TÉCNICO LISBOA

Runtime-Adaptable Cache Architecture for Modern Processors

Alexandre Carlos Martins Ruivo Correia

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Doutor Nuno Filipe Simões Santos Moraes da Silva Neves
Doutor Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Doutor António Manuel Raminhos Cordeiro Grilo
Supervisor: Doutor Pedro Filipe Zeferino Aidos Tomás
Member of the Committee: Doutor Nuno Cavaco Gomes Horta

December 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfils all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UNIFY (DOI: 10.54499/2022.06780.PTDC).

Abstract

High-performance processing systems must balance data processing with memory access latency to avoid performance bottlenecks as well as excess power consumption. Processor manufacturers address this by optimizing cache hierarchies to reduce cache misses and memory latency. However, traditional caches are static and unable to adapt to specific memory access patterns, either of particular applications or their distinct phases. For instance, caching unsuitable data structures can lead to cache pollution by evicting valuable, reusable data. Additionally, different application phases may vary in their reuse of data structures and exploitation of temporal and spatial locality, resulting in changing cache demands.

Recent research on adaptable cache architectures shows promise but often limits reconfiguration flexibility to avoid coherence issues, and overlooks the impact of modern prefetching. This thesis seeks to advance adaptable cache designs by identifying their limitations and exploring options for performance-oriented reconfiguration. Key objectives include mechanisms for reconfiguration, coherence maintenance, and cache autonomy. Experimental evaluations with and without prefetching, as well as varying adaptability constraints, demonstrate that the proposed architecture maintains performance, optimizes low-demand phases, and achieves up to 16% in cache power savings.

Keywords: cache adaptability, coherence management, prefetching, memory access patterns

Resumo

Sistemas de processamento de alto desempenho devem equilibrar o processamento de dados com a latência de acesso à memória para evitar restrições de desempenho e consumo excessivo de energia. Os fabricantes de processadores abordam este problema ao otimizar hierarquias de cache para reduzir misses e latência de memória. No entanto, caches tradicionais são estáticas e incapazes de se adaptar a padrões específicos de acesso à memória, seja de aplicações particulares ou das suas fases distintas. Por exemplo, guardar estruturas de dados inadequadas pode causar poluição da cache, substituindo dados valiosos e reutilizáveis. Além disso, diferentes fases de uma aplicação podem variar na sua reutilização de estruturas de dados e na exploração de localidade temporal e espacial, resultando em mudanças nos pedidos à cache.

Investigações recentes sobre arquiteturas de cache adaptáveis mostram-se promissoras, mas frequentemente limitam a flexibilidade de reconfiguração para evitar problemas de coerência e ignoram o impacto de técnicas modernas de prefetching. Esta tese procura melhorar os designs de cache adaptáveis, identificando as suas limitações e explorando opções para reconfiguração dedicadas ao desempenho. Os objetivos principais incluem mecanismos para reconfiguração, manutenção de coerência e autonomia da cache. As avaliações experimentais com e sem prefetching, bem como variações nas restrições de adaptabilidade, demonstram que a arquitetura proposta mantém o desempenho, otimiza as fases de baixa exigência e alcança até 16% de economia de energia de cache.

Palavras-chave: cache adaptável, gestão de coerência, prefetching, padrões de acesso à memória

Contents

Abstract . . . iv
Resumo . . . v
List of Tables . . . viii
List of Figures . . . ix
Acronyms . . . xi

1 Introduction **1**
1.1 Objectives . . . 1
1.2 Contributions . . . 2
1.3 Report Outline . . . 2

2 Background and State of the Art **3**
2.1 Cache Memory Fundamentals . . . 3
2.1.1 Cache Memory Origin . . . 3
2.1.2 Base Cache Structure . . . 3
2.1.3 Coherence Protocols . . . 7
2.1.4 Cache Memory Technology . . . 8
2.1.5 Data Prefetching . . . 9
2.2 Adaptable Cache Solutions . . . 10
2.2.1 Change in Size and Associativity . . . 11
2.2.2 Change in Line Size for Adaptable Caches . . . 15
2.3 Cache reconfiguration methodology . . . 16
2.3.1 Minimal Cache Flushing Solutions . . . 17
2.3.2 Configuration Decision Algorithm . . . 18
2.4 Discussion . . . 20

3 Runtime-adaptable Cache Architecture **21**
3.1 Motivation . . . 21
3.2 Starting Point . . . 24
3.3 Bank Concatenation . . . 25
3.4 Bank Allocation for Size and Energy Control . . . 27
3.5 Coherence Preservation . . . 28

3.5.1 Correction Policy . . . 29
3.6 Complete Cache Bank Selector Circuit . . . 30
3.7 Cache Reconfiguration Algorithm . . . 31
3.7.1 Manual Cache Reconfiguration . . . 31
3.7.2 Automatic Cache Reconfiguration . . . 31
3.7.3 Conflict Miss Identification . . . 31
3.7.4 Dead Set Identification . . . 32
3.8 Cache Request Differences . . . 32
3.9 Reconfiguration State Diagram . . . 32
3.10 Cache line composition . . . 33
3.11 Adaptability Overheads . . . 34
3.12 Complete Runtime-adaptable Cache Architecture . . . 34
3.13 Runtime Procedure . . . 36
3.14 Summary . . . 38

