (cit. on p. 5).
- [65] G. Memik, G. Reinman, and W. Mangione-Smith, “Just say no: Benefits of early cache miss determination,” in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 307–316. DOI: 10.1109/HPCA.2003.1183548 (cit. on pp. 100, 102).

- [66] U. Meyer and P. Sanders, “ Δ -stepping: A parallelizable shortest path algorithm,” *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003 (cit. on pp. 82, 116).
- [67] G. E. Moore, “Cramming more components onto integrated circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998 (cit. on p. 2).
- [68] G. E. Moore *et al.*, “Progress in digital integrated electronics,” in *Electron devices meeting*, Washington, DC, vol. 21, 1975, pp. 11–13 (cit. on p. 2).
- [69] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, “Exploiting locality in graph analytics through hardware-accelerated traversal scheduling,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 1–14 (cit. on pp. 70, 95, 96).
- [70] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468. DOI: 10.1109/HPCA.2017.54 (cit. on pp. 94, 100, 128).
- [71] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, “Berti: An accurate local-delta data prefetcher,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 975–991 (cit. on pp. 31, 34, 35, 101, 107, 116, 117).
- [72] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, “Prefedge: Ssd prefetcher for large-scale graph traversal,” in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR 2014, Haifa, Israel: Association for Computing Machinery, 2014, 1?12, ISBN: 9781450329200. DOI: 10.1145/2611354.2611365. [Online]. Available: <https://doi.org/10.1145/2611354.2611365> (cit. on p. 70).
- [73] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, “Graphgen: An fpga framework for vertex-centric graph computation,” in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 25–28. DOI: 10.1109/FCCM.2014.15 (cit. on pp. 70, 95).
- [74] T. Oguntebi and K. Olukotun, “Graphops: A dataflow library for graph analytics acceleration,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16, Monterey, California, USA: Association for Computing Machinery, 2016, 111?117, ISBN: 9781450338561. DOI: 10.1145/2847263.2847337. [Online]. Available: <https://doi.org/10.1145/2847263.2847337> (cit. on pp. 70, 95).
- [75] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, “Energy Efficient Architecture for Graph Analytics Accelerators,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea: IEEE Press, 2016, 166–177, ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.24. [Online]. Available: <https://doi.org/10.1109/ISCA.2016.24> (cit. on pp. 70, 94, 95, 100, 128).
- [76] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131. DOI: 10.1109/ISCA45697.2020.00021 (cit. on pp. 31, 33, 35, 101, 107, 116, 117).

- [77] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multi-processors with private cache memories,” in *11th annual International Symposium on Computer Architecture (ISCA)*, New York, NY, USA: Association for Computing Machinery, 1984, 348–354, ISBN: 0818605383. DOI: 10.1145/800015.808204. [Online]. Available: <https://doi.org/10.1145/800015.808204> (cit. on p. 79).
- [78] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016 (cit. on p. 18).
- [79] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, “Using SimPoint for accurate and efficient simulation,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, p. 2, 2003 (cit. on pp. 59, 83, 117).
- [80] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07, New York, NY, USA: IEEE, 2007, pp. 381–391, ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250709. [Online]. Available: <http://doi.acm.org/10.1145/1250662.1250709> (cit. on pp. 10, 21, 22, 51, 52).
- [81] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, “Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines,” in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 250–259. DOI: 10.1109/HPCA.2007.346202 (cit. on pp. 70, 84, 95, 100, 128).
- [82] R. Rojas and U. Hashagen, *The first computers: History and architectures*. MIT press, 2002 (cit. on p. 18).
- [83] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.,” *Psychological review*, vol. 65, no. 6, p. 386, 1958 (cit. on pp. 2, 6).
- [84] A. Sembrant, E. Hagersten, and D. Black-Schaffer, “The direct-to-data (d2d) cache: Navigating the cache hierarchy with a single lookup,” *SIGARCH Comput. Archit. News*, vol. 42, no. 3, 133–144, 2014, ISSN: 0163-5964. DOI: 10.1145/2678373.2665694. [Online]. Available: <https://doi.org/10.1145/2678373.2665694> (cit. on pp. 100, 102).
- [85] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, “The evicted-address filter: A unified mechanism to address both cache pollution and thrashing,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, 355–366, ISBN: 9781450311823. DOI: 10.1145/2370816.2370868. [Online]. Available: <https://doi.org/10.1145/2370816.2370868> (cit. on pp. 100, 103).
- [86] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks,” vol. 11, no. 4, 2015, ISSN: 1544-3566. DOI: 10.1145/2677956. [Online]. Available: <https://doi.org/10.1145/2677956> (cit. on pp. 100, 103).

