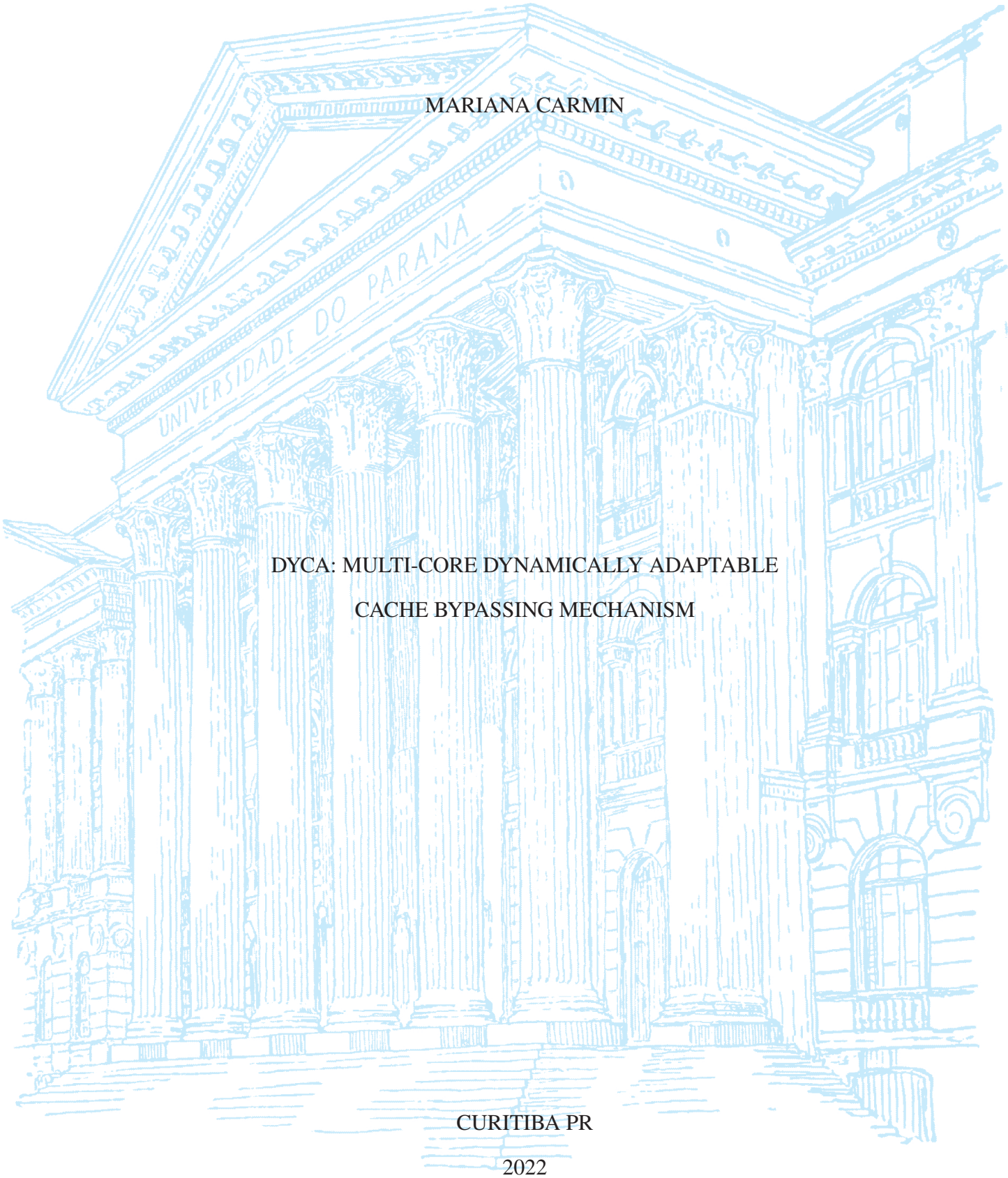
UNIVERSIDADE FEDERAL DO PARANÁ

MARIANA CARMIN



DYCA: MULTI-CORE DYNAMICALLY ADAPTABLE CACHE BYPASSING MECHANISM

CURITIBA PR

2022

MARIANA CARMIN

DYCA: MULTI-CORE DYNAMICALLY ADAPTABLE CACHE BYPASSING MECHANISM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Marco Antonio Zanata Alves.

CURITIBA PR

2022

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Carmin, Mariana
DYCA : multi-core dynamically adaptable cache bypassing mechanism / Mariana Carmin. – Curitiba, 2022.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Marco Antonio Zanata Alves

1. Memória cache. 2. Regressão linear. 3. Computação de alto desempenho. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Alves, Marco Antonio Zanata. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da dissertação de Mestrado de **MARIANA CARMIN** intitulada: **DYCA: Multi-core Dynamically Adaptable Cache Bypassing Mechanism**, sob orientação do Prof. Dr. MARCO ANTONIO ZANATA ALVES, que após terem inquirido a aluna e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestra está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Outubro de 2022.

Assinatura Eletrônica
18/10/2022 08:53:15.0
MARCO ANTONIO ZANATA ALVES
Presidente da Banca Examinadora

Assinatura Eletrônica
18/10/2022 14:23:54.0
ARTHUR FRANCISCO LORENZON
Avaliador Externo (UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL)

Assinatura Eletrônica
20/10/2022 08:29:59.0
DANIEL ALFONSO GONCALVES DE OLIVEIRA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

"What do you mean, why's it got to be built? It's a bypass. You got to build bypasses." - Douglas Adams, Hitchhiker's Guide to the Galaxy, 1979.

ACKNOWLEDGEMENTS

I will always be grateful for all the blessings received from heaven and for the people I met on this journey. First and foremost I am extremely thankful to my supervisor, Prof. Marco Zanata who guided me in the development of this research. I would also like to thank my loving husband Lucas who has constantly encouraged me throughout this process. Finally, I would like to express my gratitude to my family and friends for their encouragement and support all through my studies.

RESUMO

A maioria dos processadores modernos possuem uma hierarquia de cache formada de múltiplos níveis para mitigar a latência de acesso a memória principal. Além disso, estes processadores também possuem um número cada vez maior de núcleos, a medida que o número de núcleos aumenta, mais núcleos e threads compartilham o último nível de cache (LLC - Last Level Cache), que consome grande parte da energia e área do chip. Portanto, enquanto o tamanho e o número de núcleos aumentam, novas soluções devem garantir o melhor uso de recursos reduzindo os conflitos de cache e problemas de poluição de cache. Esta dissertação explora o conhecimento de que algumas aplicações apresentam baixa localidade temporal e espacial durante os acessos aos dados. Para esses casos, o LLC impõe uma barreira de latência extra para acessar os dados na memória principal além de um desperdício de energia durante a instalação das linhas de cache. Assim, o desvio (bypass) de um ou mais níveis de cache pode beneficiar tais aplicativos, melhorando o desempenho geral do sistema e diminuindo o consumo de energia. Nesta dissertação, propomos um preditor online para contornar a adaptação dinâmica do uso da LLC. Nosso mecanismo avalia cada fase de execução das aplicações coletando contadores de hardware habilitados por mecanismos de amostragem e usando um modelo de regressão linear capaz de identificar se o LLC beneficia o desempenho do aplicativo, adaptando o uso da LLC para cada aplicação sendo executada. Desta forma, diminuindo o tempo de execução em uma execução singular, além de aumentando o desempenho geral do sistema. Esse desvio de cache também cria desafios quanto ao protocolo de coerência em execuções de múltiplos aplicativos. Contudo, nosso mecanismo é capaz de garantido esta coerência. Nossos resultados para execuções singulares mostram um possível aumento no tempo de execução de até 22%. Já para execuções de múltiplos programas, mostram que ganhos de até 21% no tempo de execução podem ser alcançados e também a redução da taxa de faltas na LLC, oriundos da diminuição da poluição e conflitos na cache.

Palavras-chave: Cache de último nível; Regressão linear; Predição de desempenho;

ABSTRACT

Most modern processors have a multilevel cache hierarchy to mitigate main memory access latency. In addition, these processors also have an increasing number of cores, as the number of cores increases, more cores and threads share the Last Level Cache (LLC), which consumes a lot of energy and area of the chip. Therefore, while the size and number of cores increases, new solutions should ensure the best use of resources by reducing cache conflicts and cache pollution issues. This dissertation explores the knowledge that some applications have low temporal and spatial locality during data accesses. For these cases, LLC imposes an extra latency barrier to accessing the data in main memory as well as a waste of energy when installing the cache lines. Thus, bypassing one or more cache levels can benefit such applications, improving overall system performance and lowering power consumption. In this dissertation, we propose an online predictor to dynamic adapt the use of LLC. Our mechanism evaluates each execution phase by collecting hardware counters enabled by sampling mechanisms and using a linear regression model capable of identifying whether LLC benefits application performance or not, adapting LLC usage for each running application. In this way, decreasing the execution time in a single execution, in addition to increasing the overall performance of the system. This cache bypass also creates protocol coherence challenges in multi-application runs. However, our mechanism is capable of guaranteeing this coherence. Our results for single program executions show possible performance gains of up to 22%. As for the execution of multiple programs, they show that gains of up to 21% in execution time can be achieved and also a reduction in the miss rate in the LLC, resulting from the reduction of pollution and conflicts in the cache.

Keywords: Last-level cache; Linear regression; Performance prediction;

LIST OF FIGURES

LIST OF TABLES

Acronyms

ATD Auxiliar Tag Directory. 30

CAT Cache Allocation Technology. 32

CMP Chip Multiprocessor. 31

CPU Central Process Unit. 24

DRAM Dynamic Random Access Memory. 15, 16, 24, 30

DVFS Dynamic Voltage and Frequency Scaling. 17

DYCA Multi-core Dynamically Adaptable Cache Bypassing Mechanism. vii, viii, xiv, 18, 19, 32, 35, 36, 39–42, 46, 48, 49, 51–58, 64

FIFO Fist In, First Out. 20

GAM Generalized Additive Models. 26, 39, 42, 57

GPU Graphics Processing Unit. 24

HPKI Hits per Kilo Instructions. 30, 37, 43

IPC Instructions per Cycle. ix, 18, 31, 36–39, 42–44, 46, 50, 55, 64

LFU Least Frequently Used. 20

LLC Last-Level Cache. vii, 15–19, 21, 24, 29–33, 35–37, 39–43, 46, 48–58, 64

LRU Least Recently Used. 20, 22, 23, 30

MOESI Modified, Owned, Exclusive, Shared, Invalida. 24

MPB Message Passing Buffers. 30

MPKI Misses per Kilo Instructions. 30, 37, 38, 43, 52, 53, 56

MSE Mean Squared Error. 43–45

MSR Model-Specific Register. 31

NASA National Aeronautics and Space Administration. 15

NN Neural Network. 32

OrCS Ordinary Computer Simulator. 48

OS Operating System. 36

PC Program Counter. 31

PH Potential Hydrogen. 25

SRAM Static Random Access Memory. 15

TLB Translation Look-aside Buffer. 24, 41

Glossary

R-sq Coefficient of determination . 43

CONTENTS

1 INTRODUCTION

The architectural model proposed by John von Neumann (49) defines processing and storage units as separate components. In this model, instruction and data must travel from memory to the processor to be computed. Given this scenario, we face a problem inherent to this type of architecture: a performance limitation caused by the time consumption associated with transferring data between the processor and the memory (49). It is worth mentioning that the von Neumann architecture continues to be used in modern processors, which means this bottleneck remains a problem to be mitigated.

In 1991, the problem known as memory wall was formally described by National Aeronautics and Space Administration (NASA) scientists. Experiments were carried out to understand the behavior of a scientific application on Intel's iPSC/860. With these experiments, it was possible to notice the disparity between the latency of memory access and the speed at which the processor computed the operations (32).

This problem has intensified over the years, as memories have not advanced as quickly as processors. For example, when we analyze a historical clipping from 2004 to 2011, it is possible to notice a performance increase of 4.6× in processors, while for memories, the reduction in access latency is only 1.3× (8).

This way is possible to notice the importance of using a cache memory hierarchy to mitigate this latency difference since it brings the data with more potential to be used by the processor closer, decreasing its access latency.

When comparing Static Random Access Memory (SRAM) cache memories to Dynamic Random Access Memory (DRAM) main memories, we can observe that despite having lower latency, SRAM technology uses a more significant area per bit and, therefore, a higher monetary cost (20).

These cache memories, proposed in 1965 (50), use the temporal and spatial locality principles to create the illusion of a larger and faster memory despite its small storage area. The temporal locality principle is based on the fact that referenced addresses tend to be referenced again shortly. In contrast, the spatial locality principle relies on the observation that data with addresses close to the referenced data tend to be referenced soon also (20).

Considering their small storage capacity, when compared to main memory (i.e., DRAM), the data stored in caches must be chosen wisely. Otherwise, the effect is a polluted cache where data that will be shortly referenced is evicted, giving place for new data that will not be reused soon by the processor. Thus, polluted cache memories can increase data access latency as processor requests must fetch several levels of cache before being sent to main memory (20).

Possible solutions to reduce data conflict and pollution of caches were the creation of larger caches formed by bigger associative sets and better replacement policies to reduce the conflict of addresses. In addition, data prefetching techniques could also reduce the cache miss ratios (20).

Modern processors have several levels of cache memory. The adoption of this type of cache hierarchy aims to increase the possibility of the searched address being present at a level closer to the processor. Generally, the cache memory hierarchy has per-core private and shared levels. In most commercial solutions, we observe two private levels (L1 and L2) for each processing core, with an increasing latency and size. It also has a shared last level (Last-Level Cache (LLC)), closer to DRAM, that presents higher latency and size compared to the upper levels (i.e., L1 and L2) (20).

In general, a data request is performed sequentially in this memory hierarchy. For instance, if requested data is present in the first cache level, it will have the lowest possible access latency. Nevertheless, suppose the cache memory is not used efficiently. In that case, many cache misses may occur, and the data requests to DRAM will have higher latency due to lookup on all cache levels from the hierarchy.

Observing the cache hierarchies, we can observe a symmetrical and homogeneous structure between the different cache levels. In this way, each processing core within the same chip has an identical cache memory structure as the other cores.

Also, cache memories take up a large portion of space on the chip. For instance, in a microarchitecture such as Intel's Sandy Bridge, with eight cores and a LLC of 20 MB, 29% of the chip area is occupied only by LLC (42).

Besides, as the number of cores on the same chip increased, the pressure on the LLC also increased. Thus, we can observe a trend in modern processors, where the industry adopts larger shared LLCs formed by multiple independent and non-blocking banks.

1.1 PROBLEM

When considering a scenario formed by multiple processing cores and a cache memory hierarchy, we can see applications with poor cache usage can be negatively impacted by the latency of searching the data throughout the cache hierarchy before the request arrives in the main memory.

A fact to be highlighted is that applications that do not efficiently use the cache hierarchy present an additional latency in each access arising from multi-level searches and data insertion into the cache hierarchy. This additional latency worsens the overall performance of such applications (20).

According to Santos et al. (42), several benchmark applications improve performance when removing the LLC. Therefore the authors noticed that within this set of applications, some presented a data-streaming behavior where the memory cache was not very useful.

Furthermore, we can conjecture that scenarios with different applications competing for the cache memory could have a higher performance impact. For example, consider two applications A and B. Application A performs data streaming and presents low data reuse. On the other hand, application B presents temporal and spatial locality in the memory accesses. In this scenario, we can assume that application A will be slowed by the cache hierarchy, as presented earlier. In addition, this application will pollute the cache harming the application B.

Thus, there is a clear need for studies evaluating the use of adaptable cache memories in conjunction with a decision-making algorithm to extract the best performance from each application considering their access pattern.

1.2 MOTIVATION

Thinking about processing requirement changes over time, we recall the asymmetric multi-core architectures. These architectures are also known as single-ISA heterogeneous multi-cores, which are a solution that allows adaptation between high-performance or reduced power consumption cores (39). An example of this type of architecture is the ARM big.LITTLE (12).

The ARM "big.LITTLE" architecture has two types of processing cores. The big cores have higher processing power and energy consumption. On the other hand, the LITTLE cores present lower processing power and energy consumption (39). This architecture focuses on the smartphones and tablet market. It can map the application with intensive computation, as

observed in mobile games, and low computation, with tasks like emails or standby cell to different cores(6).

The correct mapping of applications between these heterogeneous cores is crucial for these architectures. The first techniques developed mapped high-memory applications to smaller cores and compute-intensive applications to larger (39) cores. However, some research shows that this type of mapping does not lead to the optimal solution and proposes more advanced techniques performed during execution that allow the dynamic migration of applications. Van Craeynest et al. (48) proposes a mechanism that, while executing the application in a specific core, collects indicators that allow simulating the execution in the other type of core, migrating the applications to the cores that best adapt to the observed and simulated profile.

These heterogeneous architectures amplify their asymmetry with techniques such as Dynamic Voltage and Frequency Scaling (DVFS) to reduce energy consumption. DVFS dynamically adjusts the processor voltage and frequency to the needs required by the application.

Like this idea, adaptable caches can improve performance and reduce energy consumption by adapting the cache hierarchy based on the application's behavior, finding the correct cache configuration to improve performance avoiding any degradation (31).

It is important to note that even if the cache layer is shared between the different cores, each running application may present a different behavior that needs to be considered while adapting the use of the LLC. Adapting the use of a cache layer can be accomplished in several ways, such as gated-Vdd, which adds sleep transistors to cut down leakage power but loses data stored in SRAM cells (37). Another way is bypassing, with can be done in a complete cache layer, bypassing all the accesses on this layer.

Thereby, two possible improvements can be reached by developing an approach to adapt the use of the shared cache layer using the bypass. First, when a single application runs, the leakage energy consumption can be reduced by turning off the cache layer. More than that, when multiple applications execute together, bypassing the ones that present poor temporal and spatial locality behavior can improve the average system performance as the cache conflicts and pollution problems are mitigated. For instance, Figure1.1 illustrates two applications: App *A* running in Core 0 and App *B* running in Core 1. While App *A* has performance benefits when using the LLC, App *B* do not show benefits from using the LLC. In this scenario, if App *B* bypass the LLC, both App *A* and App *B* may improve the performance since, App *B* will not bring non-reusable data into the LLC.

The central point of this proposal is developing an adaptable LLC, based on the observation that different applications have different preferences for caches. Even the same application can show a different behavior between running phases. Bypass is a well-explored technique to adapt the cache use. Most works use it in a fine-grained (e.g., instruction-grained) and do not adapt the use of LLC for each application. Together with the rise of heterogeneous architectures, such as *big.LITTLE*, in each, the application matches the best architecture for it. We claim that such heterogeneity can be extended for cache memories using an adaptable cache mechanism, leading to an improvement in average system performance.