4 Evaluation **40**
4.1 Simulator . . . 40
4.1.1 Simulation Methodology . . . 41
4.2 Evaluation Criteria . . . 42
4.2.1 Performance Evaluation Metrics . . . 42
4.2.2 Power Consumption Analysis . . . 43
4.3 Different Evaluation Environments . . . 44
4.3.1 Fully Enabled Adaptability . . . 44
4.3.2 Fixed Size Evaluation . . . 50
4.3.3 Fixed Associativity Evaluation . . . 52
4.3.4 Only L1 Enabled Adaptability . . . 54
4.3.5 Only L2 Enabled Adaptability . . . 56
4.3.6 Fully Enabled Adaptability With Prefetching . . . 58
4.4 Discussion . . . 60
4.5 Summary . . . 61

5 Conclusion **62**
5.1 Future Work . . . 63

Bibliography **63**

List of Tables

3.1 Results of the cache size benchmark 22
3.2 Results of the associativity benchmark 23
3.3 L1 and L2 caches parameters. 36

4.1 Miss Rate evaluation of a fully adaptable cache system. 45
4.2 CPI evaluation with a fully adaptable cache system. 45
4.3 Miss Rate evaluation of a fixed size adaptable cache system. 51
4.4 CPI evaluation with a fixed size adaptable cache system. 52
4.5 Miss Rate evaluation of a fixed associativity adaptable cache system. 53
4.6 CPI evaluation of a fixed associativity adaptable cache system. 53
4.7 Miss Rate evaluation of a L1 adaptable and L2 static caches. 55
4.8 CPI evaluation with a L1 adaptable and L2 static caches. 55
4.9 Miss Rate evaluation of a L1 static and L2 adaptable caches. 57
4.10 CPI evaluation with a L1 static and L2 adaptable caches. 57
4.11 Miss Rate evaluation of L1 and L2 adaptable caches with prefetching enabled. 59
4.12 CPI evaluation of L1 and L2 adaptable caches with prefetching enabled. 59

List of Figures

2.1 Basic structure of a cache memory. 5
2.2 Base cache hierarchy. 7
2.3 Adaptation of a reinforcement learning system to a Prefetcher. 10
2.4 Performance and energy consumption comparison. 11
2.5 A way-concatenable four-way set-associative cache architecture. 12
2.6 Cache partitioning for a total size of 128kB. 13
2.7 Associativity control logic. 14
2.8 Size and Associativity addressing logic. 15
2.9 Indexing example on heterogeneous way size cache. 16
2.10 Virtual cache line. 16
2.11 Miss identification accuracy. 19

3.1 Simulation of caches with varying size with the different working sets mentioned if algorithm 1. 23
3.2 Simulation of caches with varying associativity. 24
3.3 Foundation for a runtime-adaptable cache architecture. 25
3.4 Possible adaptable cache configurations, controlled by a cache bank control unit. 26
3.5 Cache Bank Selector circuit. Responsible for associativity control. 27
3.6 Cache Bank Selector circuit. Adjusted for size control. 28
3.7 Swap policy, steps per cycle. 29
3.8 Cache Bank Selector circuit. Responsible for access remapping 30
3.9 Reconfiguration State Diagram. 33
3.10 Cache line comparison. 34
3.11 Basic structure of a Runtime-Adaptable Cache Architecture. 35
3.12 Execution pipeline of a Runtime-Adaptable Cache Architecture. 36

4.1 Miss rate evaluation of both L1 and L2 adaptable caches. 45
4.2 CPI evaluation of a fully adaptable cache system. 46
4.3 State frequency for both adaptable caches. 46
4.4 Frequency of each percentage of conflict misses collected during runtime for each state of the L1 cache. 47

4.5 Frequency of each percentage of conflict misses collected during runtime for each state of the L2 cache. 48
4.6 Frequency of each percentage of dead sets collected during runtime for each state of the L1 cache. 49
4.7 Frequency of each percentage of dead sets collected during runtime for each state of the L2 cache. 50
4.8 Miss rate evaluation of both L1 and L2 adaptable caches with fixed size. 51
4.9 CPI evaluation of a fixed size adaptable cache system. 52
4.10 Miss rate evaluation of both L1 and L2 adaptable caches with fixed associativity. 53
4.11 CPI evaluation of a fixed associativity adaptable cache system. 54
4.12 Miss rate evaluation of L1 adaptable and L2 static caches. 55
4.13 CPI evaluation of a memory system with an adaptable L1 and static L2 caches. 56
4.14 Miss rate evaluation of L1 static and L2 adaptable caches. 57
4.15 CPI evaluation of a memory system with a static L1 and an adaptable L2 cache. 58
4.16 Miss rate evaluation of both L1 and L2 adaptable caches with prefetching enabled. 59
4.17 CPI evaluation of a fully adaptable cache system with prefetching enabled. 60

Acronyms List

CPI Cycles Per Instruction

CPU Central Processing Unit

eDRAM Embedded Dynamic Random Access Memory

ISA Instruction Set Architecture

LLC Last Level Cache

LRU Least Recently Used

MOESI Modified Owned Exclusive Shared Invalid

MRU Most Recently Used

MSI Modified, Shared, Invalid

PC Program Counter

SRAM Static Random Access Memory

STT-MRAM Spin-Transfer Torque Magnetic Random Access Memory

Chapter 1

Introduction

In modern high-performance processing systems, the latency associated with a load of data from memory can heavily impact performance. Furthermore, the working set size of applications is increasing, which puts increasing pressure on the memory subsystem to quickly supply large quantities of data [1]. This highlights the importance of using an hierarchical cache memory that can significantly reduce the memory access latency while providing a high bandwidth.