- [87] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying deep learning to the cache replacement problem,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, 413–425, ISBN: 9781450369381. DOI: 10.1145/3352460.3358319. [Online]. Available: <https://doi.org/10.1145/3352460.3358319> (cit. on pp. 3, 10, 30, 46, 53, 58, 62, 83, 94, 100, 117, 118, 128).
- [88] Y. Shiloach and U. Vishkin, “An $O(\log n)$ parallel connectivity algorithm,” Computer Science Department, Technion, Tech. Rep., 1980 (cit. on pp. 82, 116).
- [89] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “GraphR: Accelerating Graph Processing Using ReRAM,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543. DOI: 10.1109/HPCA.2018.00052 (cit. on pp. 94, 100, 128).
- [90] G. Southern and J. Renau, “Analysis of parsec workload scalability,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2016, pp. 133–142 (cit. on pp. 40, 41).
- [91] (). SPEC CPU® 2006, [Online]. Available: <https://www.spec.org/cpu2006/> (cit. on pp. 10, 12, 22, 30, 40, 41, 90, 93, 95, 116, 118, 129, 152, 153).
- [92] (). SPEC CPU® 2017, [Online]. Available: <https://www.spec.org/cpu2017/> (cit. on pp. 10, 12, 22, 30, 35, 40, 41, 90, 93, 95, 116, 118, 129, 152, 153).
- [93] P. Sweazey and A. J. Smith, “A class of compatible cache consistency protocols and their support by the ieee futurebus,” in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA ’86, Tokyo, Japan: IEEE Computer Society Press, 1986, 414–423, ISBN: 081860719X (cit. on p. 79).
- [94] D. Tarjan, K. Skadron, and M. Stan, “An ahead pipelined alloyed perceptron with single cycle access time,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, p. 8. DOI: 10.1.1.65.8454 (cit. on p. 57).
- [95] E. Teran, Y. Tian, Z. Wang, and D. A. Jimenez, “Minimal disturbance placement and promotion,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain: IEEE, Mar. 2016, pp. 201–211, ISBN: 978-1-4673-9211-2. DOI: 10.1109/HPCA.2016.7446065. [Online]. Available: <http://ieeexplore.ieee.org/document/7446065/> (cit. on pp. 57, 59).
- [96] E. Teran, Z. Wang, and D. A. Jimenez, “Perceptron learning for reuse prediction,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, ser. MICRO ’16, Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–12, ISBN: 978-1-5090-3508-3. DOI: 10.1109/MICRO.2016.7783705. [Online]. Available: <http://ieeexplore.ieee.org/document/7783705/> (cit. on pp. 3, 23, 52, 59).
- [97] E. Teran, Z. Wang, and D. A. Jiménez, “Perceptron learning for reuse prediction,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783705 (cit. on pp. 100, 102).
- [98] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, “XS Bench – the development and verification of a performance abstraction for monte carlo reactor analysis,” in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014, p. 12. [Online]. Available: <https://www.mcs.anl.gov/papers/P5064-0114.pdf> (cit. on pp. 10, 46, 50).