1.3 PROPOSAL AND OBJECTIVES

In this work, based on the assumption that some applications benefit from systems without LLC, we intend to answer the following research question: **Can we improve the execution time of computer systems using a dynamically adaptable LLC memory?**

Thus, this dissertation initially aims to evaluate the influence of the cache memory system on isolated applications and in groups of applications sharing the LLC. At this point, we

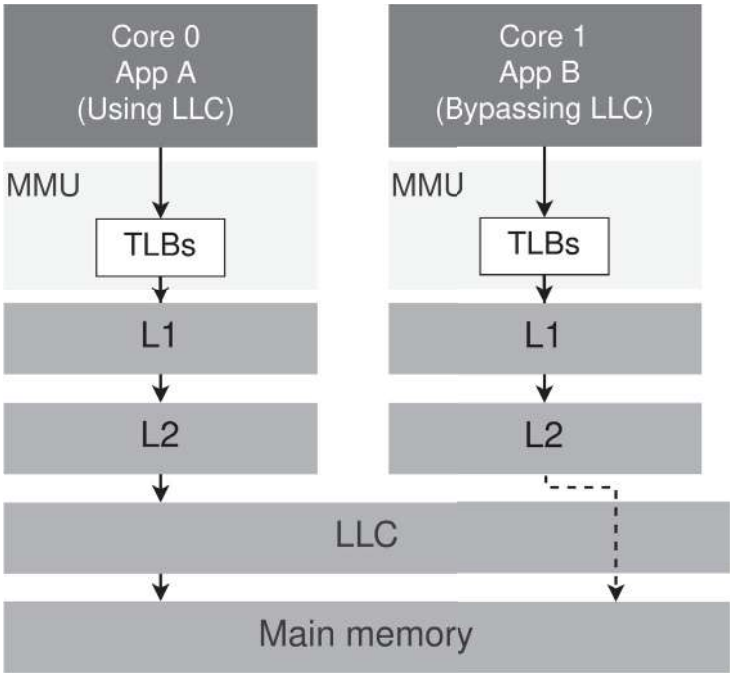


Figure 1.1: Main function idea applied in two cores (0 and 1) while core 0, running application A, is using the LLC the core 1, running App B, is bypassing the LLC.

Source: Modified from Hennessy and Patterson (20).

propose using a dynamically adaptable cache capable of adapting the cache according to the application behavior.

Thus we present Multi-core Dynamically Adaptable Cache Bypassing Mechanism (DYCA) a dynamic adaptive cache, supporting the dynamic adaptation of LLC usage through a decision-making mechanism capable of making adaptations considering the dynamically harvested memory access behavior of applications and different program phases. Our mechanism relies on the fact that applications tend to change between phases slowly. Therefore we analyze the past observed window to predict the next execution window.

In a second moment, we intend to increase the complexity of the mechanism and add support for multi-program to make decisions about the cache memory system, considering the pattern of all programs running.

As a result of the dynamically adaptable cache memory development, this dissertation has the main contributions:

1. **To create a linear regression model** : capable of predicting the future Instructions per Cycle (IPC) based in hardware counters from the past;

2. **To produce a single program bypass mechanism**: identifying program phases and adapting the use of cache according to the application behavior;

3. **To develop a multi-program bypass mechanism**: in an application granularity, adapting the cache hierarchy according to each application individually, proving benefits to all applications running;

As a result, it was possible to conclude that a dynamic adaptable LLC may bring more opportunities to adapt the architecture to the real application needs. This leads to an increase in the overall efficiency of the system.

Another problem that could be mitigated with DYCA is the cache pollution or cache noise, observed when there are lines in the cache that are not going to be reused shortly, which can cause premature evictions in other useful lines. With DYCA, we could observe gains up to 21% on average system performance in a multi-program execution, reducing cache pollution and conflicts. Additionally, in single executions, we also observed performance gains reaching 22%. Not only improving performance but also a reduced number of misses was observed.

Besides, DYCA solution gets close to the best possible LLC use configuration, even considering the performance overhead inherent in a dynamic mechanism. Also, the additional hardware is negligible compared to other state-of-art solutions.

1.4 DOCUMENT ORGANIZATION

The rest of this dissertation is organized as follows: Section 2 deals with the fundamentals necessary for a complete understanding of the text, namely, the concept of memories cache, the concept of adaptable cache architectures, the cache bypass concept, and a section about the linear regression model. In Section 3, a review of the available bibliography for the topic is carried out. In contrast, in Section 4, the proposal mechanism, named DYCA is described, and in Section 5, all experiments and validations are discussed. Finally, Section bring the final discussion and conclusion provided by the development of this work.

2 BACKGROUND

This chapter discusses the fundamental concepts necessary to understand this proposal fully. The first section deals with cache memories, the second with bypassing requests in cache memory, the third with adaptable cache architectures, and the fourth with linear regression models.

2.1 CACHE MEMORIES

Cache memories introduced in the 1960s are small memories closer to the processor. This type of memory uses the principle of spatial and temporal locality, existing in most applications.

Due to their reduced storage space, mapping techniques are used, efficiently allowing many blocks in the main memory to be mapped to a reduced set of blocks (cache lines) in cache memory (20).

2.1.1 Address mappings

The cache memory mapping defines how to install and look for each specific address. Directly mapped caches (Figure 2.1 (a)) present a unique location where each memory address should be mapped. The fully associative cache (Figure 2.1 (b)) has a mapping where each location can store any memory address. There are downsides related to the techniques listed above. Direct mapping usually has a simple implementation and a higher address conflict ratio due to its restriction on address installation. While fully associative mapping presents lower cache conflicts and higher hardware costs associated with the number of comparators used to verify if the block is present or not, in addition to increasing the complexity during data replacement (20).

The technique that strikes a better balance between the direct and fully associative mapping technique is the set-associative mapping design (Figure 2.1 (c)). In this model, each address is mapped to a set of cache lines, and each set has n ways (this n has a bottom limit of two). This mapping strategy is called n ways set associative cache (20). This type of mapping presents associative ways which reduce the address conflict, considering that n addresses mapped to the same set can be stored simultaneously. In addition, in the set-associative mapping, the hardware overhead is reduced, as only n address must be compared per access, presenting low complexity cache line replacements (20).

2.1.2 Replacement policies

In addition to the mapping technique, another important aspect that contributes to the performance of the cache memory architecture is the line replacement policy used. During a cache miss, this policy is responsible for choosing which block will be replaced by the new address to be stored (7).

Some of the best-known policies are: Least Frequently Used (LFU), Fist In, First Out (FIFO), and Least Recently Used (LRU). The most commonly used technique is LRU, where the block replaced is the one with the longest time interval between the last reference and the current time (20). The removal is done based on the least recently used cache line.

Nevertheless, perfect LRU policies are challenging to implement for memories with large associative sets. Often, such a policy leads to approximate implementations, which bring a balance between accuracy and implementation cost (25). These policies usually follow the principle of temporal locality, which says that data tends to be reused in a short time.

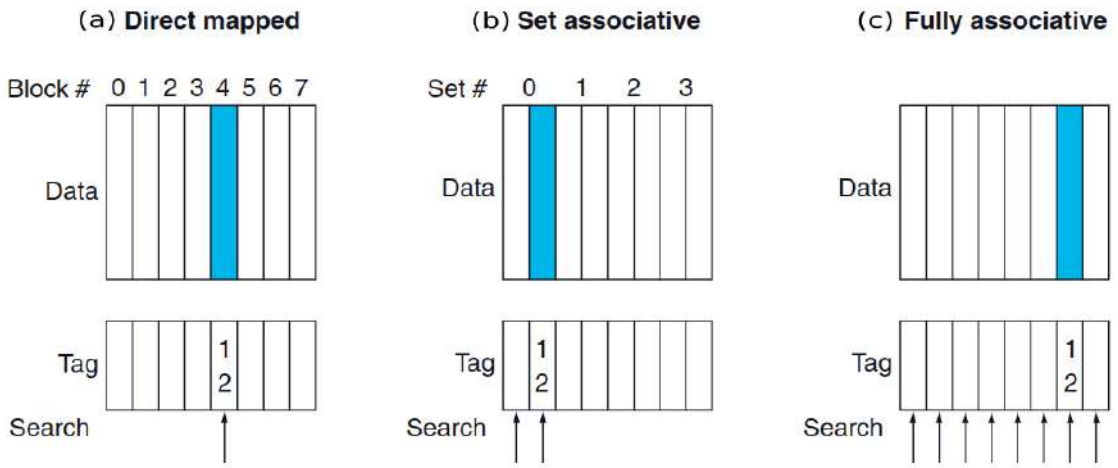


Figure 2.1: Different cache geometries (a) Direct mapped, (b) Set associative and (c) Fully associative

Source: Modified from Hennessy and Patterson (20).

2.1.3 Cache hierarchy

Modern processors tend to adopt a multi-level cache architecture. For example, the most common architecture nowadays for desktop and server processors consists of two private levels to the core (L1 and L2) and a shared level between the different cores (Last-Level Cache (LLC)).

When a read or write is performed, data is fetched sequentially in cache memory from levels close to the processor, which have lower latency, such as L1, to the furthest levels. If the data is found, there is a cache hit, and the cache sends the data to the processor. Otherwise, the data must be fetched at more distant levels, with higher access latency, such as LLC. Finally, if there are no cache hits at any level of the cache hierarchy, the data is fetched from the main memory or disk. This hierarchical architecture works as an access filter since a percentage of the accesses will be answered by L1, another sub-percent by L2 and only a smaller portion of requests will be sent to the LLC (20). After a cache miss, the address is installed in all cache levels closer to the processor, following the temporal principle.

By observing the pattern of access to addresses by several applications, we can observe that most applications have a well-behaved data access pattern. Using the spatial locality pattern, for instance, during linear accesses to a vector, and the temporal locality during the access of the control variable inside a loop. Nevertheless, some programs do not benefit from these concepts. For instance, some graph applications use large amounts of data in a non-linear way (presenting low spatial locality) and do not reuse most of them (20), presenting low temporal locality. Therefore, they do not benefit from cache memories. We could also cite data streaming applications with low or no temporal locality, making the cache usage less fruitful.

2.1.4 Inclusiveness policies

When dealing with multiple cache levels, we need to decide which storage policy to adopt between the different levels. There are generally three policies: exclusive, inclusive, and non-inclusive. In the exclusive storage policy, data is present in only one cache level. For this, during insertion, the data are initially on the first level (L1), and as they are replaced, they move towards LLC. The inclusive storage policy replicates the data in all cache levels farthest from the processor. Therefore, any address in a cache level closer to the processor must be at all levels further from

the processor (e.g., any address present in L1 must be in L2 and LLC). For addresses removed from L1, they stay in L2 and LLC. For this to occur, when inserting data, insertion is done at all cache levels. Finally, in this technique, when removing data from the last level, an invalidation of the higher levels (i.e., closer to the processor) must be performed (47).

We notice that inclusive policy waste storage space in cache memory, but maintaining coherence is more convenient in this model. On the other hand, the exclusive policy allows full use of space but makes maintaining coherence a more tricky task.

Finally, in the non-inclusive policy, the address installation follows the inclusive policy. However, removing an address from a certain level does not imply removing it from other levels. Thus, the inclusion of addresses is not guaranteed (47).

Some commercial examples use different policies within the same processor. For example, Intel Sandy Bridge has a non-inclusive L2 cache and an inclusive LLC (23). In another commercial example, the ARM Cortex-A15 (5), the L2 cache is inclusive, and the LLC is exclusive, whether it is present or not. The AMD Ryzen 9 5950X-A has an inclusive L2 and a non-inclusive LLC.

2.1.5 Non-blocking cache memories

In order to increase performance, non-blocking caches are a way to increase the bandwidth of a cache. Non-blocking caches are a natural feature for out-of-order execution processors. These caches allow cache hits to be responded to even while a cache miss is pending (19).

This type of optimization, called "hit under miss", reduces the miss penalty, considering that other requests from the processor are not delayed when a miss occurs (19). In addition to this optimization, there are two more complex optimizations: "hit under multiple misses" and "miss under miss." In these two, there is the transposition of multiple misses. In the second scenario, "miss under miss," the memory system must be able to respond to multiple misses. Commercial examples are present in processors such as the Intel Core i7 and the ARM A8 (19).

2.1.6 Multi-bank cache memories

Another architecture used in modern chips like the Cortex-A8 and Intel Core i7 is the multi-banked caches. This architecture divides the address space into multiple banks, which can be accessed individually in parallel. The efficiency of this technique is linked to the type of mapping performed. The intention is to guarantee that the cache accesses are equally distributed among the multiple banks; otherwise, there will be a problem of bank conflict, multiple addresses mapped to the same bank, causing a higher contention. (19).

Sequential interleaving is a form of addressing that guarantees good performance, consisting of assigning the addresses of the blocks sequentially between the banks, as shown in Figure 2.2. In an example with four banks, two bits from the memory address indicate which bank stores each specific address.

2.2 BYPASS IN CACHE MEMORIES

We commonly consider that bringing data to the cache typically results in a performance gain (11). Despite this, it is trivial to present cases where the execution time is worse using cache memories than without them.

For example, we can assume a fully associative cache composed by two lines only, with a line size of one byte and a LRU replacement policy. Then, consider the accesses to blocks

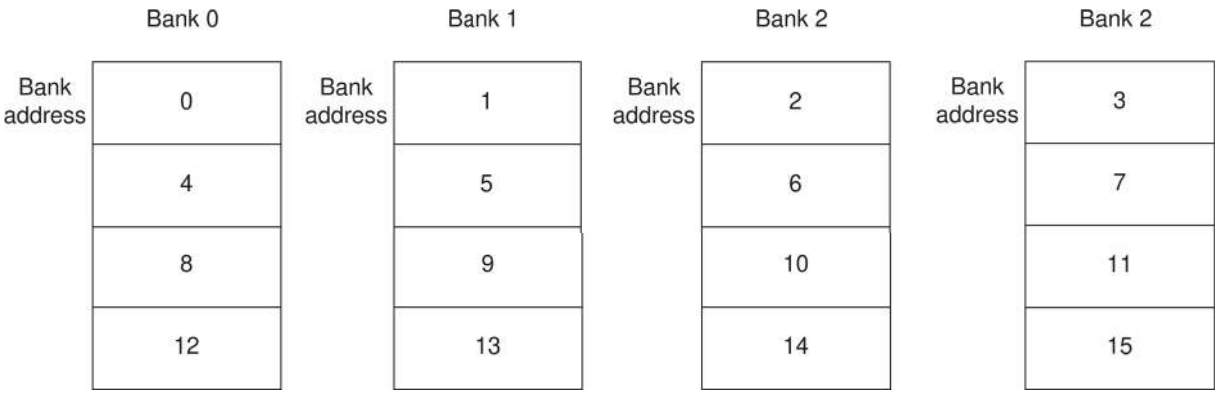


Figure 2.2: Four cache banks that feature sequential interleaving block addressing. Bank 0 presents the block addresses whose module 4 result is 0 (0, 4, 8, and 12), bank 1 contains the block addresses with result 1 when done modulo 4 (1, 5, 9, and 13), bank 2 the block addresses with result 2 (2, 6, 10 and 14) and bank 3 the block addresses whose module 4 has result 3 (3, 7, 11 and 15).

Source: Modified from Hennessy and Patterson (19).

(1,2,3,1,2,3) commonly observed in applications with loops. The representation of this execution is in Figure 2.3.

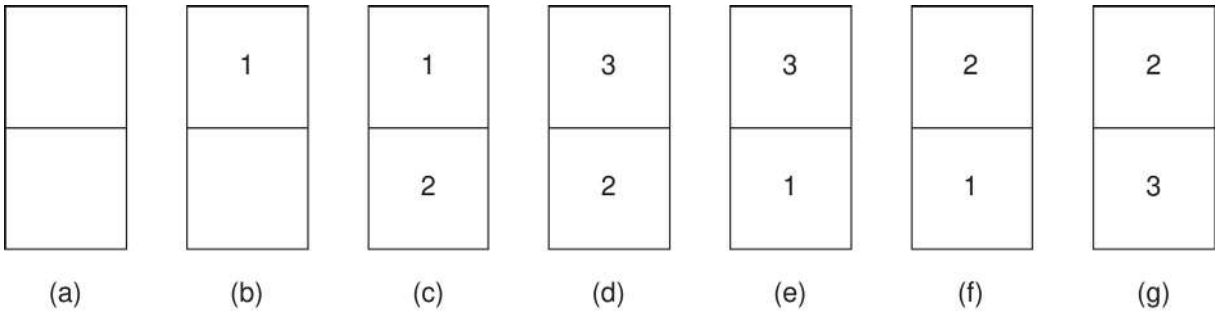


Figure 2.3: Execution of access sequence for blocks (1,2,3,1,2,3) in a fully associative cache of size 2 and substitution policy LRU, initially empty. In (a) the empty cache, in (b) the first access to "1", in (c) the first access to "2", in (d) the first access to "3", in (e) the second access to "1", in (f) the second access to "2" and in (g) the second access to "3".

Source: Translated and modified from Chi and Dietz (11).

After accessing the blocks (1, 2, 3, 1, 2, 3), the result is six cache misses, the worst case scenario (11). Considering that misses generated extra costs associated with the tasks of 1) fetching data from main memory, 2) saving to cache, and; 3) selecting which data should be removed in cases where the cache is full.

In the presented scenario, there is no possibility of not inserting data into the cache. Considering the possibility of bypassing the cache memory, a scenario where the access to block 3, was sent directly to the main memory would result in four misses and two hits (11).

We can argue that such a scenario is unrealistic because it uses a small cache. However, we must remember that when scaling the size of the cache, we can also scale the size of the problem to be solved, reaching the same scenario on a larger scale.

A second argument we can think of is the efficiency of the cache's line replacement policy. In this sense, a different policy could lead to a better cache hit ratio. In the example of Figure 2.3, for instance, using a Most Recently Used (MRU) policy could decrease the number of misses to four.

The technique of bypassing cache memory during requests consists of bypassing addresses with low reuse probability from certain levels of cache memory. In this way, the cache theoretically has a lower amount of pollution. In other words, it presents a smaller amount of addresses that will not be reused (27). Also, reducing the overhead associated with fetching and storing data in the cache memory as the latency of cache memory levels is removed.

This bypass strategy can be performed at all cache levels, directly accessing main memory (11). Alternatively, it can be performed at specific levels (27).

This bypass can be executed statically, before the application execution (11), or dynamically, during the execution of the workloads (27). Some related work (9; 14; 27) use metrics to define when an address shall bypass. Some examples are: building an execution flow and, calculating the cost to insert or remap data into the cache, using an extra bit to signal when an address should be installed in the cache memory hierarchy. Another technique used is the reuse counter, incremented when an address provides a hit in LLC. The counter value defines when a request should bypass the LLC (27).

Besides the use in Central Process Unit (CPU), the technique of *bypassing* has been used in other architectures, for example, in Graphics Processing Unit (GPU) (51).

Other proposals in the literature present bypassing techniques from LLC (29), (27). In the work of Köhler and Alves (29), the data request identified as possible *misses* in LLC is bypassed. In parallel, it sends a request to the main memory system, thus reducing data fetching costs. However, this proposal does not consider multi-threading systems or a layer cache bypass.

One of the challenges in the use of bypassing techniques is the control of the cache memory coherence protocol. Since, change cache accesses can lead to a non coherence stage of the cache. The protocol can be simplified using the inclusive policy, although bypass is also applied to exclusive caches, using for instance a coherence directory array disconnect from the cache (16). Changing the coherence protocol is not trivial, because of that works have different approaches, adapting the mechanism in a way where there is no need to change the protocol (52), using the Modified, Owned, Exclusive, Shared, Invalida (MOESI) protocol writing back dirty cache lines when they are invalidated (21).

Modern processors provide non-temporal load and store memory instructions. In this type of instruction, the access is done to a uncacheable region at the L1 level for address translation by Translation Look-aside Buffer (TLB). After the address translation, they are sent to the main memory Dynamic Random Access Memory (DRAM), thus saving access latency. (22).