Given the complex nature of cache memory, there is inevitably a trade-off when selecting its architecture and parameters to maximize performance across a range of applications while keeping in mind space restrictions and energy consumption. Moreover, modern computing systems execute diverse applications that have significantly different requirements and priorities [2, 3], as well as different memory access patterns in different groups of instructions.

As the structure of cache levels is statically defined, it does not optimally fit all applications, limiting its full potential. The ability to dynamically adapt aspects such as size and associativity during runtime can mitigate this issue. Moreover, adjusting these parameters can yield more advantages, as caching large or one-time access blocks of data is typically counterproductive: increasing power consumption as it may pollute the cache leading to capacity misses. There are already proposed implementations of adaptable caches that try to improve performance by fitting the hardware behavior to the requirements of applications, that have shown good results, [4, 5] achieving a reduction of around 17% of energy consumption without significantly hindering performance. However the decision process used to deal with the reconfiguration and the overheads generated by the design choices have room for optimization.

1.1 Objectives

With this in mind, the focus of this project is to investigate a new family of software-guided adaptable cache systems that can be changed in real-time by software, at the level of the whole application, at the level of a single program kernel (loop) or of a single load. For this, a set of relevant applications should be profiled to discover different optimization points. Taking these optimization points into consideration, a flexible cache structure should be proposed and validated either on a state-of-the-art simulator or by

simulating it with real hardware. Additionally, it can be relevant to propose a compilation pass to identify and automatically introduce commands in the program binary to re-configure the cache behavior.

1.2 Contributions

This thesis started from building upon already proposed implementations, while trying to understand their limitations and study different solutions and design choices that may further improve their performance.

The contributions of this thesis are as follows:

- Recreation of a state of the art architecture while also trying to reduce its overheads and limitations, namely minimizing or removing completely the need to flush multiple cache blocks at a time during a cache reconfiguration.
- Study of methodologies which were applied when changing the reconfiguration of a cache, based on different statistics and some basic phase confirmation logic.
- Analysis of the advantages and disadvantages of pairing an adaptable cache with a prefetcher. The respective results were considered in the final evaluation of the proposed architecture.
- Implementation of an architecture able to maintain performance, optimize for low-demand phases, and achieve up to 16% cache power savings.

1.3 Report Outline

This report is organized as follows: Chapter 2 briefly explains the basic architecture of a modern cache and displays the state of the art of the different adaptable cache implementations and other access pattern based cache optimizations. Chapter 3 introduces and explains the runtime adaptable cache architecture that is the focus of this thesis. Chapter 4 presents the results of many simulations done to evaluate the proposed architecture effectiveness. Finally, in Chapter 5 conclusions are taken based on the research work and on the achieved objectives.

Chapter 2

Background and State of the Art

Cache memory systems have long been a subject of investigation, given their crucial role in mitigating latency associated with memory accesses [6, 7]. Consequently, numerous implementations have been proposed. This thesis, however, centers on the dynamic adaptability of cache configurations during runtime.

To lay the groundwork, this chapter will initially define the standard hierarchy of a cache system and the fundamental structure of cache memory. This foundational understanding is crucial, as it serves as a basis for exploring different configurations. Subsequently, it will enumerate and describe the state-of-the-art architectures and techniques for adaptable cache systems.

2.1 Cache Memory Fundamentals

2.1.1 Cache Memory Origin

The initial concepts and implementations of cache memory trace back to the 1960s when the imbalance between the computational performance of real-time systems and the transportation of data from memory to the Central Processing Unit (CPU) became apparent and posed a limiting factor [8].

The notion of incorporating a small and faster memory component between the main memory and the CPU can be compared to having a journal with notes rather than carrying volumes of books. This innovative solution revolutionized computing and remains a fundamental component in every computer to this day. It paved the way for the development of faster and more efficient processing components, eliminating a significant bottleneck caused by memory access latency.

The Atlas 2, or Titan, created by the University of Cambridge in collaboration with Ferranti, was a pioneering example of these new types of memory systems [9].

2.1.2 Base Cache Structure

Cache memories are highly beneficial due to their capacity to take advantage of two crucial principles: temporal locality and spatial locality. Temporal locality suggests that if a memory address is accessed,

there is a good chance it will be accessed again in the near future. Similarly, spatial locality indicates that if a memory address is accessed, there is a high likelihood that neighboring addresses will also be accessed in the near future.

To take advantage of these principles, a cache is composed of multiple lines, each divided into data control and where the latter is typically a multiple of the processor maximum word size—around 32B, 64B, or 128B. For instance, a 32B cache line can accommodate 32 characters but only 8 32-bit integers.

Whenever a value is requested from main memory or a higher-level cache, the cache line is filled not only with that value but also with its neighboring values, such as to fill the whole line. The components of a cache will be explained in the following paragraphs, and Figure 2.1 illustrates a basic cache structure.

Cache indexing

To retrieve a particular value from the cache, an address must be supplied. The least significant bits, known as the offset, are utilized to pinpoint the initial byte within a line containing the desired value. The subsequent least significant bits, referred to as the index, determine the line within the cache. Finally, the remaining bits serve as a tag, ensuring that the accessed line corresponds precisely to the one requested by the CPU. This tag is also stored with every cache line to be compared at every access. This safeguards against the possibility of retrieving a different line which happens to be stored in the same cache location due to a conflict in the hashing function(i.e., the index bits).

Associative cache

Depending on how cache lines are organized it is possible to have a direct-mapped or an associative cache system. With the first option each cache set has only one line. The latter approach is more robust and involves dividing the cache into multiple identical ways. If two addresses share the same index, they will map to the same set but can be stored in blocks of different ways. This design allows the cache to simultaneously hold two values that would otherwise map to the same line, potentially causing conflicts.