- [99] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403 (cit. on p. 2).
- [100] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, “Exploiting page table locality for agile tlb prefetching,” in *Proceedings of the 48th International Symposium on Computer Architecture*, 2021. DOI: 10.1109/ISCA52012.2021.00016 (cit. on p. 117).
- [101] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, “Page size aware cache prefetching,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 956–974. DOI: 10.1109/MICRO56248.2022.00070 (cit. on p. 118).
- [102] H. Wei, J. X. Yu, C. Lu, and X. Lin, “Speedup graph processing by graph ordering,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16, San Francisco, California, USA: Association for Computing Machinery, 2016, 1813–1828, ISBN: 9781450335317. DOI: 10.1145/2882903.2915220. [Online]. Available: <https://doi.org/10.1145/2882903.2915220> (cit. on pp. 70, 95, 100).
- [103] —, “Speedup graph processing by graph ordering,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1813–1828 (cit. on p. 102).
- [104] B. Widrow and M. Lehr, “30 years of adaptive neural networks: Perceptron, madaline, and backpropagation,” *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1442, Sep. 1990, ISSN: 00189219. DOI: 10.1109/5.58323. [Online]. Available: <http://ieeexplore.ieee.org/document/58323/> (cit. on p. 57).
- [105] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, “SHiP: Signature-based hit predictor for high performance caching,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 ’11*, Porto Alegre, Brazil: ACM Press, 2011, p. 430, ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155671. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2155620.2155671> (cit. on pp. 22, 23, 46, 52).
- [106] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, “Ship: Signature-based hit predictor for high performance caching,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 430–441 (cit. on pp. 10, 94, 100, 128).
- [107] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1, 1995. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=216585.216588> (cit. on pp. 16, 46).
- [108] J. Yang and J. Leskovec, “Community-affiliation graph model for overlapping network community detection,” in *2012 IEEE 12th international conference on data mining*, IEEE, 2012, pp. 1170–1175 (cit. on pp. 74, 83, 118).
- [109] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, “Speculation techniques for improving load related instruction scheduling,” *SIGARCH Comput. Archit. News*, vol. 27, no. 2, 42–53, 1999, ISSN: 0163-5964. DOI: 10.1145/307338.300983. [Online]. Available: <https://doi.org/10.1145/307338.300983> (cit. on pp. 100, 102).

-
- [110] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “Imp: Indirect memory prefetcher,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, Waikiki, Hawaii: Association for Computing Machinery, 2015, 178–190, ISBN: 9781450340342. DOI: 10.1145/2830772.2830807. [Online]. Available: <https://doi.org/10.1145/2830772.2830807> (cit. on pp. 31, 35, 36, 54, 70, 94, 128).
 - [111] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 544–557. DOI: 10.1109/HPCA.2018.00053 (cit. on pp. 94, 100, 128).
 - [112] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, “Making caches work for graph analytics,” in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302. DOI: 10.1109/BigData.2017.8257937 (cit. on p. 54).
 - [113] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, “Making caches work for graph analytics,” in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302. DOI: 10.1109/BigData.2017.8257937 (cit. on pp. 3, 70).
 - [114] —, “Making caches work for graph analytics,” in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 293–302 (cit. on pp. 100, 102).
 - [115] Zhigang Hu, S. Kaxiras, and M. Martonosi, “Timekeeping in the memory system: Predicting and optimizing memory behavior,” in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 209–220. DOI: 10.1109/ISCA.2002.1003579 (cit. on p. 59).
 - [116] X. Zhuang and H.-H. Lee, “A hardware-based cache pollution filtering mechanism for aggressive prefetches,” in *2003 International Conference on Parallel Processing, 2003. Proceedings.*, 2003, pp. 286–293. DOI: 10.1109/ICPP.2003.1240591 (cit. on pp. 100, 103).