2.3 REGRESSION

Regression models are statistical techniques that enable us to assess the impact of explanatory variables over response variables by estimating quantities that measure this effect. If these quantities are significantly different from 0, there is evidence of a significant effect of the explanatory variable over the response. This impact evaluation uses hypothesis tests considering the probability distribution of the estimates. Using these estimates in a regression model, it is also possible to make predictions of the response based on the observed values of the explanatory variable.

2.3.1 Linear regression

The commonly used model assumes that a linear relation exists between the explanatory variables and the outcome - the variable to be predicted by the model. With this linear model it is also possible to measure the strength of the relation between these two variables, named R^2.

In Figure 2.4, we can see one example of a linear regression between the cider Potential Hydrogen (PH) and the apple content in kilogram. In this example is possible to model the relation with a straight line, as illustrated in Figure 2.4(b), in a satisfactory nevertheless not optimal way.

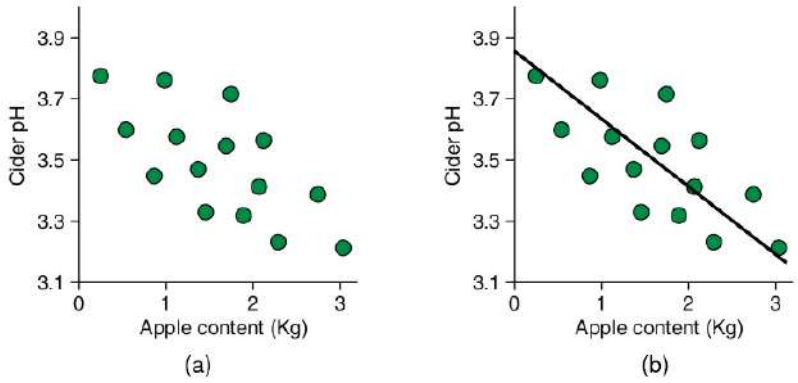


Figure 2.4: Linear relation between the pH of cider and the apple content (Kg).

Source: Rhys (41).

A limitation present in this model arises from the assumption of a linear relationship between the observations and also the use of a normal distribution(Clark).

2.3.2 Polynomial Regression

A common model used when the variables of study do not show a linear relation is the use of a polynomial regression model (Clark). In Figure 2.5, the same example of Figure 2.4 is illustrated. However, a quadratic term is added to the function, shown in blue, better capturing the relation between the variables.

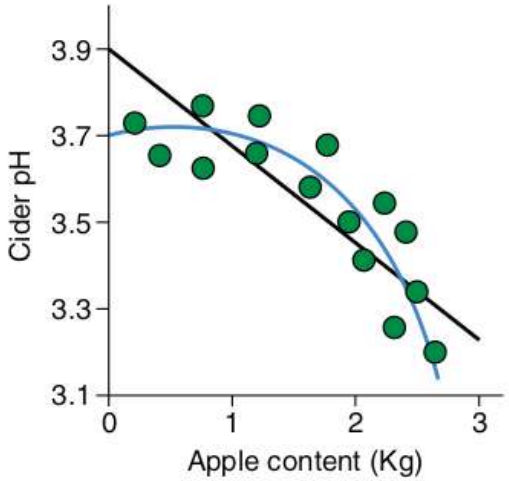


Figure 2.5: Polynomial quadratic relation between two variables.

Source: Rhys (41).

Although, with complex relations, it is difficult to fit the data with a quadratic, cubic, or higher level polynomial function.

2.3.3 Generalized Additive Models

Assuming that most real-world problems do not show a linear correlation between the variables analyses and the ones where the relationship is too complex to be represented with a polynomial equation, Generalized Additive Models (GAM) is an option to represent these relations.

GAM is a model capable of approximating a polynomial equation, using linear relation, from the predictor variables and the outcome (41).

In order to fit complicated relations between variables, use a higher degree of a polynomial. The higher the degree, the higher flexibility it will get, adapting better to the data. It can be a trick since better adaptation can lead to a overfit model (41).

Another way of adapting to a non-linear relation is using splines. Splines are piecewise polynomial functions, meaning each region is learned as a different polynomial function. The limitation for each region is known as knots, as shown in Figure 2.6.

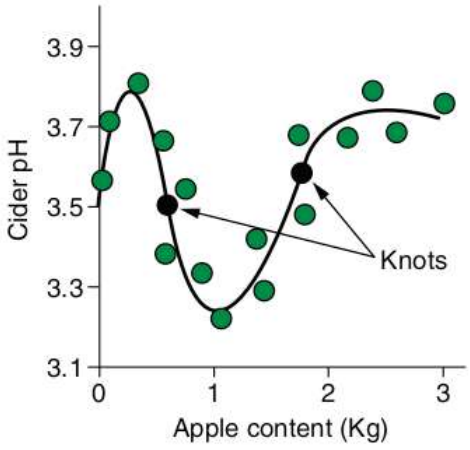


Figure 2.6: Example of a nonlinear correlation function of two variables (Apple content and Cider PH) fitted by a spline model.

Source:Rhys (41).

Although splines fit well in complicated relation, it has some limitation, seeing that all the knots and the degrees of each spline need to be chosen manually. GAMs models are more flexible, using a function for each predictor variable, most of the cases represented by a smoothing function or a combination of multiple splines (41).

3 RELATED WORK USING SYSTEMATIC MAPPING

In order to assess the main contributions present in this dissertation, it is necessary to observe the related work in the area, thereby making it possible to understand the current state-of-the-art and evaluate our work with the comparable ones available in the literature.

We used the Systematic Mapping methodology proposed by Petersen et al. (38) to find the most relevant related work. This methodology guides us while searching the correlated work by defining metrics and steps to make the process more accurate and ensuring that all essential work for this dissertation was considered without a personal author bias. The following sections discuss all the developed steps. In Section 3.1, we describe the research question, also the scientific databases and search strings used. In addition, the inclusion and exclusion criteria are also discussed. Section 3.2 describes in detail the selected papers from Section 3.1. Furthermore, Section 3.3 makes a comparison between the most related papers, and Section 3.4 provide the main similarity and differences between the papers analyzed with the presented dissertation.

3.1 SYSTEMATIC MAPPING MYTHOLOGY

According to the systematic mapping of Petersen et al. (38) one of the first steps is to describe a group of analysis questions.

Considering the goal of this dissertation, we developed the eight questions below to understand the area of reconfiguring cache memories and how to create a new dynamic and efficient tool for this task. The questions are the following:

1. RQ1: What is the primary objective of using a tool to reconfigure the cache architecture?
2. RQ2: How is the cache architecture, and what level is used?
3. RQ3: What technique is used to perform the reconfiguration?
4. RQ4: Which hardware components are added?
5. RQ5: What is the granularity of the change?
6. RQ6: Is it a dynamic tool?
7. RQ7: Does the research improve performance?
8. RQ8: Is hardware implemented?

These questions aims to extract the acknowledgment and the main objective of the tools analyzed. Gathering information about which cache level is used is the study, if is a dynamic and online tool, as well as the granularity of the change. Also, identifying how this tool is made: what components are added and if these components are hardware implemented. Moreover, also evaluates the improvement in performance or energy consumption. Mapping the related works leaded us to understand how we can improve the state-of-the-art in this topic, and improve the performance gains compared to similar tools already reported in the literature.

We execute the search for correlated works on three distinguished bases: IEEE Xplore, ACM Digital Library, and Springer Link. All of these research bases provide search mechanisms that support a range of dates, and we have searched these bases using a range of 2016 until now.

We took this decision based on the survey of Mittal (33), with brings with all the most relevant work in the bypassing area. Therefore, all papers cited by Mittal (33) were merged with the result obtained by the search on all bases.

About the strings used, we search for "adaptable cache hierarchy", "cache hierarchy reconfiguration," and "heterogeneous cache hierarchy" combined with "performance prediction", "cache bypass", "linear regression," and "machine learning". These combined strings are shown in Table 3.1.

Table 3.1: Search strings used in each search base

Search base	Date	Search string
IEEE Xplore	10/02/2022	(("heterogeneous cache" OR "cache heterogeneous" OR "adaptable cache" OR "cache adaptable" OR "cache reconfiguration" OR "reconfiguration cache" OR "re-configurable cache" OR "selective cache" OR "exclusion cache") AND ("performance prediction" OR "bypass" OR "bypassing" OR "linear regression" OR "machine learning"))
ACM Digital Library	10/02/2022	(("heterogeneous cache" OR "cache heterogeneous" OR "adaptable cache" OR "cache adaptable" OR "cache reconfiguration" OR "reconfiguration cache" OR "re-configurable cache" OR "selective cache" OR "exclusion cache") AND ("performance prediction" OR "bypass" OR "bypassing" OR "linear regression" OR "machine learning"))
Springer Link	10/02/2022	(("heterogeneous cache" OR "cache heterogeneous" OR "adaptable cache" OR "cache adaptable" OR "cache reconfiguration" OR "reconfiguration cache" OR "re-configurable cache" OR "selective cache" OR "exclusion cache") AND ("performance prediction" OR "bypass" OR "bypassing" OR "linear regression" OR "machine learning"))

Gathering the papers from IEEE Xplore, ACM Digital Library, Springer Link, and the papers cited by Mittal (33) we found a total of 231, 71 from IEEE Xplore, 48 from ACM Digital Library, 22 from Springer Link, and 90 from Mittal (33), described in Table 3.2.

Table 3.2: Number of papers per database

Database	Search Results
IEEE	71
Springer Link	22
ACM	48
Mittal (33)	90

In the next step discussed in Petersen et al. (38), some selection is necessary to make the number of papers smaller and select the ones with higher relevance to our proposal. To do this, we define the inclusion and exclusion criteria, listed in Table 3.3, to guide the selection process.

Papers that use GPU and those that improve the replacement policy were excluded. Other exclusion criteria consider papers that only show the software specifications or perform

reuse distance prediction to indicate dead blocks or to predict the load and store latency. In addition, we included papers focusing on CPU cache (any level) and bypass techniques.

Table 3.3: Inclusion and exclusion criteria

Inclusion criteria	**Exclusion criteria**
1. CPU cache 2. Use bypass technique to make the cache reconfigurable or adaptable 3. Reduce energy consumption or improve performance	1. GPU cache 2. Paper developed to improve replacement policy 3. If only software specifications are described 4. Prediction of reuse, dead blocks or load, and stores latency 5. Papers that focus on security solutions 6. Papers related to network solutions

3.2 STATE-OF-THE-ART

This section discusses this proposal's state-of-the-art in detail. However, first, we separate these papers into three categories: bypassing techniques, described in Section 3.2.1, Reconfigure and adaptable caches in Section 3.2.2, and Statistical and Machine Learning approach reported in Section 3.2.3.

3.2.1 Bypass techniques

Bypass is a well-established technique used as an approach for different types of problems. In the paper using bypass to adapt the use of the LLC we focus on the ones with a coarse-grained adaptation, such as adapting an entire cache level, the cache size, or the associativity. Rather than a mechanism with a fine-grained bypass, such as an instruction-grained mechanism.

The Kharbutli et al. (27) work uses cache block granularity, presuming that there is a large set of blocks that are not reused. These blocks can be classified according to the re-access range they have. Blocks with short re-access (i.e., frequently re-accessed) can be easily accommodated in the L1 and L2 cache levels. In addition to the short frequency, there are moderate and long frequencies. Long frequencies cannot be accommodated even by the larger LLC. However, blocks with moderate frequency can be allocated in the last level, not being successful if allocated in higher levels of smaller sizes. Therefore, moderate frequency blocks should be kept in LLC.

In order to perform the bypass, a method capable of detecting the re-access range of the block is therefore required. Therefore, a table of tags is implemented, containing a miss counter, incremented every time a miss of the block occurs. While this counter is smaller than 3, the block undergoes bypass; when the corresponding block counter reaches 3, the block is inserted into the LLC (27). The mechanism could detect the reuse behavior of the blocks at run-time. The result is 75% speedup in the best case (SPEC-CPU 2006 mcf application) and 18% of speedup on average, considering SPEC-CPU 2006. The authors also show that between 82% to 95% of the blocks were bypassed using this mechanism. Their algorithm implementation shows that it can eliminate up to 28% of cache misses in a non-inclusive LLC.

Gupta et al. (17) also proposes a solution to perform bypass although in inclusive caches. In this work, a buffer is used to store the tags of the cache lines that undergo bypass in the LLC.

When a tag is replaced in this buffer, the upper levels are invalidated, keeping the inclusion principle. This work demonstrated that for a relatively small buffer, with 512 entries in the largest test, it was possible to obtain the benefits of performing bypass on inclusive LLCs.

In the work of Köhler and Alves (29), the bypass of data requests identified as possible misses in the LLC is performed through a predictor of misses. For these accesses identified as possible misses, the DRAM and the LLC request is performed in parallel, thus decreasing latency cycles. Multi-threading was not considered to perform the bypass.

The work of Egawa et al. (14) uses the value of Hits per Kilo Instructions (HPKI) and miss penalty associated with the energy consumption of each cache layer in the system to decide if there is any layer that can be disabled without hurting the performance, using bypass and Gated-vdd techniques. However, in this related work, this decision is not adaptable during execution time.

After Liu et al. (31) used the mechanism proposed by Egawa et al. (14), thus adding an adaptable system to identify the better capacity and associativity to the LLC without losing performance or increasing LLC misses. The decision is made using the Misses per Kilo Instructions (MPKI) metric. The proposal reached an improvement of 10% on average in energy-efficient, although this is a static mechanism.

To finish, bypass can also be used to increase the replacement policy efficiency, Khan and Jimenez (26) uses a sampling cache and a decision tree to improve the efficiency of LRU policy in the LLC. They consider that L1 and L2 already filter part of the temporal locality accesses, so they aim to predict which block in LLC can be replaced and which one could be bypassed in the LLC. As a result, they show 5.2% better performance when compared to execution with only the LRU policy itself.

3.2.2 Reconfigure and adaptable cache

Partial shutdown techniques or reconfiguration of adaptable caches are used for various functions, such as reducing energy consumption when resources are not being used.

In cache reconfiguration techniques, caches are divided into several smaller sub-ways, and these sub-ways are activated or deactivated, according to the pattern found in the application, to obtain a better performance or reduce energy consumption.

Mittal et al. (34) focus on reducing energy consumption. Different sizes of LLC are tested and evaluated according to energy consumption metrics. The smallest possible size is selected for the LLC, considering an acceptable threshold of performance loss. The work showed a maximum energy saving of 15%.

Another work by Mittal et al. (35) also aims to minimize energy consumption. In this work, energy savings reached 26.2%. The technique used for the *profiling* of applications has a *Auxiliar Tag Directory (ATD)* to identify the general cache pattern, turning off associative paths from LLC that are being representative used and also not represent a huge improvement in application performance. This technique has little overhead, as ATD adds only a little extra storage space.

In addition, in reconfiguration mechanisms where the cache memory geometry is changed or some part disabled, some proposals allocate the unused cache space, during execution, to another purpose, as Lai et al. (30) and Han et al. (18).

Lai et al. (30) presents a hybrid model of the cache storing data and traces. The authors assume that for specific applications, not all the cache storage capacity is used. Cache ways from the associative set are reconfigured as trace buffers. On the other hand, Han et al. (18) proposes a reconfigurable cache, where the cache lines are reconfigured to Message Passing Bufferss (MPBs) according to need and usage. Promoting better use of storage within the chip.

El-Sayed et al. (15) proposed a reconfigurable partition of the LLC, profiling applications and making clusters of similar ones that share the same cache partitions, improving throughput by 24% on average. Moreover, the mechanism uses detailed profiling information, gathered offline or online. The online one uses a sampling technique while the offline executes the application in all different partition sizes.

Sato et al. (43) developed an associativity-adaptable cache architecture, using hardware-implemented perceptron. The features used in the perceptron are: Program Counter (PC); tag upper and lower addresses of the accessed block, and the PCs of the latest three instructions that cause memory accesses. Using this perceptron to predict dead blocks and prevent them from being installed by bypassing them, the unused ways from the set-associative can have the power supply interrupted. For this, sampling is used to maintain the cache metrics. The approach reaches a 14% of reduction in energy consumption using bypass, although they did not provide any performance improvement.

Navarro et al. (36) reconfigure three cache parameters: size; line size and associativity. To reach the ideal combination for the application, they use the program's frequency of assembly instructions to train a machine learning model capable of predicting the best combination of parameters in terms of energy consumption. Although they make this reconfiguration only once at the beginning of program execution, considering the features extracted offline from the application, not being able to adapt to program phases.

Commercial solutions support the shutdown of cache hierarchy levels, such as Intel processors based on the Intel NetBurst architecture, which allows the shutdown of the last cache level. This change is done by a bit 6 of the $IA32\_MISC\_ENABLE$ Model-Specific Register (MSR) and before any changes (disable or enable) software should disable and flush all the processor caches (23).

It is possible to notice that several techniques applied to Chip Multiprocessors (CMPs) were found in the literature. These solutions aim to reduce the dissipated energy of the cache memory hierarchies (21) and reduce the energy consumption and the execution time (10). To reach this, they reconfigure the parameters such as cache line size (10), storage capacity (34) or change the number of associative ways (35). In this sense, it is possible to notice that changing the geometry of the cache presents positive results in terms of reducing energy consumption and reducing execution time. However, our proposal differs from the related work described above by considering the entire LLC for each running application, dynamically adapting the use of the LLC on every context switch basis, thus, identifying and adapting for the different application phase.

3.2.3 Statistics and Machine Learning Approaches

Both bypass and reconfigure techniques can be done using statistics or machine learning approaches. Teran et al. (46) uses a hardware-implemented perceptron, using tables and reuse counters to estimate the reuse distance for cache blocks, bypassing and improving the replacement of the blocks, proving a speedup up to 18.3%. Although the work from Teran et al. (46) is based on a fine-grained bypass, it differentiates from the others by the hardware implementation of the machine learning model.

While Jain et al. (24) uses reinforcement learning to perform dynamic cache co-partitioning. The action space of the model is the possible reconfigurations (i.e., possible ways allocated for that cache layer). The model decides based on the Instructions per Cycle (IPC) observed, which is also the reward used, proving a model that tries to maximize the value of IPC. Kim et al. (28) also use machine learning to dynamically adapt the size of the shared LLC using

the Intel Cache Allocation Technology (CAT) by using the Lasso model to predict the IPC across different sizes of LLC.

And Al-Obaidy et al. (1) uses a neural network algorithm to reconfigure the LLC, consequently improving system energy efficiency on average 45.2% and performance by 13%. This paper predicts latency and change between a 16-MB SRAM and a 512-MB STT-RAM. In this case, LLC are shared between cores, and any change is applied to all cores far from our work that identifies each core demand.

Al-Obaidy et al. (2) uses a Neural Network (NN) to train a prediction model capable of predicting the demanded application's latency. Based on the latency, the authors reconfigure the LLC using emerging technology such as STT-RAM and determining the space of the LLC used by the application, not disabling the full use of the LLC.

Although some works do not use the bypass, they present a cache adaptation mechanism. Zhu and Zeng (52) use hardware counters and a decision tree to predict the best cache associativity. Adapting the L2 for the associativity find. Nevertheless, this decision is time-consuming since five execution phases are performed, each trying a different associativity configuration to make a decision.