During an access to a specific address, the cache compares the tags of all ways within to the incoming request, determining whether it results in a miss or a hit. This set-associative organization provides a middle ground between the simplicity of direct-mapped caches and the flexibility of fully associative caches, mitigating the risk of conflicts and enhancing overall cache performance.

Eventually, all the ways in a set become filled, and one of the lines must be evicted to accommodate another line requested by the CPU. While this decision could be made randomly without significantly impacting cache performance, a replacement policy is typically employed. Through the use of auxiliary bits, this policy can determine, for example, the line that was least recently used (LRU).

There is a correlation between the cache total data size, the size of a line, the number of lines and the associativity and it is described by in Equation 2.1.

$$\begin{aligned} \text{CacheSize} &= \text{Size of a Line (Bytes)} \times \text{Number of Lines} \\ \text{Number of Lines} &= \text{Number of Sets} \times \text{Number of Ways} \end{aligned} \tag{2.1}$$

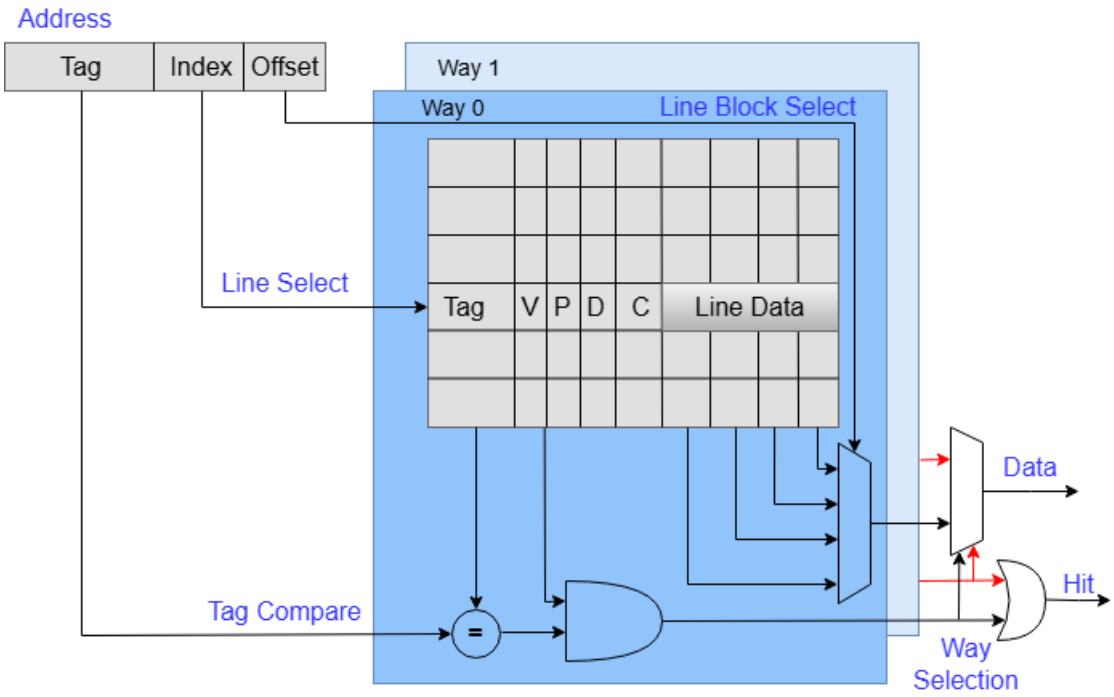


Figure 2.1: Basic structure of a cache memory. V:Validation bits, P: Replacement policy bits, D:Dirty bit, C: Coherence Bits.

Auxiliary bits

Beyond the tag and data blocks, each cache line incorporates additional bits with crucial information for maintaining the correct functioning of the cache. For example, a valid bit can indicate whether a cache line has been used and contains valid data ready to be requested or is still unused and should not be considered.

Replacement policy bits monitor which line will be evicted from a set based on the chosen policy. Additionally, a dirty bit indicates whether any data block in a line has been altered, signaling the need for an update in main memory or a higher-level cache before the line is evicted.

A set of validation bits upholds coherence information based on a pre-determined protocol, as explained in Section 2.1.3. Depending on the used protocol, these bits may indicate whether the data in the line is modified, if the cache owns line or it is shared with other caches, and whether the data is valid or not.

Types of cache misses

Each cache access has two potential outcomes: a hit or a miss. In the case of a hit, the requested data is delivered to the CPU. Contrarily, when the value is not found in the cache, it is considered a miss, and the request is forwarded to a higher cache level or the main memory.

There are four major types of possible misses:

1. **Compulsory Misses**: First access to a given address.

2. **Capacity Misses**: Access to an address that was accessed before, but was removed from the cache due to the working set size (smaller size would not result in a miss).

3. **Conflict Misses**: Access to an address that was accessed before, but was removed from the cache due to a conflict with another variable.

4. **Coherence Misses**(on multi-core systems): When one core reads a specific address, but another core alters it after the line has been loaded into the cache, rendering it invalid. Hence, a miss is generated if the first core attempts to read the value again.

Average Memory Access Latency

For each CPU request, it is possible to compute the average number of cycles necessary for the data to be provided for processing. In the case of a hit, there is an associated latency derived from the cycles required to map the address to a cache block, compare the tag, and transmit the data. In the event of a miss, this latency persists, but due to the absence of data in the initial cache level, a latency linked to a higher cache level or main memory access is multiplied by the miss rate and added to the total latency, i.e,

$$T_{\text{Latency\_Av}} = T_{\text{L1 Latency}} + \text{MissRate\_L1} \times T_{\text{MEM Latency}} \tag{2.2}$$

The equation for an access with a single cache layer is depicted in 2.2. In the presence of more than one cache level, their associated latencies are sequentially linked until main memory is reached.