APPENDIX

A

ARTIFACT EVALUATION – A TWO LEVEL NEURAL APPROACH COMBINING OFF-CHIP PREDICTION WITH ADAPTIVE PREFETCH FILTERING

A.1 Abstract

This appendix serves as the Artifact Evaluation (AE) provided to reviewers of the HPCA 2024 conference for the contribution presented in Chapter 5. Our artifact provides i) the implementation of TLP, ii) the simulation infrastructure, iii) the set of workloads, iv) scripts for launching the experiments, and v) Python scripts bundled in Jupyter notebooks to exploit the simulation results and reproduce some of the key figures of this paper.

A.2 Artifact check-list (meta-information)

- **Program:** Memory traces of SPEC 2006 [91], SPEC 2017 [92], and GAP [15] workloads.
- **Compilation:** GNU GCC and CMake.
- **Metrics:** Performance improvements, reduction in DRAM transactions, statistics on inaccurate off-chip predictions, L1D useful & useless prefetches, and L1D prefetchers' accuracy.
- **Output:** We provide scripts that generate all single-core figures (Figures 5.1, 5.2, 5.4, 5.5, 5.6, 5.10, 5.11, 5.12).
- **Experiments:** We provide scripts that submit the required jobs. The only requirement is a SLURM manager.
- **How much disk space required (approximately)?:** 140GB.
- **How much time is needed to prepare workflow (approximately)?:** About 1 hour.
- **How much time is needed to complete experiments (approximately)?:** About 12 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Workflow framework used?:** SLURM for job management.
- **Archived (provide DOI)?:** The code is available at <https://doi.org/10.5281/zenodo.10100304>. The trace set is available in 3 volumes:
 - Volume 1: <https://doi.org/10.5281/zenodo.10083542>
 - Volume 2: <https://doi.org/10.5281/zenodo.10088347>
 - Volume 3: <https://doi.org/10.5281/zenodo.10088525>

A.3 Description

A.3.1 How to access

Our artifact is available at <https://doi.org/10.5281/zenodo.10100304>.

A.3.2 Hardware dependencies

Any hardware capable of compiling and running ChampSim [29].

A.3.3 Software dependencies

Our artifact depends on the following tools: CMake, Jupyter, Python 3.8.10, matplotlib, and SLURM.

A.3.4 Data sets

Memory traces of SPEC 2006 [91], SPEC 2017 [92], and GAP [15] workloads.

A.4 Installation

First, Download the artifact from our GitHub repository using the appropriate `git clone` command. Second, download the trace set from the following Zenodo records:

- Volume 1: <https://doi.org/10.5281/zenodo.10083542>
- Volume 2: <https://doi.org/10.5281/zenodo.10088347>
- Volume 3: <https://doi.org/10.5281/zenodo.10088525>

A.5 Experiment workflow

To reproduce all the single-core figures of this work, take the following steps:

- `cd TLP-HPCA30-artifact`
- Move the three volumes of the traces artifact to the root of the code's artifact.
- Extract the traces using `tar -xmf TLP-HPCA30-artifact-traces.VOLUME1.tar`. This command is interactive and will request you to provide the name of the next archive's volume as follows `TLP-HPCA30-artifact-traces.VOLUME2.tar`, etc. A new directory named `traces/` will be present in the artifact directory.
- set paths and username in `scripts/run_single_core.sh` (lines 4, 7, 8, 10, and 11), `scripts/run_single_core_legacy.sh` (lines 4, 7, 8, 10, and 11), `scripts/run_single_core.j` (lines 6, 9, 10, and 12).
- Execute `./scripts/compile_single_core.sh` to compile the binaries.
- Execute `scripts/run_single_core.sh` and `scripts/run_single_core_legacy.sh` to launch all single-core simulations.

Running all the jobs takes around 12 hours, depending on the cluster and the number of jobs that can be launched in parallel.

A.6 Evaluation and expected results

When all jobs are finished, generate the single-core figures using the Jupyter notebooks provided in the notebooks directory. We recommend using the Jupyter extension in the VS Code editor, as it is how the workflow was originally designed.

The single-core figures will be available in the `plots/` directory.

A.7 Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-badging>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>