3.3 OVERALL COMPARISON

Table 3.4 summarize this proposal's most relevant related work, comparing these work to each other and with Multi-core Dynamically Adaptable Cache Bypassing Mechanism (DYCA).

The aspects analyzed are in the first line of Table 3.4. First, we compare the technique used to adapt the architecture, derived from the research questions we assign six categories for each work, which can be Gated-vdd, bypass, the use of heterogeneous cores, or the reconfigure and adaptation of cache layers. We can also have a mixed of these techniques.

The second aspect is the granularity of change, we can see works with fine-grained, such as request and cache lines. Also, works with an entire cache level, size, or associative path as the granularity of adaptation.

The third and fourth are binary aspects, considering it is an online mechanism (i.e., if the changes are defined online during program execution or if they are done offline before the execution) and if the adaptation is performed and decided during run-time (i.e., if the changes designated in the previous categories can change during the execution).

The next two columns bring results about the hardware overhead needed and the performance gains observed. The aspects reported as N/D are not determined in the work. For the *reported speedup* it can be either N/D (i.e., the paper does not bring results about performance) or 0% in the case where performance is not increased or have a negligible reduction.

3.4 CONCLUSION

To conclude, our systematic mapping found many papers related to this dissertation in the three main areas: machine learning and statistic approaches, adaptable cache, and bypass. However, the results in the papers selected show possible improvements in performance and energy consumption reduction, no prior work provides a dynamic adaptable LLC mechanism in an application granularity.

Although Jain et al. (24); Egawa et al. (14); Sato et al. (43); Al-Obaidy et al. (1); Kim et al. (28) adapt the LLC configuration, this reconfiguration is applied to the whole system not aware of each application present in multi-uses systems. In addition, Egawa et al. (14) implements a not static mechanism, unable to adapt to program phases. Furthermore, the previous techniques

Paper	Technique	Grain	Online	Run-time	Hardware overhead	Reported speedup
(26)	Bypass	Cache line	•	•	N/D	12.0%
(48)	Heterogeneous cores	Process	•	•	N/D	5.5%
(27)	Bypass	Cache line	•	•	32KB	9.0%
(34)	Gated-Vdd	Cache parameters	•	•	1000KB	−2.0%
(35)	Gated-Vdd	LLC associativity	•	•	512KB	0%
(46)	Bypass	Cache line			10KB	18.3%
(43)	Gated-Vdd and bypass	Cache line	•	•	44KB	0%
(14)	Gated-Vdd and bypass	Cache layer	•		N/D	0%
(29)	Bypass	Request	•	•	96B	13.3%
(1)	Cache parameters	Cache layer		•	N/D	13.0%
(31)	Gated-Vdd and bypass	Cache layer	•		N/D	0%
(36)	Adaptable cache	Cache parameters			N/D	N/D
(2)	Adaptable cache	Cache parameters	•		N/D	25.0%
Ours	**Bypass**	**Application**	•	•	**55KB**	**22.0%**

Table 3.4: Summary of most relevant papers.

using bypass techniques commonly do it in a fine-grained (e.g., instruction-grained) (29), in contrast, our mechanism makes decisions on every context switch basis.

4 DYCA

Our proposal is an application-aware dynamic mechanism using cache bypass to adapt the usage of LLC in the cache hierarchy. Consequently, we need an online mechanism to adjust the cache during run time.

Multi-core Dynamically Adaptable Cache Bypassing Mechanism (DYCA) principles depend on a learning phase and identification and run-time phase. First, in the learning phase, we use metrics from SPEC CPU 2006 (44) to train the models. On the one hand, **wLLC**, which stands for With LLC, predicts the IPC when LLC is used. On the other hand, **woLLC**, which stands for Without LLC, predicts IPC when the use of LLC is disabled (i.e., bypass is performed). Both use metrics collected for each execution window.

Second, at the run-time stage, DYCA is performed during program execution, deciding the best cache configuration for the next application's execution window based on the IPC predicted by the functions **wLLC** and **woLLC**, using the hardware counter from the last execution window. This way, we provide an online and dynamic mechanism to adapt to multi-program workloads and detect the different program phases. This process is illustrated in Figure 4.1.

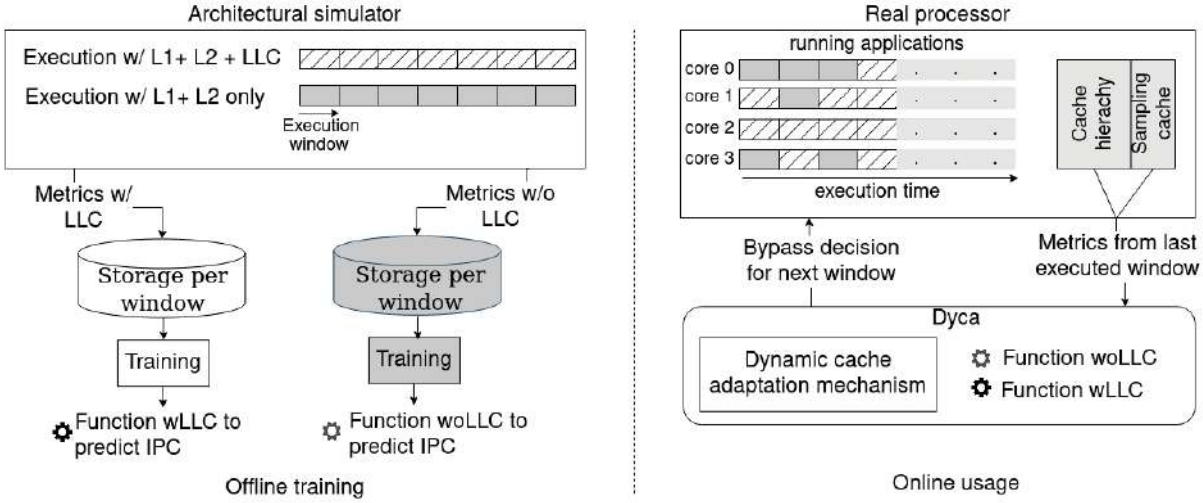


Figure 4.1: General view of DYCA operation. In (a) the offline training where the metrics of two executions - one with only L1 and L2 caches and another with all cache levels (L1, L2, and LLC) - are used to train two functions to predict the application's IPC with and without the LLC and in (b) the online usage where the metrics from each window execution are used to predict the IPC of the next window and decide to use or not the LLC because on the previous calculated IPC.

The following sections will detail the mechanism. We first start describing the process of collecting the metrics during execution, Section 4.0.1. After, the hardware counters collection and selecting the most relevant ones in Section 4.0.2. Next, we describe the sampling cache method used to keep the LLC counters when bypassing the Last-Level Cache (LLC) (40) in Section 4.0.3. After that, the architecture configuration is described in Section 4.0.4. And the linear regression model is explained in Section 4.0.5. To conclude, we discuss the performance and hardware overhead in Section 4.0.8 and the general flow of our mechanism DYCA in Section 4.0.9.

4.0.1 Metrics collection via execution window

Since programs present different phases, adjusting the LLC cache usage or not for each phase is necessary. Therefore, we adopt the execution window approach to make it possible. An execution window consists of a slice when the program is being scheduled for execution by the operating system. Our approach defines the execution window as 200 million cycles long for simulation purposes. This decision is based on the average frequency that a Operating System (OS) context switch happens (3). The execution window being the OS context switch slot reduces the performance overhead.

The main idea is to execute DYCA switches between cache and cacheless execution simultaneously with the context switch. Thus, our mechanism hides most of its required overhead due to the cache invalidation needed to guarantee the coherence protocol. It happens because, during a context switch, most of the cached content is likely to be invalidated by the other applications' execution.

The metric measurements are performed at the end of each execution window, as described in Figure 4.2. This approach is used in the offline step to obtain hardware counters to train the model. Further, it is used during the online step to decide the configuration for the next execution window and change the cache hierarchy if necessary. It is essential to mention that this offline training step is executed for a series of applications for our model to infer based on the hardware counters when it is beneficial or not to use the LLC.

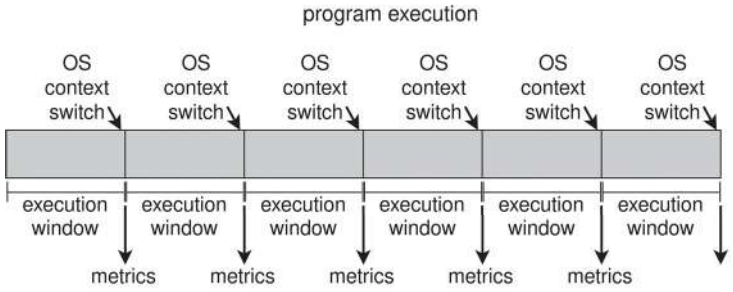


Figure 4.2: Execution window approach used in DYCA where the length of each execution window is equal to the average time in which an OS context switch happens, at this moment the metrics for these windows are obtained, and the counters reset to measure the next window.

4.0.2 Hardware counters and feature selection

In order to choose which hardware counters to use, we first consider the papers that use hardware counters to classify the efficient use of the cache (48; 14; 31; 15; 24; 28). Next, we exploit many hardware counters to select the ones with a higher correlation with performance. Table 4.1 shows all the hardware counters from the cache hierarchy and each core considered in this first evaluation step.

Since the state-of-the-art proposals use different hardware counters, we evaluate each counter fairly to decide whether or not each one will be considered in the regression model. This model is described in Section 4.0.5.

We use information collected from SPEC-CPU-2006 (44) and SPEC-CPU-2017 (45) to evaluate these metrics. All the hardware counters were collected in each execution window of 200 million cycles. We use Pearson correlation to measure the correlation between each hardware counter and the value of IPC.

Component	Hardware counters	Execution window	Full execution
L1, L2, and LLC	Number of misses	•	•
	Number of hits	•	•
	Number of accesses	•	•
	Number of writes	•	•
	Miss rate	•	•
	Hit rate	•	•
	Misses per Kilo Instructions (MPKI)	•	•
	Hits per Kilo Instructions (HPKI)	•	•
Processing Core	Instructions per Cycle (IPC) using LLC	•	•
	IPC without LLC	•	•
	Number of instructions		•
Memory system	Load	•	•
	Store	•	•

Table 4.1: Summary of features (i.e., the hardware counters) selected to be used in the linear regression model.

To analyze, we use the R and p-value. R indicates the strength and direction of the correlation, varying from 1 to -1. The closer to zero, the weaker the correlation. Positive R values indicate positive correlations (i.e., variables tend to grow in a directly proportional way), and negative values indicate a negative correlation (i.e., values are inversely proportional).

Furthermore, *p-value* value considers the null hypothesis, which defines that the variables are uncorrelated, and measures the potential to observe the specific obtained *R* value considering the null hypothesis. Most research defines statistically significant as a p-value equal to, or under 0.05 (41).

First, we evaluate the absolute counters, such as the number of accesses, hits, and misses. Figure 4.3 show the correlation graphics for the absolute number of misses in the LLC, with R-value equal to −0.21.

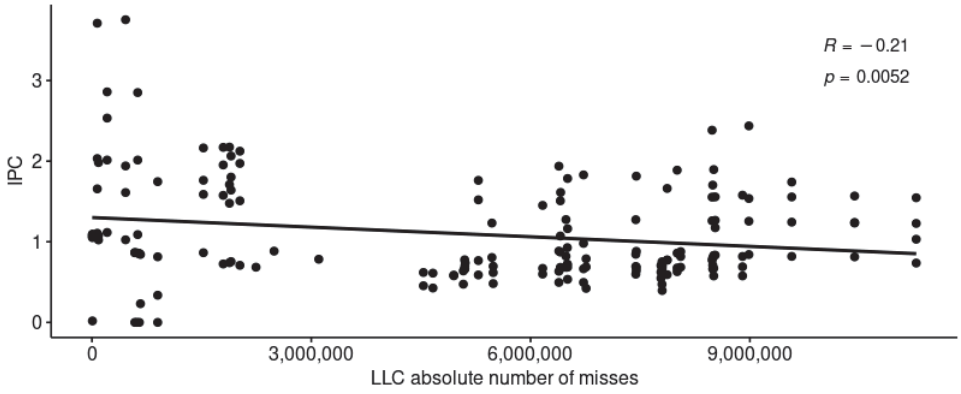


Figure 4.3: Correlation between the absolute number of misses in LLC and the observed IPC in SPEC CPU 2006 applications.

Moreover, Figure 4.4 shows the correlation between the absolute number of misses in the L2 and the IPC, presenting an R-value equal to −0.48.

As described in both figures, the absolute number of hits and misses represents a low R-value. Additionally, using an unbounded number can lead to an overfitting model, incapable of predicting data outside the trained base.

Secondly, we analyze the counters commonly used in literature (48; 14; 31; 15; 24; 28). The metrics of miss and hit rate and HPKI and MPKI. Since miss and hit rate are complementary metrics, we are only discussing one.

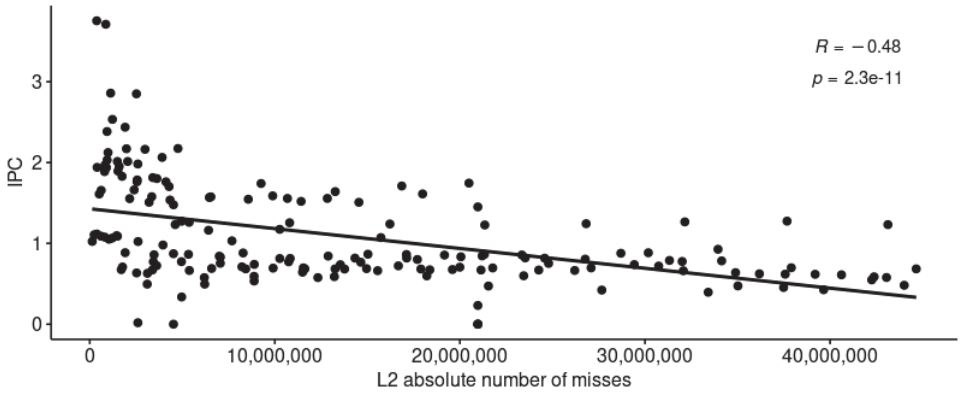


Figure 4.4: Correlation between the absolute number of misses in the L2 and the observed IPC in SPEC CPU 2006 applications.

Figure 4.5 shows the correlation between the miss rate observed in L2 and the IPC, with R equal to −0.60.

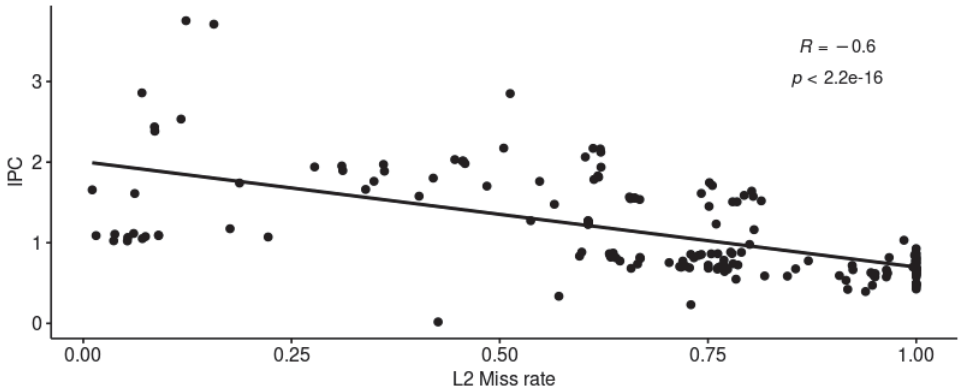


Figure 4.5: Correlation between the miss rate in L2 and the observed IPC for SPEC CPU applications.

As we can see, this metric represents a higher correlation to the IPC than the previous ones (which use absolute values). This fact remains true to the MPKI, illustrated in Figure 4.6. Because these metrics represent a higher correlation value, they are used in the next step to obtain a linear regression model.

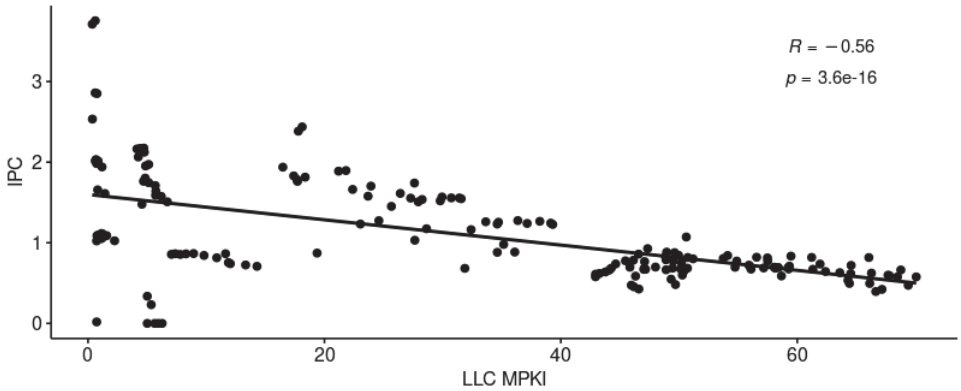


Figure 4.6: Correlation between MPKI in LLC and the observed IPC for SPEC CPU 2006 applications.

Finally, we derive a new metric from the collected ones described in Table 4.1. We construct the accesses per cycle metric using the absolute number of access per cache level. This new metric relies on the fact that the number of cache access could be associated with how well the previous cache layer performs (i.e., the measured access depends on how many accesses are filtered by the previous layer).

To sum up, our analyses reduce the search space of possible features to create the linear regression model. Besides, it validates the metric choice in this dissertation. Additionally, it is essential to note that the presented metrics do not have a strong linear correlation to the IPC value. Thus, no metric presented an R-value higher than 0.60. However, most of the p values are lower than 0.05, indicating a static significance in the correlation.

Due to this, we use not only a combination of the various metrics as a way of tracing a significant relationship with the IPC value but also the Generalized Additive Models (GAM) model because it has the capacity to smooth the terms, and tracing a non-linear relation.

4.0.3 Sampling cache

The sampling cache is a component used to simulate the hardware counters needed to predict the performance by the regression model whenever the LLC is not used (i.e., no actual hardware counter will be available for LLC). This sampling cache also has the benefit of low hardware overhead. In this section, we explain how it works and assess the accuracy of this cache.

The main idea behind the sampling cache is to simulate the behavior of the LLC in a small mechanism with just a few sampled sets. We sample the LLC sets to keep track of LLC usefulness in the execution window where we perform the bypass. The fact found by Qureshi et al. (40) is that a small among of sets can reproduce the entire cache performance. Similarly, this mechanism reproduces all the access and changes in the LLC, storing only the tag information. Thus no data storage is required as we are interested in the hardware counters that can be acquired only with the tag info.

The sets reproduced in the sampling cache (i.e., the sets direct mapped to the sampling cache) are named leader sets. The method to select K leader sets assumes N to be the number of sets in the LLC. Then, we logically divide the LLC into K equally-sized regions, each including N/K sets. The number of ways per leader set is the same as the associativity in the LLC. For each K region, one leader set is selected. This selection may vary depending on the selection policy.

The simple-static policy (40) consists of selecting the set 0 from the first region, 1 from the second region, 2 from the third, and so forth. We select the n^{th} set from each region. Considering only cache with the numbers of sets equals to powers of two, there are no remaining sets in the division process.