Standard Cache Hierarchy

In modern cache memory systems, various levels of cache memory exist, each with different sizes. Each layer in this hierarchy involves a trade-off between its size and access latency. A larger cache size reduces susceptibility to capacity and conflict misses, but this advantage comes at the cost of increased access latency. This is due to the additional cycles required for indexing and data validation.

The standard practice is to have smaller caches positioned closer to the CPU. Only if the requested data is not present in that level will the system escalate the request to a higher level and ultimately to the main memory, which is associated with the highest access latency.

Since both data and instructions must be fetched from memory to the CPU, cache subsystems are typically designed to store both. To handle these requests, caches can be either split or unified [10]. It is common to split the L1 cache, and sometimes the L2, while unifying larger, higher-level caches, which offer more capacity and bandwidth. Each design has its trade-offs: unified caches are more cost-effective and occupy less hardware space, but they cannot serve data and instructions simultaneously. In a scenario where the CPU requests instructions every cycle, any load or store operation can create conflicts between these requests. A split cache architecture eliminates this conflict, though it slightly increases complexity and hardware requirements. The hierarchy is illustrated in Figure 2.2.

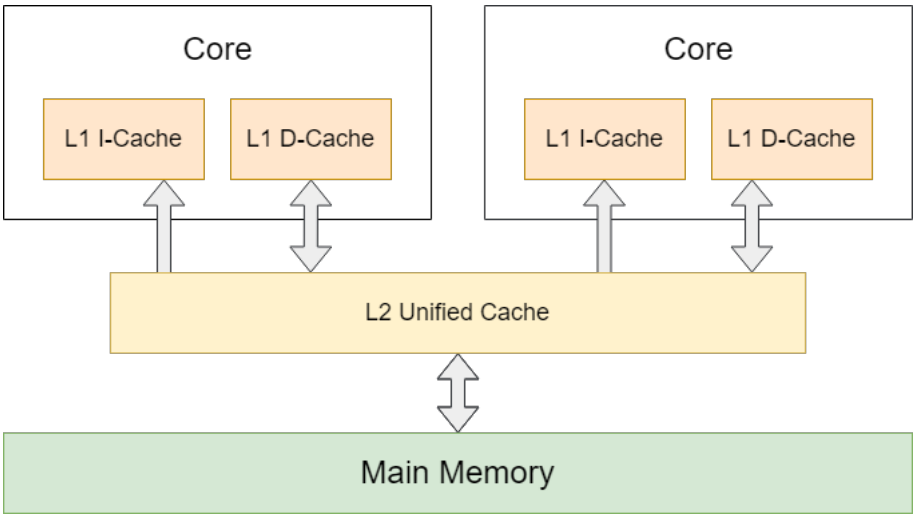


Figure 2.2: Base cache hierarchy.

2.1.3 Coherence Protocols

The first layer of cache memory not only is separated into a data and an instruction cache but is also associated with a single core. When dealing with multi-core systems it is imperative to keep coherence so that all processors, at any time, have a consistent view of the last globally written value to each memory location [11, 12, 13].

A simple example where having coherence is imperative happens when both a processor A and processor B read a block X from memory. If after both processors have a private copy of the value and one of them updates the X and only writes it back to the local cache, without updating other caches and the main memory, every other processor will perform incorrect operations because it has no way of knowing that the block is now invalid.

Each core is linked to main memory through a shared data bus which allows the cache controllers to track or snoop the memory requests made by caches from other processors, then depending on a coherence protocol a set of actions occur in order to keep coherence.

While snooping is effective, it has the downside of not being expandable, usually limited to systems of at most four cores [14, 15]. A popular alternative is the use of a directory scheme [16], stored in a centralized location, often near the main memory or Last Level Cache (LLC), holds the validity state of each line. If any core makes a memory request it is also sent to the directory to determine if another cache already possesses a valid copy of the memory block. Instead of a single storage location, it is possible to apply a distributed directory scheme where the directory entries are scattered between remote nodes and when a request is made the directory can instead redirect the request to the remote node that holds the information about the requested line.

The MSI (Modified, Shared, Invalid) [17] protocol is a straightforward and widely-used cache coherence mechanism that manages the states of cache lines in a multiprocessor environment. Each cache line can be in one of three states: *Invalid*, *Shared*, or *Modified*. When a cache line is in the *Invalid* state, it does not contain valid data, resulting in a miss when the processor attempts to access it. Thus, the data must be fetched from memory or another cache with the valid copy. In the *Shared* state, the cache

line contains a valid copy of the data that may also be held in other caches, allowing multiple caches to have the same line in this state and enabling read access without any modifications. Conversely, when a cache line is in the *Modified* state, it contains valid data that has been altered and differs from the copy in main memory. This cache is the only one with this line, and it must send an invalidation notification to the other caches when a modification occurs, ensuring coherence across the system. The MSI protocol effectively maintains cache coherence by ensuring that any modifications to shared data are appropriately communicated among caches.

The MOESI (Modified, Owned, Exclusive, Shared and Invalid) [18] is a robust and popular cache coherence protocol that associates each cache line with one of five states by changing how the validation and dirty bits of each line operate. If it is *invalid* a miss will be generated and the block must be fetched from memory or another cache with the valid data. The *exclusive* state indicates that this cache is the only one with a copy of the line. When more then one cache have a valid and unmodified copy of the data block the corresponding state is set to *shared*. Only one cache can have the state *owned* for a given line (every other cache with the same valid line is in the *shared* state) and it is responsible to update other caches when a modification is made to the block, implying a dirty sharing of data. When a modification is made to a line its state is changed to *Modified* and an invalidation notification is sent to the bus.

2.1.4 Cache Memory Technology

Different cache layers in modern processors can be made from various memory technologies, each chosen to balance speed, power consumption, and cost. Two of the most common memory technologies used in caches are Static Random Access Memory (SRAM) [19, 20] and Embedded Dynamic Random Access Memory (eDRAM) [21, 22], each with distinct characteristics that make them suitable for different levels of the cache hierarchy.

SRAM is known for its high speed, making it particularly suitable for applications where quick access to data is critical. It is structured using a series of flip-flops made from transistors, allowing it to retain data as long as power is supplied. This stability comes at the cost of increased power consumption and physical space since SRAM requires more transistors per memory cell. Due to its complexity and size, SRAM is limited in capacity.

On the other hand, eDRAM stores data in capacitors, which makes it much denser and less expensive to produce. However, because capacitors leak charge over time, eDRAM requires periodic refreshing to maintain its data, leading to higher latency compared to SRAM. Despite being slower and slightly less stable, DRAM is more power-efficient per bit stored.

Given these characteristics, L1 caches, which are the closest to the CPU, use SRAM because they require extremely fast access to data to keep up with the processor's speed. The small size of L1 caches, typically ranging from 4 KB to 128 KB, is a reflection of SRAM's high cost and power consumption, but the speed benefit is crucial for overall CPU performance.

L2 caches, while still needing to be fast, are generally larger, slower and slightly farther from the CPU

core. This allows for a balance between speed and capacity. Although L2 caches often use eDRAM, some designs also incorporate SRAM or a hybrid approach combining both SRAM and eDRAM. This hybrid design leverages eDRAM's higher density and lower power consumption, allowing for larger cache sizes without excessively increasing power usage or cost. The use of eDRAM in L2 caches, either on its own or in combination with SRAM, provides a compromise that maintains reasonable speed while significantly expanding the cache capacity, making it a versatile solution in modern processors.

For the LLC an architecture with Dynamic Random Access Memory (DRAM) can be used, but different technologies like Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM) [23] are also being tested and applied. The latter design offers a near zero static power and a high cell density at the cost of a high write latency.

2.1.5 Data Prefetching

An alternative method that seeks to minimize memory access latency by dynamically adapting to the access pattern of one or more applications is prefetching [24, 25, 26, 27, 28]. The core principle of prefetchers is based on the observation that applications memory accesses follow predictable patterns. Hence, if these patterns can be correctly predicted, data can be loaded from memory preemptively.

Prefetching can be performed by either software or hardware. Software prefetching inserts inline prefetch instructions in source code based on a programmer or compiler execution analysis. Hardware prefetchers use additional storage, typically lookup tables, to detect specific memory access patterns such as strided or stream accesses.

The simplest type of prefetcher, next-line prefetcher [29], proactively loads the following memory block after a requested line that originated a miss, anticipating future access needs. It was further improved to a strided prefetch where by repeatedly identifying the same stride between access and associating it with a confidence level it can correctly predict simple non sequential patterns.

Stream prefetching [30] tries to detect data streams that can be fetched from main memory before they are requested by the CPU and instead of saving the prefetched data in the cache, it is saved in a stream buffer, hence reducing unnecessary cache pollution. When the first line of the stream is requested it is then loaded into the cache and the next predicted stream will substitute the previous one.

Spatial prefetching [31] observes and records accesses to a loaded memory page while it is being actively used and once it is removed from the cache an event is associated with it creating an [event, pattern] pair and storing it in a table. When the event is triggered data from the linked memory page is loaded depending on its previous footprint, stored as a bit vector where each bit indicates if a specific line was used by the access pattern or not.

The Bingo prefetcher [25] is a hardware mechanism designed to enhance cache performance by predicting memory access patterns, especially in applications with irregular data structures like linked lists. It operates by tagging and marking accessed memory addresses, allowing it to recognize patterns in those accesses. Once a pattern is detected, the Bingo prefetcher fetches the anticipated cache lines into a buffer before they are needed, helping to reduce cache misses and improve performance. Bingo

achieves a 60% system performance improvement over a baseline with no prefetching and an 11% improvement over previous spatial data prefetchers.

A recently proposed hardware prefetching framework that takes advantage of reinforcement learning is Pythia [32]. The main objective of this implementation is to improve the prefetching accuracy and coverage when dealing with multiple program contexts at the same time.

The main idea consists of demanding requests at a defined frequency. With every new demand Pythia will take into account information about the state of the processor and the memory system, and based on it decide whether to prefetch or not and in the first case what data to prefetch. After each decision a numerical reward is feed into the algorithm based on the accuracy and timeliness of the prefetched data. A simple representation of this idea is shown in Figure 2.3.