This simple policy with a LLC of N sets and a sampling cache of K sets is illustrated in Figure 4.7.

There is a particular case when the number of K regions is bigger than the N/K region size. In that case, when the N/K position is reached in the region, we restart from region position zero. Figure 4.8 shows this case, and we can see a LLC with 8 sets and a sampling cache of 4 sets. In this case, N/K is equal to 2. The mapping starts from the first region, which contains sets 0 and 1, selection of the first set of the region (i.e., set 0 from the LLC). Next, set 1 is selected from the second region (i.e., set 3 of *LLC*). Since we reach the maximum number of sets in each region the next one is the first set, in Figure 4.8 set 4 of the LLC. Worth noting that we pass through all positions in the N/K regions.

When using a sampling cache, we must decide the number of leader sets. Qureshi et al. (40) reports that 16 to 32 sets are sufficiently suitable for their application. Nevertheless, we must evaluate which size is better for DYCA.

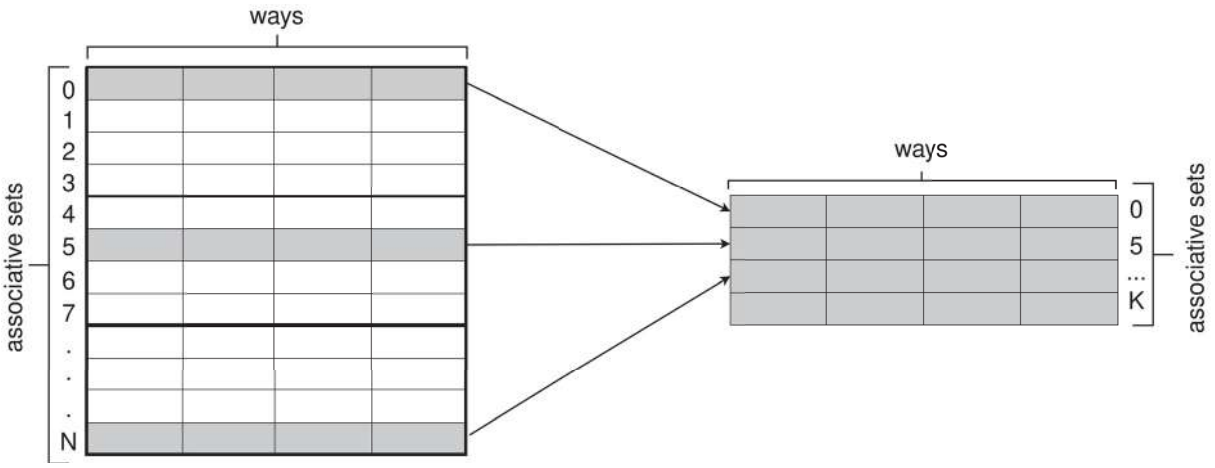


Figure 4.7: Mapping the K leader sets, of each N/K region in a LLC of size N, into a sampling cache of size K, each K leader set mapped to a set in the sampling cache.

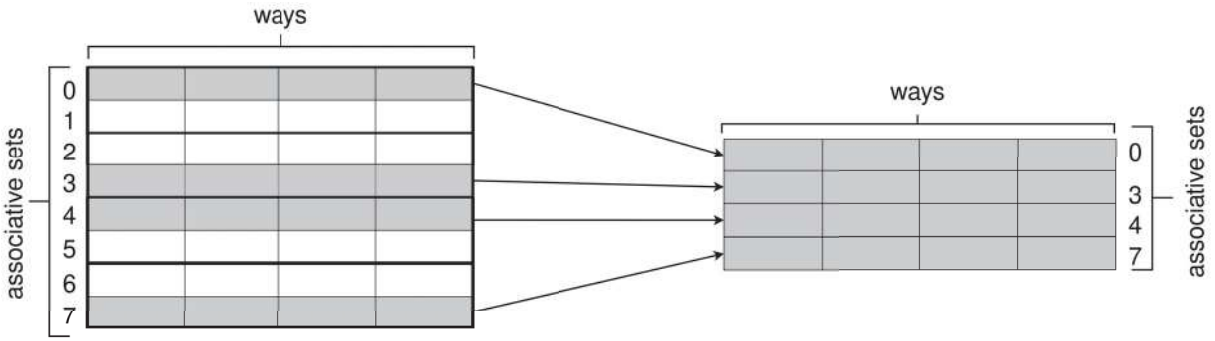


Figure 4.8: Mapping of leader sets when there are more K sets than N/K regions.

For our proposal, we tested the accuracy of the sampling cache over a few sizes to choose the best size (i.e., the one with the higher accuracy and low hardware overhead). Based on Qureshi et al. (40) results, we tested 16 and 32 sets. However, we also tested 64, 128, and 256 sets. The benchmark used for testing was SPEC-CPU 2017(45). The accuracy is measured by the percentage difference between the miss rate and hit rate observed in the real LLC and the one provided by the sampling cache. Figure 4.9 show the average accuracy of all application in SPEC-CPU 2017(45) for all the leader sets sizes experimented (i.e., 16, 32, 64, 128, and 256).

As shown in Figure 4.9, the mechanism's accuracy increases while the size of the sampling cache increases. Although it is possible to notice that the difference between 64, 128, and 256 sets is negligible (i.e., the standard deviation for these three sizes remains similar). On the other hand, the difference observed in sizes 16, 32, and 64 is very significant, especially the standard deviation. The meaning of a higher deviation is associated with a wrong prediction for some of the applications in SPEC-CPU 2017(45). For those reasons listed above, we choose 64 as the number of sets to be used in the following experiments, especially considering the hardware overhead that a larger sampling cache can cause in DYCA.

A change in hardware counters provided by the sampling cache is also required. The reason is that the counter dependent on absolute counters, such as MPKI and HPKI, that use the total number of instructions, would not have the same value as the counter observed at the real LLC due to the difference in size. Therefore, we assume that the sampling cache represents only a K percent of the LLC behavior. In this case, we can multiply the observed value by the

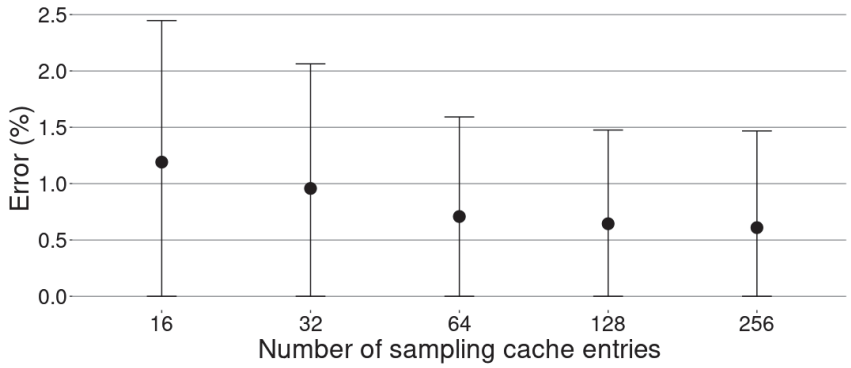


Figure 4.9: Average accuracy and standard deviation for different sampling caches sizes (16, 32, 64, 128, and 256 sets) in the SPEC CPU 2006, calculated using the difference in the hardware counters observed in the sampling cache and in the LLC.

N/K factor, obtaining a result as if the sampling cache has the same among sets of LLC. This adaptation is unnecessary for ratio metrics, such as the hit and miss ratio, since the result is a tax of the observed values. Even though the among of access, misses, and hits is low, the metric considers the relation between the values.

4.0.4 Architecture configuration

We develop two possible architectures using the sampling cache. The first one, named **SingleSC**, where only one sampling shared cache is accessed by all the cores. The second architecture, named **MultiSC**, has a sampling shared cache and a private sampling cache for each core. Figure 4.10(a) shows **SingleSC** architecture configuration with only a shared sampling cache accessed by all cores. Figure 4.10 (b) the **MultiSC** configuration is shown with four cores, each with its sampling cache and a shared sampling cache accessed by all cores, all the access comes from the L2 level.

The key idea behind these two architectures is to evaluate if an individual application behavior of the LLC better detects its usage. Whereas an exclusive cache for each core can improve accuracy, it adds extra hardware overhead. Both aspects are evaluated and discussed in this dissertation, with the aim of finding the one that best fits the mechanism.

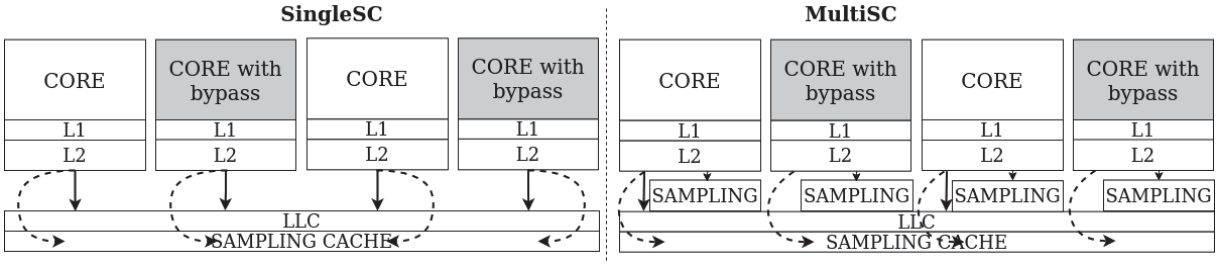


Figure 4.10: Architecture configurations for DYCA in four cores (a) with only one shared sampling cache, and (b) with both one shared and private sampling cache.

Both architecture configurations are tested with the multi-core version. However, only the first approach was tested for the single core evaluation, since the LLC is used only by this one core.

Adding an extra bit is enough to store the behavior decided by the mechanism. Since it is a binary decision, the bit is added to the Translation Look-aside Buffer (TLB).

4.0.5 Linear regression model

To create a model capable of generalization, we used regression models in DYCA to predict the application performance. This section describes the model used and the construction process, including selecting the features and designing the test and training bases.

The applications of SPEC-CPU-2006 (44) were used to create the training bases, and the test is performed using the applications of SPEC-CPU 2017(45). This newer SPEC benchmark suite has new applications and also preserve some of the application from the SPEC-CPU-2006. Thus, we believe this method simulates a hypothetical scenario where the industry would train its processors without our mechanism. Later on, the users would keep using the system with newer applications.

For training bases in the single thread version, 24 applications of SPEC-CPU 2006(44) were executed. The test is performed in the 17 applications of SPEC-CPU 2017(45). The summary of the training and testing bases for the single thread version is described in Table 4.2.

Bases type	**Benchmark**	**Quantity**	**List of applications**
Training	SPEC-CPU 2006	24	astar, cactus, calculix, dealII, games, gems, gobmk, gromacs, h264, hmmer, lbm, leslie, libquantum, milc, namd, omnetpp, perlbench, povray, sjeng, soplex, sphinx3, tonto, xalancbemk and zeusmp
Testing	SPEC-CPU 2017	17	bwaves, cactus, exchange, image, lbm, leela, mcf, nab, omnetpp, perlbench, roms, wrf and xalancbmk

Table 4.2: Summary of training and testing bases parameters.

For DYCA, two models are required, one model predicts the IPC using LLC (**wLLC**), used when bypass is performed. And one model to predict IPC without LLC (**woLLC**), this one used when the LLC is enabled.

Both models aim to create equations capable of predicting the application's IPC in different scenarios. These equations are based on the features described in Table 4.3 and find the maximum global combination of features in this scenario, creating the most accurate equation possible, each feature was combined with IPC value to plot and measure the correlation between them, consequently, remove the ones with no correlation.

All metrics, even when the model predicts the IPC for a bypassing scenario, are obtained from an LLC sampling cache. For this reason, we trained the models with information from the sampling cache.

Since we are using the hardware counters observed in time t to predict the future IPC in time $t + 1$, the training base needs to reflect this behavior. Therefore, we shift the IPCs from one execution window ahead to match the previous hardware counters observed, as illustrated in Figure 4.11.

Using metrics from time t (i.e., the finished execution window) to predict the behavior of the following $t + 1$ window is based on the knowledge that programs are well-behaved without sudden behavior changes. Although a program phase can extend for millions of cycles, there are changes in the behavior that our mechanism shall detect. Furthermore, using these models prevents the use of a specific decision threshold, preventing the overfitting and making the model suitable for diverse programs.

Each model was developed using the R language and the GAM package. The features used to train the model are described in Table 4.3.

When creating the models, our approach was to create a general equation with all variables, measure each variable's correlation, and select the ones more relevant to explain the

```
application, core, IPC_LLC, IPC_SLLC, miss_llc, hit_llc, miss_sampling_cache, hit_sampling_cache
BWAVES, 0, 3.0167530, 3.2490450, 0.008434, 0.991566, 0.000573, 0.999427
BWAVES, 0, 3.4028880, 3.4396540, 0.004816, 0.995184, 0.001621, 0.998379
CACTUS, 0, 0.8727450, 0.9431940, 0.170157, 0.829843, 0.003642,0.996358
CACTUS, 0, 0.8735980, 0.9434470, 0.170158, 0.829842, 0.996204,0.003796
```

Figure 4.11: Example of the resulting file used for the training phase.

Component	Features	Execution window	Full execution
L1	Miss rate	•	•
	Hit rate	•	•
	Access rate	•	•
	MPKI	•	•
	HPKI	•	•
L2	Miss rate	•	•
	Hit rate	•	•
	Access rate	•	•
	MPKI	•	•
	HPKI	•	•
LLC	Miss rate	•	•
	Hit rate	•	•
	Access rate	•	•
	MPKI	•	•
	HPKI	•	•
LLC sampling cache	Miss rate	•	•
	Hit rate	•	•
	Access rate	•	•
	MPKI	•	•
	HPKI	•	•
Processing core	IPC using LLC		•
	IPC without LLC		•
	Number of instructions		•
Memory system	Number of loads	•	
	Number of stores	•	

Table 4.3: Summary of features used in the linear regression model.

IPC. We evaluate the R-sq or R^2 as a metric of the model accuracy and the Mean Squared Error (MSE).

Then, after making a new equation with the variables, we measure the correlations, select the more relevant ones so far, and so on. This process stops when the MSE of the model increases or the R^2 decreases. The process of building the model is described in Figure 4.12.

Therefore, we describe the two models (**wLLC** and **woLLC**) separately in the following subsections.

4.0.6 wLLC model

After analyzing all the steps described in Figure 4.12, the resulting equation to **wLLC** model is:

$$IPC = s_0 L1\ MPKI + s_1 L1\ hit\ ratio + s_2 L2\ MPKI + s_4 LLC\ HPKI + s_5 LLC\ accesses + s_6 Loads \tag{4.1}$$

As said before, the hardware counters used as features in this model are obtained from the sampling cache, this way the LLC features used come from the LLC sampling cache.

Figure 4.13 shows the smooth functions observed for the metrics described in equation 4.1.

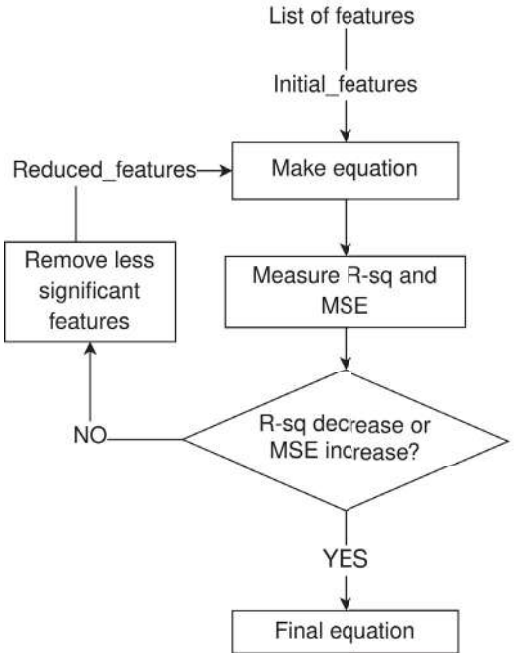


Figure 4.12: Process to select the features and build the linear regression model.

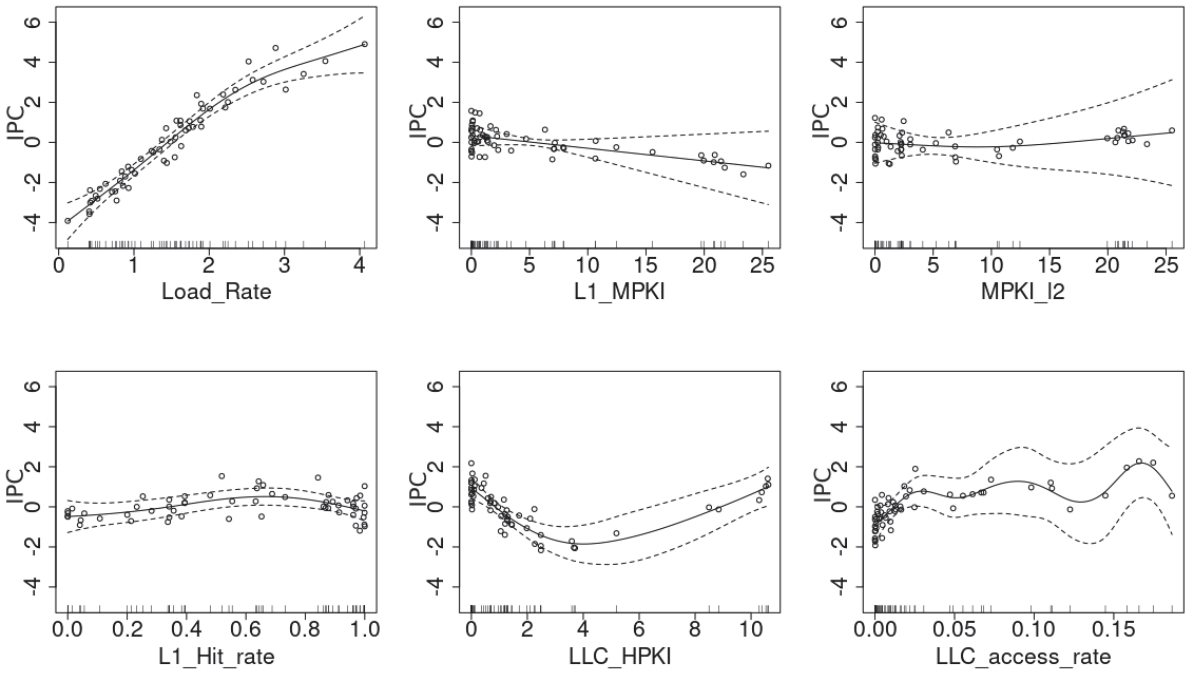


Figure 4.13: Correlation of hardware counters for **wLLC** model.

Both positive and negative correlations are seen in Figure 4.13. Even though, all features show a high significance in the equation, *Load_Rate* and *L1_Hit_Rate* presented a higher correlation to the IPC values, the observed correlations reflect in the accuracy of the model. For SPEC-CPU 2006 (i.e. testing base) this equation presents a R^2 of 0.93 and 0.50 for MSE.