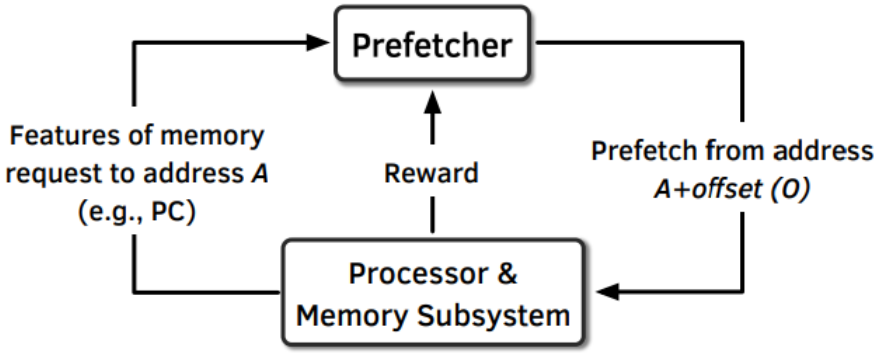


Figure 2.3: Adaptation of a reinforcement learning system to a Prefetcher from [32]

The final product of this proposed prefetcher implementation achieved a performance improvement of up to 20% when compared to other state-of-the-art prefetcher like Bingo[25] with only a 1.03% area overhead.

An alternative prefetching strategy was proposed by [33] and has the main objective of reducing hardware space overhead which is a large drawback of common used prefetchers.

Instead of storing various memory access patterns based on trigger request, which has the downside of requiring the eviction of certain entries when storage tables are full, accesses to the same memory region are merged while still maintaining a good prefetching accuracy.

This proposed prefetcher architecture, when compared to Bingo [25] achieves a 2.6% performance increase with 30x less storage overhead and an 8.2% performance increase with 6x less storage than Pythia [32].

An increase in dynamic and static energy was detected in memory hierarchy in the presence of recent prefetcher implementations [34]. In an attempt to reduce this effect a studies continue to be made to improve detection of critical instructions and perform prefetcher-specific threshold tuning.

2.2 Adaptable Cache Solutions

The concept of adaptable cache memory systems is not new, it has been studied for a considerable time. As a result, numerous strategies have been proposed [35, 36, 37], and quite a few of them overlap

in terms of their ideas. Three fundamental cache parameters—cache size, associativity, and cache line size—have exhibited substantial effects on both performance and energy consumption, these were illustrated in a small data cache by [38] on Motorola's Powerstone and MediaBench benchmarks and are represented in Figure 2.4.

As the authors concluded, by looking at the varying bar heights in each group of bars, the total cache size has the largest average impact on energy and miss rate. By looking at the difference in the same colored bar for different line sizes, we notice very little miss rate variation but significant energy disparity. Finally, by examining the same colored bars for different associativity, we notice very little change in energy consumption, indicating that associativity has a smaller impact on energy consumption than either cache size or line size.

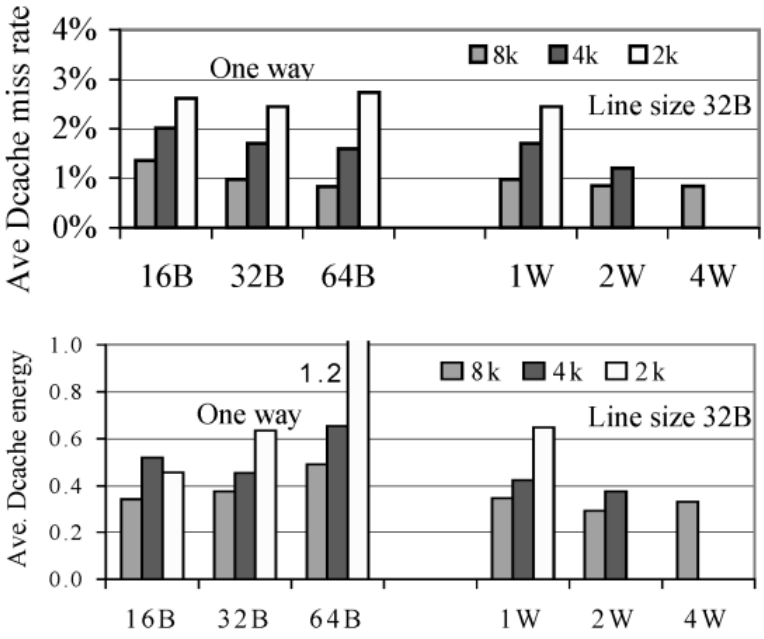


Figure 2.4: Performance and energy consumption with different cache configurations from [38].

2.2.1 Change in Size and Associativity

In general, a cache with a larger total size tends to yield performance improvements. However, when the working set is small, having a larger cache will not provide additional benefits, and the energy consumption will surpass what is necessary. Adjusting the cache size offers the possibility to minimize energy waste while ensuring a minimal miss rate for the executed application.

The influence of a specific cache associativity is not immediately evident, as it is heavily dependant on the memory access pattern. However, the ability to adjust cache associativity based on an access predicting algorithm can decrease the number of conflicts, thereby enhancing the cache's performance.

Various proposed implementations share common foundational ideas when it comes to altering the size and associativity of a cache [4, 5, 38, 39, 40, 41, 42]. Their architectures will be grouped and explained in a more detailed way in the following sub sections.

Way Concatenation

With the objective of controlling cache size and associativity, C. Zhang et al [38] and W. Wang et al [42] proposed a cache architecture approach where the cache is compact yet highly configurable. It offers options for a size capacity of 8, 4, or 2 kB, and associativity of 1, 2 and 4. Additionally, the size of a cache line can be adjusted, as discussed in Section 2.2.2.

This architecture, shown in Figure 2.5, makes use of 4 banks, each functioning as a cache way. Through a technique known as way concatenation, it becomes possible to control the associativity. This method involves integrating a small configuration circuit that does not impact the critical path. The circuit utilizes the last 2 bits of the tag and 2 control registers. When both registers are enabled, the cache operates with 4-way set associativity. Contrarily, when both registers are disabled, the cache functions as fully direct mapping. Enabling one register while disabling the other configures the cache to operate as 2-way set associative. On Powerstone and MediaBench benchmarks, this proposed implementation reduces on average 40% of total memory access energy over a standard static reference cache.