4.0.7 woLLC model

We evaluate the features of each cache layer separated after merging all features to get the final equation for this model. The model is the same for single and multi-core approaches. The resulting equation is:

$$IPC = s_0 L1\ MPKI + s_1 L1\ acesses + s_2 L2\ HPKI + s_3 L2\ Hitrate + s_4 Loads \quad (4.2)$$

Similar to the previous model, for this one the hardware counters also come from the sampling cache. Figure 4.14 shows the smooth functions observed for the metrics described in equation 4.2.

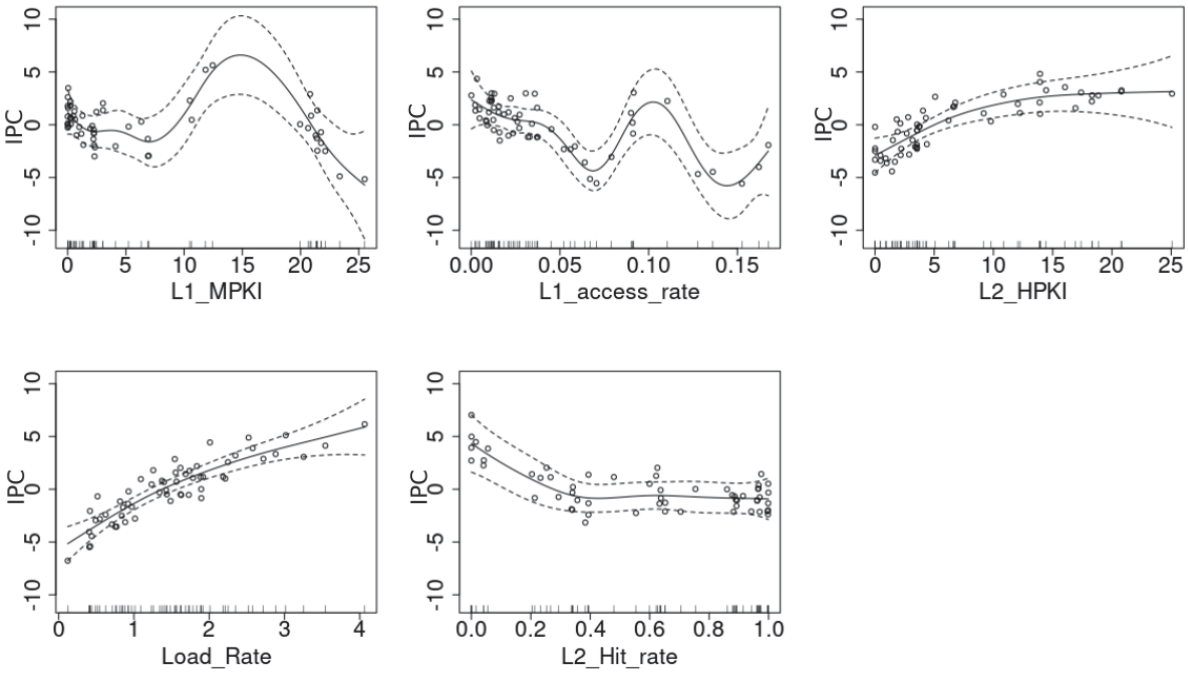


Figure 4.14: Correlation of hardware counters for **woLLC** model.

In the **woLLC** model we see a positive correlation in the *Load_Rate* feature, and a negative one in *L2_Hit_rate*, in Figure 4.14. Also, is possible to see that *L1_MPKI* and *L1_access_rate* show a harder-to-predict behavior than the other features, this is reflected in the R^2 and MSE of the model, which is lower than model **wLLC**.

Also for SPEC-CPU 2006 (i.e. testing base), this equation presents a R^2 of 0.70 and 1.19 of MSE.

4.0.8 Performance and hardware overhead

Some overhead performance is added when a behavior change is detected, from using LLC → bypassing or bypassing → using LLC. First, it is necessary to empty the pipeline before fetching another instruction as we change the architecture configuration—secondly, some changes in the cache hierarchy for correct maintenance of the coherence protocol are needed. We adopted the worst-case scenario to simulate this overhead by invalidating all cache levels.

Analyzing the hardware overhead caused by adding a sampling cache, most of all, the total hardware overhead of the approach is not significant, representing less than 0.2% of the area

of a 1MB cache(40). The vital aspect is that a sampling cache stores only the tag address and the replacement policy information. More than that, since our mechanism is not in the critical path, the latency provided by it is equal to zero, not increasing the total latency of the cache accesses.

In DYCA, the hardware overhead presented by adding a sampling cache is equal to 55 kB, considering the choice to use 64 sets and the architecture configuration **SingleSC**, which only adds a shared sampling cache. For architecture configuration **MultiSC**, the hardware overhead will depend on the number of processing cores present. For instance, using 4 cores, the overhead will be 275 Kb (i.e., 55 kB from the shared sampling and $4x55$ for the sampling caches added for each core). Table 4.4 brings overhead evaluation for several situations.

Number of sets	Architecture configuration	Number of cores	Total hardware overhead
16	SingleSC	1	13kB
16	MultiSC	4	65kB
16	MultiSC	8	117kB
32	SingleSC	1	27kB
32	MultiSC	4	135kB
32	MultiSC	8	243kB
64	SingleSC	1	55kB
64	MultiSC	4	275kB
64	MultiSC	8	495kB

Table 4.4: Hardware overhead for different sampling cache sizes.

4.0.9 General flow of DYCA

To fully understand how DYCA works, Figure 4.15 illustrates the behavior of the mechanism during program execution in an architecture simulator.

The hardware counters are obtained every 200 million instructions execution (Figure 4.15 **A**). When this window size is reached, the two models are executed using the features (i.e., hardware counters) from the sampling cache (Figure 4.15 **B**).

The first model **wLLC**, Figure 4.15 **C**, predicts the application performance using LLC. The second model **woLLC**, Figure 4.15 **D**, predicts the application performance bypassing the LLC. Note that the hardware counters came from the sampling cache and for that reason, if an error is observed in the sampling cache, the error could affect the correctness of the model.

After predicting both performances, the next step is to make a decision. This decision depends on the values predicted and the actual LLC configuration, Figure 4.15 **E**.

If the previous window bypasses the LLC and the prediction of model **wLLC** is higher than the prediction of **woLLC**, then LLC is adapted for use the LLC, Figure 4.15 **F**.

On the other hand, if the previous window is using the LLC and the predicted IPC of model **woLLC** is higher than the predicted IPC in model **wLLC** the adaptation is done to bypass the LLC, Figure 4.15 **G**.

Some care must be taken to guarantee coherence in the cache layers when a change is done. In this case, no more instructions are fetched, Figure 4.15 **H**, and the mechanism waits until all instructions are committed, Figure 4.15 **I**. After an empty pipeline is achieved, all LLC data are written back, and the cache lines are invalidated, Figure 4.15 **J**. After these steps, the change can be done, Figure 4.15 **K**, adapting the LLC. In the last step, fetch is restarted and all the processes starts again, Figure 4.15 **L**, until the end of application execution.

DYCA is executed in a simulator, for that reason we did not consider the OS context switch in this control flow.

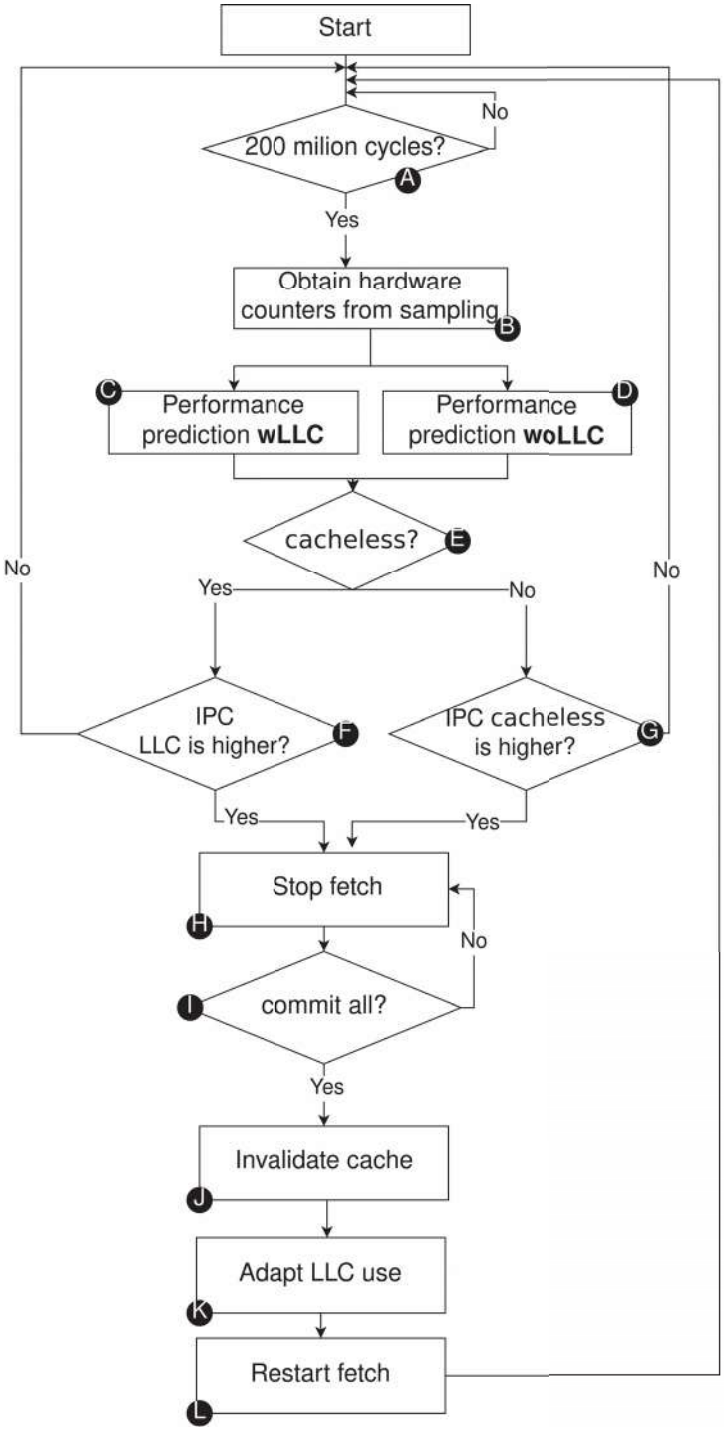


Figure 4.15: Control-flow graph of our proposal mechanism, illustrating the process, controls, and decisions made for each execution window.

5 EXPERIMENTAL EVALUATION AND RESULTS

This chapter describes the method used for the experiments and the results for the simulation of Multi-core Dynamically Adaptable Cache Bypassing Mechanism (DYCA) in single and multi-program approaches.

Section 5.0.1 describes the setup used in the experiments. While Section 5.0.2, 5.0.3, 5.0.4, and 5.0.5 discuss the experiments and results acquired in this dissertation.

In Section 5.0.2 we start to evaluate the existence of applications that not take advantage of using the Last-Level Cache (LLC) (i.e., do not improve performance using it). After, Section 5.0.3 describes an oracle mechanism, capable of executing the application in the best possible LLC usage configuration. Next, Section 5.0.4 presents and discuss the result obtained for DYCA in a single program workload. To finish, Section 5.0.5 describes the results for a multi-program workload.

5.0.1 Simulation setup

To develop this proposal, we use an in-house simulator called Ordinary Computer Simulator (OrCS) which is an evolution of SiNUCA (4). DYCA was implemented inside OrCS with all the additional hardware: the sampling caches, the equation table, and the decisions and bypass mechanisms. Table 5.1 present the simulation parameters.

Processor Cores: 8 cores @ 2.0 GHz, 32 nm; 4-wide out-of-order; 16 stages 16 B fetch size; 18-entry fetch buffer, 28-entry decode buffer; 168-entry ROB; MOB entries: 64-read, 36-write; 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle); 1-load and 1-store units (1-1 cycle); Branch Predictor: 1 branch per fetch; 4 K-entry 4-way set-assoc., LRU policy BTB; 48-entry BOB; Two-Level GAs 2-bits; 16 K-entry PBHT; 256 lines, 2048 sets SPHT;
L1 Data + Inst. Cache: 32 KB, 8-way LRU, 64 B line size; 2-cycle;MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides;
L2 Cache: Private 256 KB, 8-way LRU, 64 B line size; 10-cycle;MSHR: 4-request, 6-write-back, 2-prefetch; Stream prefetch: 2-degree, 256-streams;
L3 Cache: Shared 16 MB (8-banks), 20-way LRU; 64 B line size; 22-cycle; MOESI coherence protocol; MSHR: 8-request, 12-write-back;
LLC Interconnection: Bi-directional ring;
Private Sampling Cache: 4x 64 KB, 64-sets, 16-way LRU; 64 B line size; 2-cycle; Non-coherent;
Shared Sampling Cache: 1x 64 KB (8-banks), 64-sets, 16-way LRU; 64 B line size; 2-cycle; Non-coherent;

Table 5.1: Simulation parameters

5.0.2 First validation

For this first experiment, we investigate if there are applications that gain performance when executed without the LLC. Therefore we only execute the applications on a system with or without the LLC, without any particular mechanism.

Figure 5.1 shows the relative performance for 24 applications in SPEC-CPU 2006 (44) when executing without the LLC compared to a baseline with 16 MB of LLC. It is possible to see three groups of results. The first is composed of applications that gain performance when the LLC is not present (cactus, lbm, leslie, zeusmp, soplex, milc, gems, and libquantum). The next group contains some applications that do not change performance using or not the LLC (games, namd, povray, sjeng). In the last group, we see applications that present performance degradation caused by the absence of LLC (sphinx3, hmmer, astar, omnetpp, gromacs, h264, calculix, perlbench, xalancbemk, dealII, gobmk and tonto).

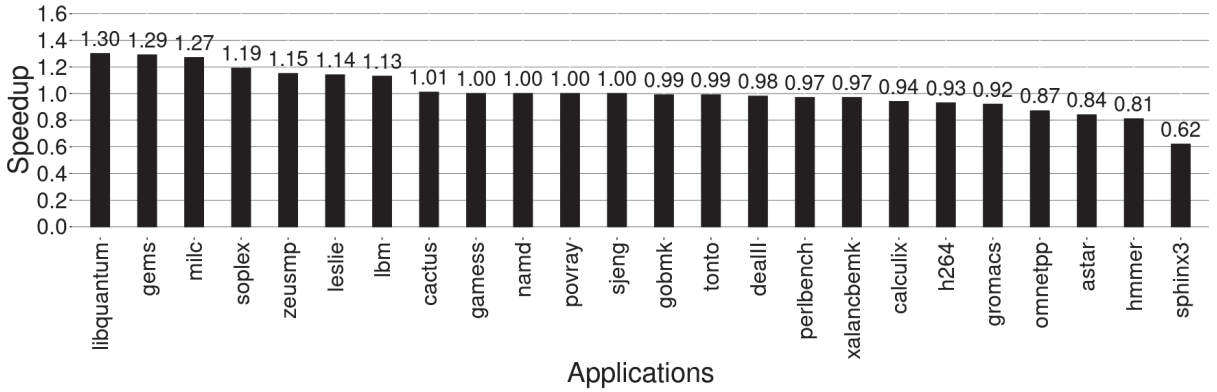


Figure 5.1: Possible performance gains executing SPEC-CPU 2006.

We can observe in Figure 5.2 that we also have these three groups of applications for SPEC-CPU 2017 (45) benchmark suite.

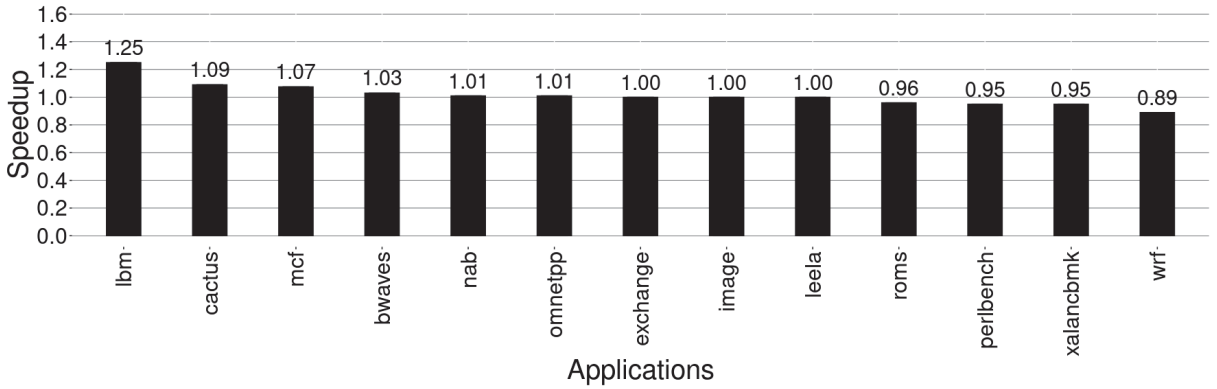


Figure 5.2: Possible performance gains executing SPEC-CPU 2017.

These experiments show us possible gains when removing the LLC, reaching up to 30% for some applications. On the other hand, the lack of LLC can hurt performance up to 48% for some applications. Thus we must emphasize the importance of a precise decision mechanism that can bypass the LLC in precise situations.

5.0.3 Oracle performance

After knowing that gains are possible, in this section, we present an oracle mechanism to model the maximum gains for DYCA.

Our oracle mechanism considers two executions of each application, the first with a 16MB LLC and the second without the LLC, exemplified in the first two lines of Figure 5.3. For each execution window (in our case, 200M of instructions), the oracle chooses the system

configuration with the highest Instructions per Cycle (IPC), as shown in the last line of Figure 5.3. Figure 5.3 show an execution slice of *lbm* application from SPEC-CPU 2017.

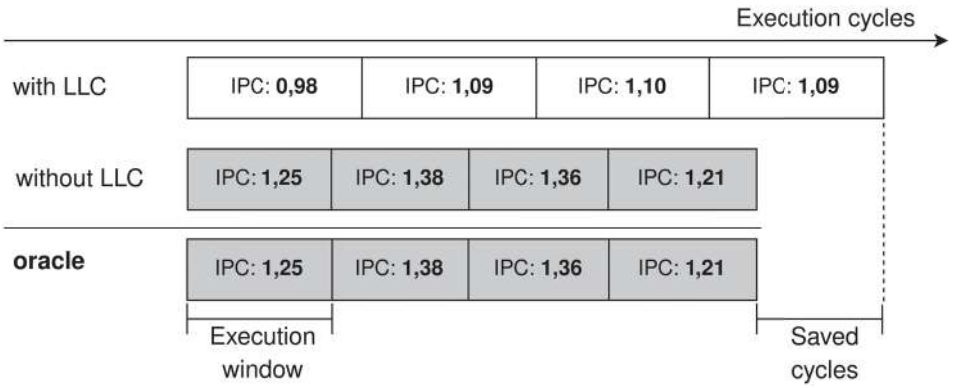


Figure 5.3: Example of an oracle execution for application *lbm* in SPEC CPU 2006.

This experiment aims to motivate this work about the possible gains in a curated mechanism to adapt the LLC access in a core granularity, considering the changes from one execution window to another. As shown in Figure 5.4 the oracle execution is capable of saving more cycles whereas it executes the application in the best possible configuration.

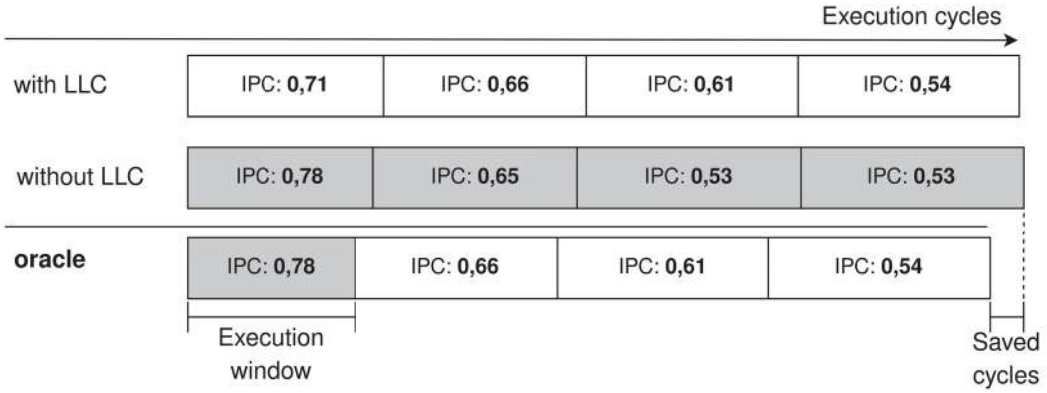


Figure 5.4: Example of an oracle execution for mixed configurations during *gcc* execution.

However, for this oracle, no overhead is considered, meaning that changing between a system with or without the LLC has no area or time overhead. Nevertheless, we acknowledge that real systems would suffer from cache coherence protocol issues such as a vast amount of line invalidations.

Figure 5.5 shows the gains observed for SPEC-CPU 2006(44). The first eight applications (libquantum, gems, soplex, leslie, zeusmp, lbm, and cactus) gain performance when switching between systems (with and without LLC) during execution. The remaining applications do not benefit from the LLC adaptation, thus, presenting no performance change (i.e., speedup equal to one).

Figure5.6 shows performance results for SPEC-CPU 2017 (45) using the oracle mechanism.

When we analyze the applications with performance improvements, the average gain is 24% for SPEC-CPU 2006 and 15% for SPEC-CPU 2017.

Although, some applications from SPEC-CPU 2006 present a similar result with the static or the oracle approach (e.g., *libquantum*, *gems*, *milc*, *soplex*, *leslie*, *zeusmp*,*lbm* and

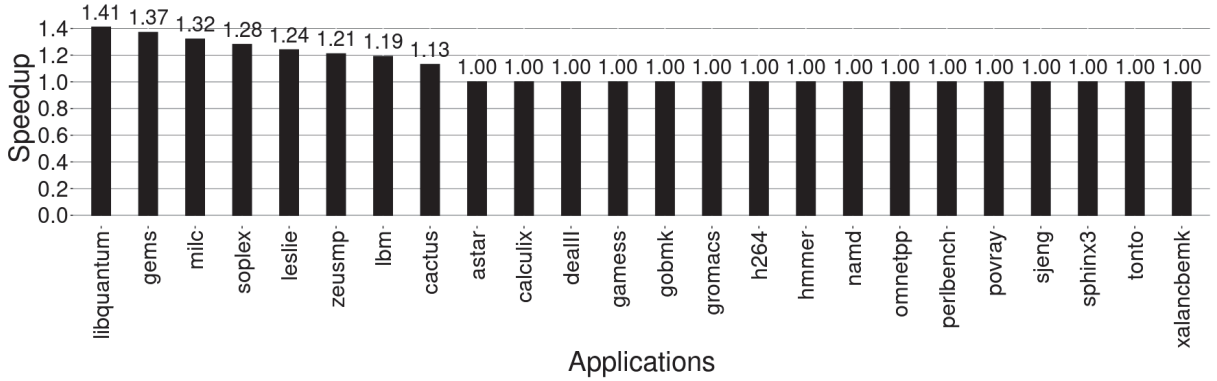


Figure 5.5: Oracle results for SPEC-CPU 2006.

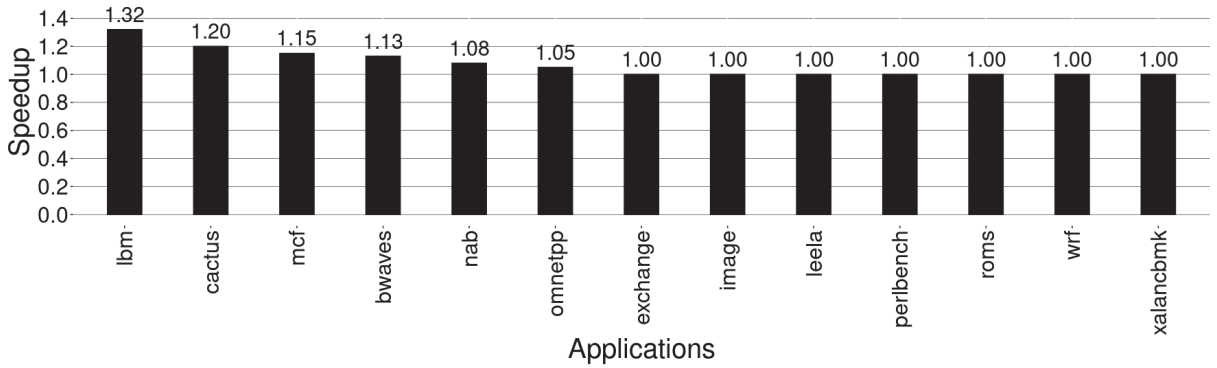


Figure 5.6: Oracle results for SPEC-CPU 2017.

cactus). Compared to the static execution (i.e., Figure 5.1 and 5.2) we can notice that our oracle mechanism could obtain average gains of 8% (against 0, 5%) for SPEC-CPU 2006 and 6% (against 1%) for SPEC-CPU 2017. Such gains are possible due to 2 factors, the dynamic behavior and the avoidance of performance loss achieved by the oracle.

hmmer and *sphinx3* in Figure 5.1. Nevertheless, the oracle mechanism retains the performance of this application, in this case showing a speedup equal to one as shown in Figure5.6.

Figure 5.2 shows similar behavior for SPEC-CPU 2017, where oracle reached a better performance than the static mechanism.

Thus, it is possible to conclude the importance of using a mechanism capable of identifying the application behavior and dynamically adapting the use of the LLC since this can lead to a higher performance improvement while preventing a performance loss in some applications. Still, achieving performance gains similar to the ones observed with the oracle is challenging due to performance overheads that realistic mechanisms may present. Moreover, such a mechanism must predict which system would present the best performance during runtime, possibly requiring training phases and data storage.

5.0.4 Single application

Here, we evaluate DYCA in the single application approach. In this case, we used single applications from SPEC-CPU 2006(44) to train the model. Later we used applications from SPEC-CPU 2017(45) to test the mechanism.

Figure5.7 presents the results for SPEC-CPU 2017 using our mechanism. We can observe a maximum performance degradation of 1% for one application (*xalancbmk*). Considering the oracle results (i.e., Figure 5.6), we can observe that this application had no performance improvement. Thus, we can infer that such degradation is due to our model's miss-prediction, causing a wrong adaptation of using LLC.

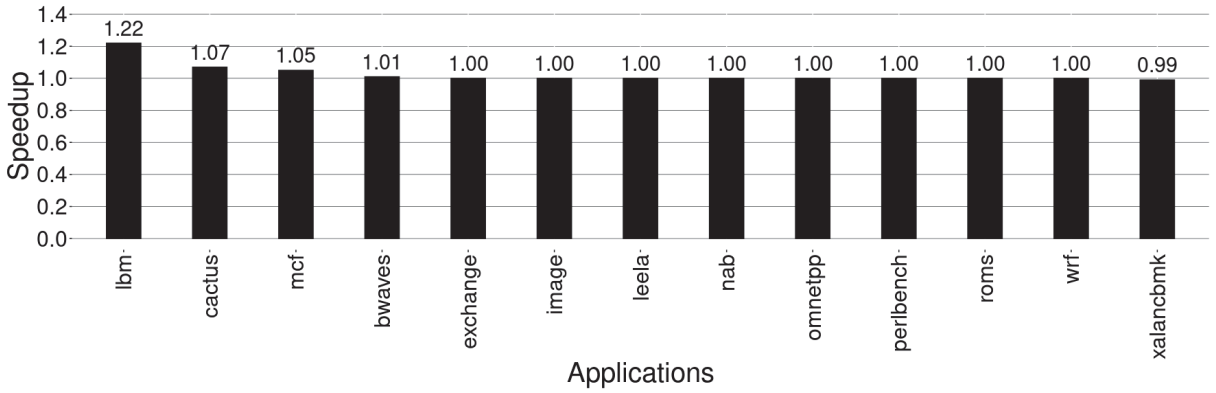


Figure 5.7: DYCA results in a single program execution of SPEC-CPU 2017.

In addition, some applications (e.g., *lbm*, *cactus*, *mcf* and *bwaves*) shows improvements in performance up to 22%. Such applications are the same that present speedup when disabling the LLC. Meanwhile, the other applications (*exchange*, *gcc*, *image*, *leela*, *nab*, *omnetpp*, *perlbench*, *roms* and *wrf*) maintain the same performance.

In order to understand the gains obtained by our mechanism, it is essential to observe two metrics during the execution of these applications, the LLC misses and the Misses per Kilo Instructions (MPKI). For instance, we expect applications with a high LLC miss rate to gain performance when we bypass the LLC.

Figure 5.9 shows the LLC miss rate for the applications of SPEC-CPU 2017 (we choose to keep the application order the same from the performance results).

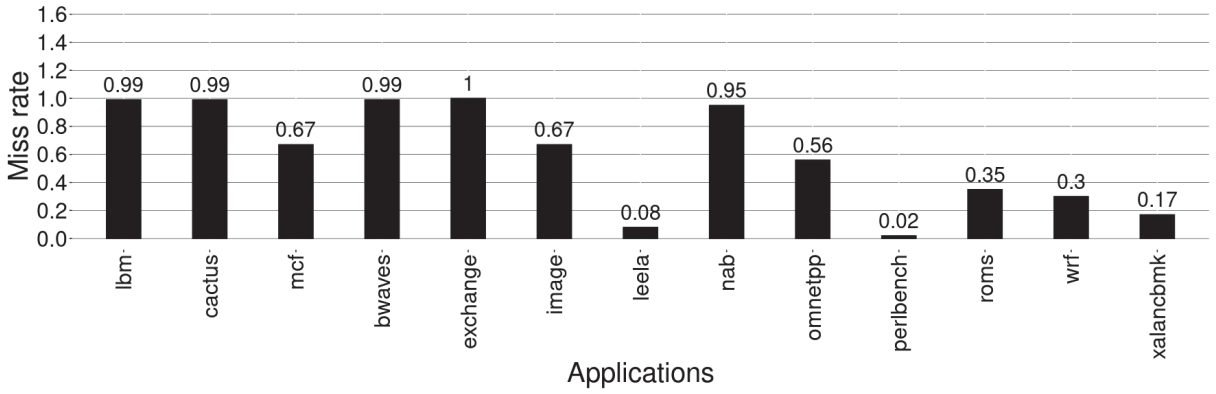


Figure 5.8: LLC miss rate for SPEC-CPU 2017 applications.

From the four applications that gain performance (i.e., *lbm*, *cactus*, *mcf* and *bwaves*) three presented a miss rate as high as 0.99%. However, this metric does not explain the results alone, as we can observe for *mcf* that gains performance while having a 0.67% of LLC cache miss, which is lower than *nab* (0.95%) that shows no performance difference. The same observation is applicable for application *exchange*.

In order to better understand our mechanism's effect, we must also analyze the pressure on LLC combined with the miss ratio. Using the MPKI, we can observe how often the LLC is

required and fail to provide data hits. Therefore, in a high-pressure scenario (i.e., high MPKI), the LLC becomes less attractive in terms of performance. Figure 4.6 shows the MPKI for the same applications of SPEC-CPU 2017.

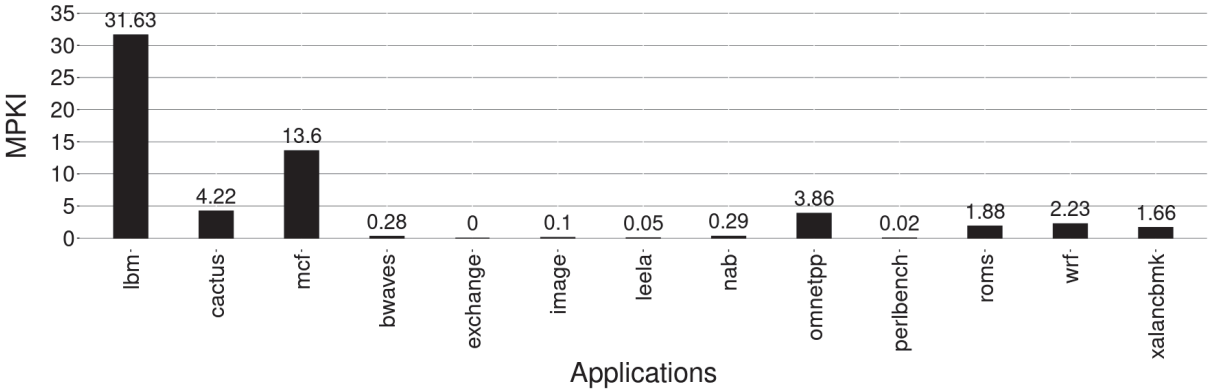


Figure 5.9: LLC MPKI for SPEC-CPU 2017 applications.

Another important aspect to be considered when changing from one state to another (i.e., from using LLC to bypass or the other way around), is that for most of the applications, the number of changes is not superior of 5. For instance, application *mcf* has 5 changes, while *lbm* and *cactus* have only one. This low number of changes contributes to a low overhead since fewer cache invalidation are required during the runtime.

The discussed applications (*exchange* and *nab*) show a lower MPKI than applications where a LLC adaptation have a more representative impact, such as *lbm* application. In this case, the *lbm* has the highest speedup, and the higher MPKI observed. On the other hand, we observe again that a single metric does not fully explain the results.

We can conclude that DYCA correctly identified the applications that should bypass or not the LLC, achieving thus consistent performance improvements and negligible performance degradation of 1% at most.

5.0.5 Multiple applications

To evaluate a multi-program scenario, we created bundles of four applications each. The first one is the **Bypass-compatible apps** bundle, created with only applications that gain performance when LLC is disabled (i.e., when bypass is used). The Bypass-compatible apps (a) consist of *lbm*, *cactus*, *mcf* and *bwaves* applications and Bypass-compatible apps (b) contain the applications *mcf*, *bwaves*, *nab* and *omnetpp*. The second bundle is the **Bypass-incompatible apps**, including only applications that lose performance when bypass is applied. In this case, containing *wrf*, *roms*, *perlbench* and *xalancbmk* applications.

Besides, we created the **Mixed apps** bundle with half applications that gain performance and half that loses it. The first bundle (i.e. Mixed apps (a)) represents the execution of *lbm*, *cactus*, *xalancbmk* and *wrf*. The second (i.e. Mixed apps (b)) considered applications *mcf*, *bwaves*. *roms* and *perlbench*.

Finally, a **Random apps** bundle with randomly chosen applications, in our experiment the chosen ones are *lbm*, *bwaves*, *image* and *perlbench*.

Figure 5.10 describes the result of DYCA using the different application bundles identified at the bottom of each column.

We can observe that DYCA gains performance in almost every configuration. Conversely, for the Random apps bundle, 1% of performance is lost. Such performance degradation is

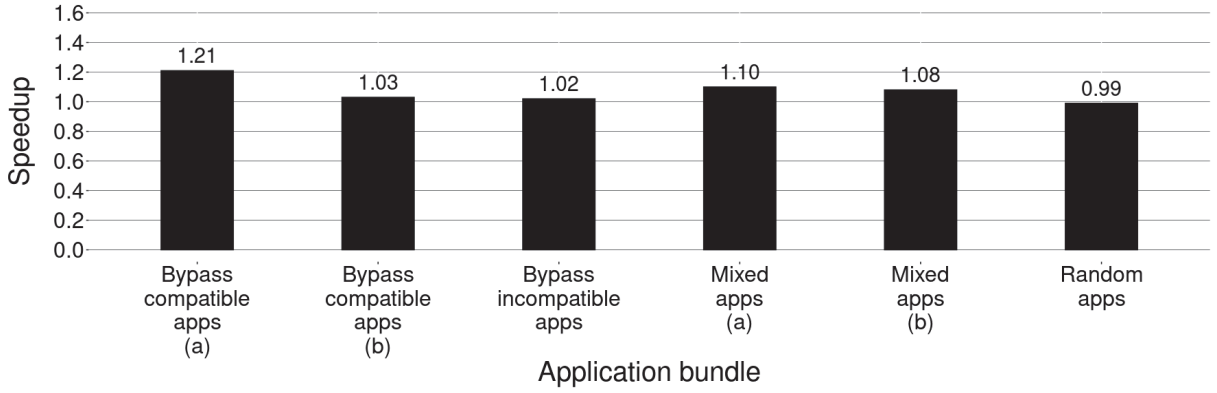


Figure 5.10: DYCA results for SPEC-CPU 2017 with bundles of four applications.

associated with a miss-prediction of the model. Because the model wrongly identifies a behavior to disable the use of the LLC hurting the performance.

In order to provide us with more comparison basis, figure 5.11 presents the results for a static approach for the same application bundle. It shows a loss of 26% in the Random bundle.

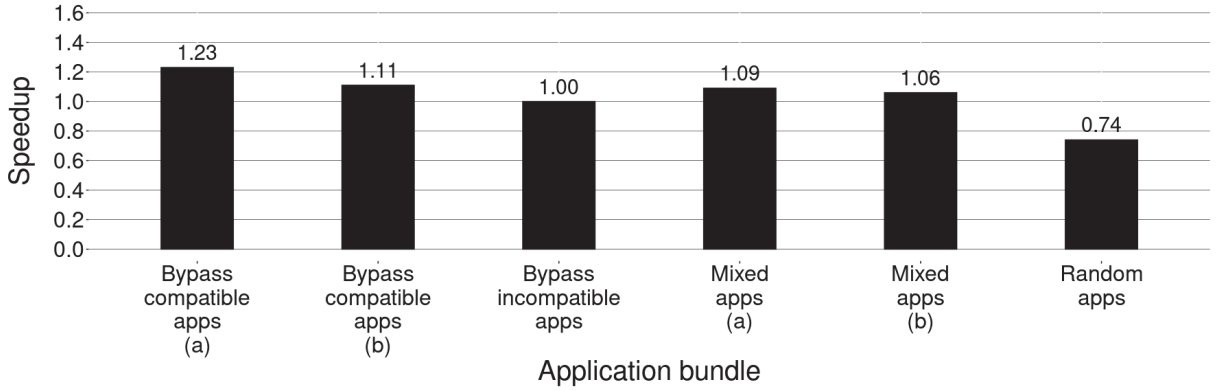


Figure 5.11: Static performance results for SPEC-CPU 2017 with bundles of four applications comparing a 0MB and a 16MB LLC.

For the Bypass-compatible bundles (*a* and *b*) improvements of 21% and 3% are observed respectively. The improvement observed is 2% and 1% lower than the static mechanism. This bundle contains only applications that gain performance when bypassing the LLC. Thus, it is expected that DYCA presents a lower performance than a static or oracle version, as our mechanism cannot make predictions for the first execution window.

Also, the improvement observed in the bundle is associated with each application improvement and behavior individually. On one hand, LLC-compatible (a) contains application *lbm*, which presented a performance improvement of 32% in oracle execution (see Figure 5.6). Also, contain application *cactus*, *mcf* and *bwaves* with a improvement of 20%, 15% and 13% respectively (see Figure 5.7). For this bundle, a speedup of 1.21 is observed.

On the other hand, LLC-compatible (b), which presented a lower speedup of 1.03, also contains an application with a lower individual speedup. Applications *mcf*, *bwaves*, *nab* and *omnetpp* show a speedup of 1.15, 1.13, 1.08 and 1.05 respectively. Such low gains happen because of our mechanism overhead. Many cache invalidations are required on every change in our mechanism's choice, hurting the performance.

Regarding the results, for both Mixed-apps bundles (a) and (b), we observe gains in performance that could be associated with a reduction in cache pollution and conflicts when our

mechanism was used. Because of this cache-friendly environment, the Mixed apps (b) achieved a speedup 3% higher than the highest speedup observed in each application of the bundle executed individually using DYCA (i.e., *mcf* with 1.05 speedup).

Besides, the bypass-incompatible bundle shows a speedup of 1.02 even containing only applications that, on average, lose performance when bypassing the LLC. This gain is explained by the fact that applications have different execution phases, and our mechanism could adapt to them dynamically. Bypassing the execution windows where some applications did not present a high IPC using the LLC could improve average system performance by being favorable to the other applications since there is a reduction in the LLC pressure.

Table B.1 contains the IPC for each execution window of *roms* application. It is possible to see two LLC adaptations to disable the use of this cache layer (800M and 1.4B of cycles) [1].

Execution window	IPC wLLC	IPC woLLC	LLC configuration	IPC DYCA
400M	1.30	1.21	wLLC	1.30
600M	1.30	1.17	wLLC	1.30
800M	1.07	1.15	**woLLC**	1.05
1B	0.99	0.88	wLLC	0.97
1.2B	1.34	1.30	wLLC	1.33
1.4B	0.96	1.06	**woLLC**	0.96
1.6B	0.81	0.76	wLLC	0.81

Table 5.2: IPC for each execution window in *roms* application.

Also, as discussed, changing between configurations may add some extra latency. For instance, in execution window 800M from table B.1, the IPC observed is lower than the execution bypassing the LLC (i.e., 1.05 against the possible 1.15). The same occurs in execution window 1.4B, where the IPC presented by DYCA execution is 0.10 lower than an execution without the LLC.

When our mechanism performs a change, the next execution window faces the cold cache effect, reflected in performance. This effect could extend for more than one execution window, as seen in execution windows 1B and 1.2B in Table B.1. Both windows present low IPC, showing that a change that occurred on the 1 B window still affected the 1.2 B.

Nevertheless, for such a scenario, since it is a multiple program execution, the other applications can befit from using the LLC with less competition.

Execution window	IPC wLLC	IPC woLLC	LLC configuration	IPC DYCA
400M	1.43	1.25	wLLC	1.43
600M	1.45	1.23	wLLC	1.45
800M	1.37	1.07	wLLC	1.37
1B	0.85	1.17	**woLLC**	0.77
1.2B	1.10	0.76	wLLC	0.89
1.4B	1.46	1.05	wLLC	1.46

Table 5.3: IPC for each execution window in *wrf* application.

Furthermore, DYCA also decreases the average system miss rate. Since there is low cache pollution, the applications can use the cache more profitably. For instance, the cache miss observed in a base execution with a 16MB LLC and a DYCA execution are described in Figure 5.12.

[1]The values correspond to a real scenario, not the one predicted by the model, the aim is to evaluate the possible gains. However, the values predicted by DYCA presented the same configuration as seen in Appendix B.

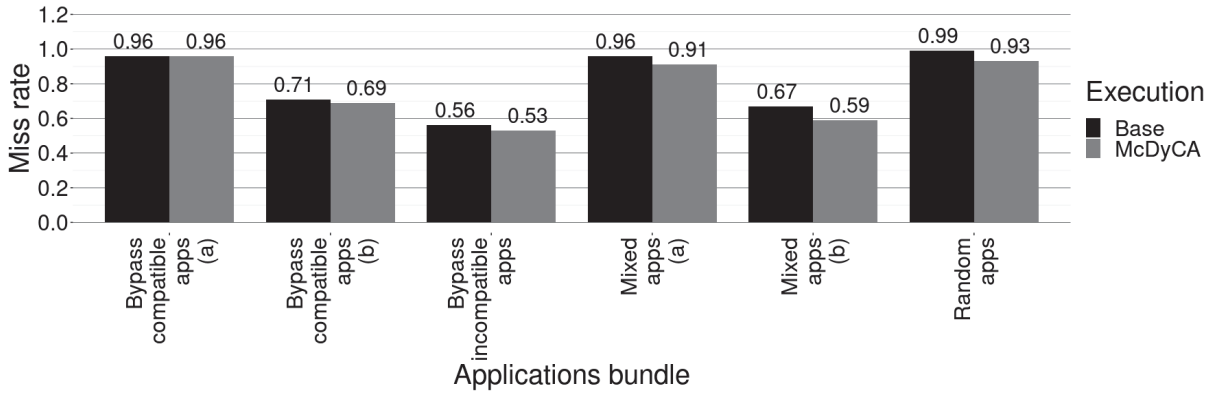


Figure 5.12: Average miss rate for SPEC-CPU 2017 with bundles of four applications when using 16MB of LLC (**base**) and when executing DYCA (**DYCA**) .

Both bypass-compatible bundles (a) and (b) show a similar miss rate. However, for bundles where the use of the LLC does not improve performance, the number of accesses is decreased because most of the execution windows are bypassed. For instance, base execution of bypass-compatible-apps bundles (b) shows a MPKI of 19.63 while DYCA execution presented a MPKI of 3.03.

To sum up, DYCA can improve average system performance at the same time as low the miss rate by dynamically adapting the use of the LLC for each application during run time, enabling an efficient configuration for LLC.

6 CONCLUSION

In this work, we proposed a new model to dynamically bypass the Last Level Cache (LLC) based on the performance prediction emulating different cache hierarchy conditions, with L1-L2-L3 or L1-L2 only. Our mechanism presents a low hardware overhead, relying only on a small sampling cache and simple decision logic.

Multi-core Dynamically Adaptable Cache Bypassing Mechanism (DYCA) was simulated in OrCs, using Generalized Additive Models (GAM) linear regression model and a sampling cache to simulate the Last-Level Cache (LLC) hardware counters regardless of whether the cache is being used or not. Moreover, a correlation analysis of the hardware counters was performed to find the best hardware counter combination.

There is a considerable quantity of work using bypass and adaptable caches as a solution to improve performance and reduce energy consumption. As far as we know, this dissertation is the first mechanism using linear regression to adapt the entire LLC use for each application, supporting a multi-program execution, and improving the general system performance.

The results for a single-application execution show performance improvements of up to 22% collecting as much as 25% of the performance achieved by an oracle mechanism. For multi-program executions, we observe an average system performance increase reaching 21%. However, some improvements in the prediction model are needed since the miss prediction observed decreased performance by 1%. On average, a performance increase of 7% is reached in different application configurations.

Therefore, we can conclude with this dissertation that a mechanism able to adapt the use of the LLC can lead to a more efficient system since the application that does not benefit from the use of the LLC does not need to face the extra latency barrier imposed by the cache hierarchy. Also, in a multi-program workload, this guide an increase in average system performance due to the reduction in global pollution and the reduction in cache conflicts and miss rate, especially seen in a mixed workload. Even considering configurations where non of the applications benefit on average from bypassing some improvements in general system performance could be observed, as the model predicts the execution windows where bypass can be performed for each application. Reducing the pressure in the LLC and favoring the other applications.

6.1 LIMITATIONS AND FUTURE WORKS

Nevertheless, DYCA presents limitations both in implementation and possible performance gains obtained using the mechanism.

One limitation observed arises from the simplifications done in a real system in order to be simulated. For instance, the size of each execution window used in DYCA is a fixed number of cycles, based in the average number of cycles in which the context switch happens on not simulated processors (3).

In future work, varying the execution windows' size, other than using the average value, could be implemented to improve the simulation accuracy when compared to a real processor.

Another aspect that can limit the DYCA performance is the size of the LLC. On one hand, in architectures with a smaller LLCs, the latency to access this cache level could be smaller however with a higher number of misses and in this case DYCA could represent a smaller increase in performance even though it reduces the number of misses.

On the other hand, for bigger LLCs this performance gain could be higher since the latency in this cache is also higher. However the number of misses tends to be smaller and in this case, it is important to analyze the accuracy of DYCA in different LLC sizes.

As future work, deeper analyses of DYCA results for each application can be done to better understand and improve mechanism accuracy. Along with using more applications in training and testing, going beyond applications from SPEC-CPU benchmark, since most applications do not benefit from DYCA. Another aspect to increase the mechanism understanding includes varying the LLC configuration, experiments could be performed to measure the impact of the LLC size on the performance of the mechanism.

The proposed sampling cache architecture could be another topic of future studies. Where a deeper analysis of the impact of the global and local sampling caches in the decision-maker mechanism can be performed to find the best configuration and parameters for the mechanism.

Also, energy consumption analyses could be executed to evaluate the possible energy saving provided by DYCA, when associated with a mechanism to turn off the LLC when this level is not being used.

Another future study that can be derivative from this dissertation is the use of different machine learning models, measuring the impact of using each different model associated with the hardware overhead of developing it.

REFERENCES

[1] Al-Obaidy, F., Asad, A., and Mohammadi, F. A. (2020). Learning-based reconfigurable cache for heterogeneous chip multiprocessors. In *2020 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5. IEEE.

[2] Al-Obaidy, F., Asad, A., and Mohammadi, F. A. (2021). Improving power-performance via hybrid cache for chip many cores based on neural network prediction technique. *Microsystem Technologies*, 27(8):2995–3006.

[3] Alves, M. A. Z. (2014). Increasing energy efficiency of processor caches via line usage predictors.

[4] Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *17th International Conference On High Performance Computing And Communications (HPCC)*, pages 605–610. IEEE.

[5] ARM, A. (2011-2013). Arm cortex-a15 mpcore processor. *Technical Reference Manual*, 1(1):392.

[6] ARM, C.-A. (2012). Series programmer's guide.

[7] Banday, M. T. and Khan, M. (2014). A study of recent advances in cache memories. In *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, pages 398–403. IEEE.

[8] Chang, K. K. (2017). *Understanding and improving the latency of DRAM-based memory systems*. PhD thesis, Carnegie Mellon University.

[9] Chaudhuri, M., Gaur, J., Bashyam, N., Subramoney, S., and Nuzman, J. (2012). Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 293–304.

[10] Chen, L., Zou, X., Lei, J., and Liu, Z. (2007). Dynamically reconfigurable cache for low-power embedded system. In *Third International Conference on Natural Computation (ICNC 2007)*, volume 5, pages 180–184. Ieee.

[11] Chi, C.-H. and Dietz, H. (1989). Improving cache performance by selective cache bypass. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track*, volume 1, pages 277–278. IEEE Computer Society.

[12] Chung, H., Kang, M., and Cho, H.-D. (2012). Heterogeneous multi-processing solution of exynos 5 octa with arm big. little technology. *Samsung White Paper*.

[Clark] Clark, M. Generalized additive models. `https://m-clark.github.io/generalized-additive-models`. Accessed: 2023-01-18.

[14] Egawa, R., Saito, R., Sato, M., and Kobayashi, H. (2019). A layer-adaptable cache hierarchy by a multiple-layer bypass mechanism. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6.

[15] El-Sayed, N., Mukkara, A., Tsai, P.-A., Kasture, H., Ma, X., and Sanchez, D. (2018). Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE international symposium on high performance computer architecture (HPCA)*, pages 104–117. IEEE.

[16] Gaur, J., Chaudhuri, M., and Subramoney, S. (2011). Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 81–92.

[17] Gupta, S., Gao, H., and Zhou, H. (2013). Adaptive cache bypassing for inclusive last level caches. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1243–1253. IEEE.

[18] Han, X., Fu, Y., and Jiang, J. (2016). Reconfigurable mpb combined with cache coherence protocol in many-core. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 385–388. IEEE.

[19] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[20] Hennessy, J. L. and Patterson, D. A. (2017). *Organização e Projeto de Computadores: a interface hardware/software*, volume 5. Elsevier Brasil.

[21] Hsu, P.-Y. and Hwang, T. (2013). Thread-criticality aware dynamic cache reconfiguration in multi-core system. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 413–420. IEEE.

[22] Intel (1998). Write combining memory implementation guidelines. *intel white paper*.

[23] Intel, I. (64). Intel 64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64):64.

[24] Jain, R., Panda, P. R., and Subramoney, S. (2017). A coordinated multi-agent reinforcement learning approach to multi-level cache co-partitioning. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 800–805. IEEE.

[25] Jaleel, A., Theobald, K. B., Steely Jr, S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3):60–71.

[26] Khan, S. and Jimenez, D. A. (2011). Decoupled cache segmentation: Mutable policy with automated bypass. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 212–212. IEEE.

[27] Kharbutli, M., Jarrah, M., and Jararweh, Y. (2013). Scip: Selective cache insertion and bypassing to improve the performance of last-level caches. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–6. IEEE.

[28] Kim, Y., More, A., Shriver, E., and Rosing, T. (2019). Application performance prediction and optimization under cache allocation technology. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1285–1288. IEEE.

[29] Köhler, R. and Alves, M. (2019). Acelerando requisições de prováveis cache misses com requisições em paralelo cache/dram. In *Anais Estendidos do IX Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, pages 101–106. SBC.

[30] Lai, C.-H., Yang, Y.-C., and Huang, J. (2014). A versatile data cache for trace buffer support. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(11):3145–3154.

[31] Liu, J., Egawa, R., Agung, M., and Takizawa, H. (2020). A conflict-aware capacity control mechanism for last-level cache. In *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 416–420. IEEE.

[32] McKee, S. A. (2004). Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162.

[33] Mittal, S. (2016). A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 6(2):5.

[34] Mittal, S., Cao, Y., and Zhang, Z. (2013a). Master: A multicore cache energy-saving technique using dynamic cache reconfiguration. *IEEE Transactions on very large scale integration (VLSI) systems*, 22(8):1653–1665.

[35] Mittal, S., Zhang, Z., and Vetter, J. S. (2013b). Flexiway: A cache energy saving technique using fine-grained cache reconfiguration. In *2013 IEEE 31st international conference on computer design (ICCD)*, pages 100–107. IEEE.

[36] Navarro, O., Yudi, J., Hoffmann, J., Hernandez, H. G. M., and Hübner, M. (2020). A machine learning methodology for cache memory design based on dynamic instructions. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(2):1–20.

[37] Park, J., Kim, S., and Hou, J.-U. (2021). An l2 cache architecture supporting bypassing for low energy and high performance. *Electronics*, 10(11):1328.

[38] Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology*, 64:1–18.

[39] Pricopi, M., Muthukaruppan, T. S., Venkataramani, V., Mitra, T., and Vishin, S. (2013). Power-performance modeling on asymmetric multi-cores. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE.

[40] Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. (2006). A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178. IEEE.

[41] Rhys, H. (2020). *Machine Learning with R, the tidyverse, and mlr*. Simon and Schuster.

[42] Santos, P. C., Alves, M. A., Diener, M., Carro, L., and Navaux, P. O. (2016). Exploring cache size and core count tradeoffs in systems with reduced memory access latency. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 388–392. IEEE.

[43] Sato, M., Chen, Y., Kikuchi, H., Komatsu, K., and Kobayashi, H. (2019). Perceptron-based cache bypassing for way-adaptable caches. In *2019 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3. IEEE.

[44] SPEC (2006). SPEC CPU 2006. https://www.spec.org/cpu2006. Online; accessed 08 November 2021.

[45] SPEC (2017). SPEC CPU 2017. https://www.spec.org/cpu2017. Online; accessed 08 November 2021.

[46] Teran, E., Wang, Z., and Jiménez, D. A. (2016). Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE.

[47] Tian, Y., Khan, S. M., and Jiménez, D. A. (2013). Temporal-based multilevel correlating inclusive cache replacement. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):1–24.

[48] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. (2012). Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224. IEEE.

[49] von Neumann, J. (June 1945). First draft of a report on the edvac. Technical report, University of Pennsylvania.

[50] Wilkes, M. V. (1965). Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2):270–271.

[51] Xie, X., Liang, Y., Sun, G., and Chen, D. (2013). An efficient compiler framework for cache bypassing on gpus. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 516–523. IEEE.

[52] Zhu, W. and Zeng, X. (2021). Decision tree-based adaptive reconfigurable cache scheme. *Algorithms*, 14(6):176.

APPENDIX A – DETAILED ACCURACY FOR THE SAMPLING CACHE

Detailed accuracy of the experiment described in Section 4.0.3. Table A.1 show the accuracy for SPEC CPU 2017 applications varying the number of sampling cache sets (16, 32, 64, 128 and 256), together with the mean and standard deviation for each used number.

Table A.1: Accuracy of different sampling cache sizes for SPEC CPU 2017.

Application	16 sets	32 sets	64 sets	128 sets	256 sets
bwaves	0.08	0.08	0.08	0.08	0.08
cactus	0.32	0.33	0.34	0.34	0.34
exchange	0.00	0.00	0.00	0.00	0.00
image	0.00	0.00	0.00	0.00	0.00
lbm	0.02	0.01	0.01	0.01	0.01
leela	1.41	0.15	0.42	0.82	0.74
mcf	3.45	2.60	0.96	0.98	0,69
nab	0.55	0.91	0.83	0.64	0.64
omnetpp	1.89	1.44	1.73	0.67	0.67
perlbench	2.45	3.34	2.64	2.64	2.64
roms	3.35	2.26	2.19	2.19	2.19
wrf	1.96	1.32	0.00	0.00	0.00
xalancbmk	0.00	0.00	0.00	0.00	0.00
MEAN	1.19	0.96	0.71	0.64	0.61
STD	1.26	1.11	0.88	0.83	0.86

APPENDIX B – DETAILED PREDICTIONS FOR MULTIPLE SYSTEMS

Detailed of the predicted IPCs for multiple systems using DYCA. Table B.1 show the predicted IPC values for *rom* application in each execution window.

Execution window	**Instructions per Cycle (IPC) wLLC**	**IPC woLLC**	**LLC configuration**
400M	1.32	1.22	wLLC
600M	1.34	1.15	wLLC
800M	1.09	1.30	**woLLC**
1B	0.91	0.85	wLLC
1.2B	1.36	1.35	wLLC
1.4B	0.91	1.07	**woLLC**
1.6B	0.85	0.74	wLLC

Table B.1: Predicted IPC for each execution window in *roms* application.

Additionally, Table B.2 show the predicted values for *wrf* application for each execution window until the end of application execution.

Execution window	**IPC wLLC**	**IPC woLLC**	**LLC configuration**
400M	1.49	1.22	wLLC
600M	1.48	1.23	wLLC
800M	1.35	1.00	wLLC
1B	0.82	1.18	**woLLC**
1.2B	1.19	0.75	wLLC
1.4B	1.36	1.05	wLLC

Table B.2: Predicted IPC for each execution window in *wrf* application.