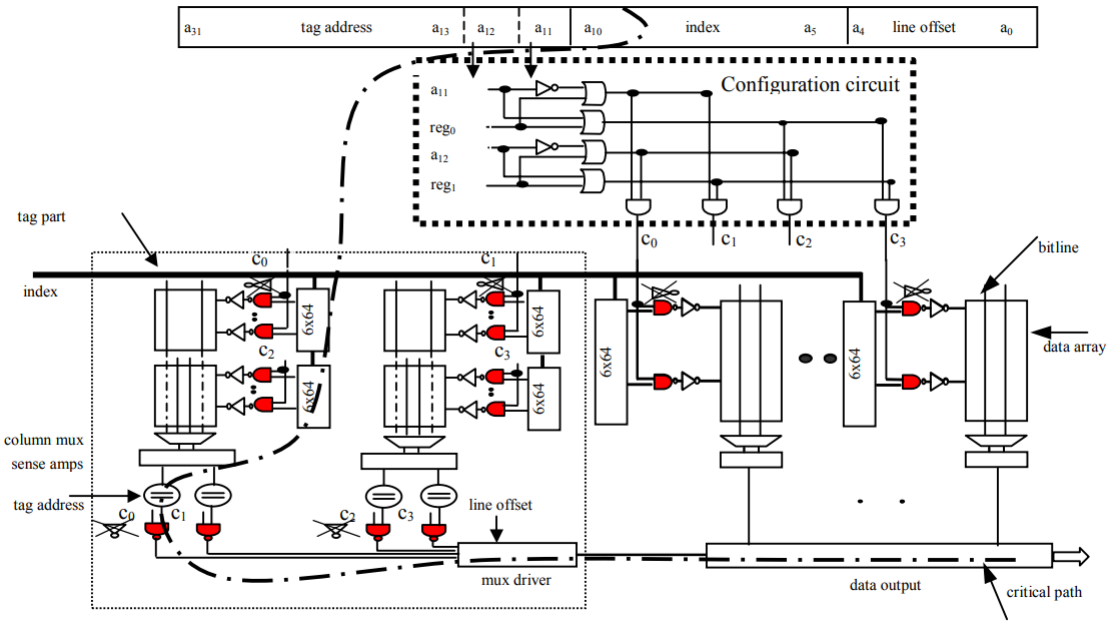


Figure 2.5: A way-concatenable four-way set-associative cache architecture with the critical path shown from [38].

Way Shutdown

To manage the overall size of the cache, the Way Shutdown technique is employed [42]. This approach consists of reducing the power feed to certain cache ways essentially allowing individual cache ways to be in a ”drowse state”, reducing both cache capacity and power consumption. A cache controller would decide which ways to shutdown depending on the desired cache capacity. At its limit, if all but one way is shut off—equivalent to the smallest cache size possible—the result would be a direct mapping configuration. This process introduces a marginal increase of approximately 5% in the critical path with

less than a 1% increase in area.

Cache partitioning

The design proposed by [39] tries to break the connection between cache capacity and associativity. To adjust the total cache size, the cache is organized with sub-arrays of varying sizes. For a maximum size of 128kB, it is divided into six sub-arrays: 2 with 4kB and the others with 8kB, 16kB, 32kB, and 64kB. The utilization of capacity registers, in conjunction with address bits, enables the selection of valid subarrays. The other disabled subarrays have their local word lines disabled, and their recharge and sense amplifiers are not activated, effectively reducing energy dissipation. This cache partitioning is illustrated in Figure 2.6.

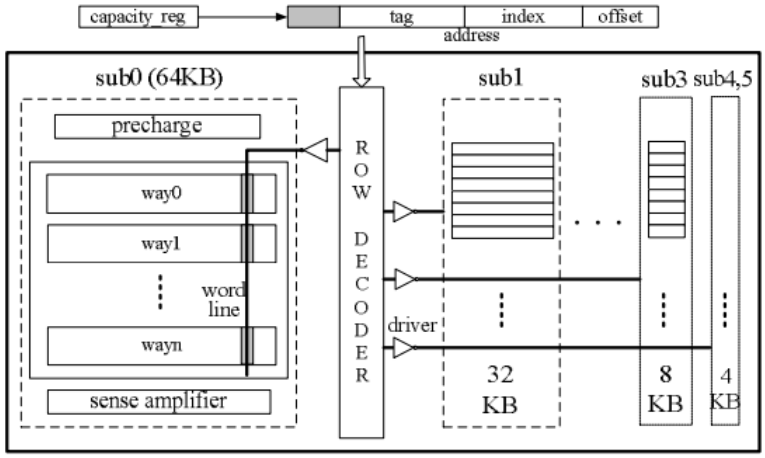


Figure 2.6: Cache partitioning for a total size of 128kB from [39].

Each subarray is subdivided into 8 ways, and to alter its associativity, controllable enable bits are employed. These enable bits, guided by decision logic, determine whether the data arrays are precharged, which word lines are selected, and whether the sense amplifiers are or not prevented from firing [39, 40]. An illustration of the architecture is presented in Figure 2.7.

Although the mechanisms for changing the size and associativity are different, there is still a correlation between the two parameters. When changing from an 8 to a 4 way set associative configuration half of the subarray is essentially disabled, consequently only half of the lines are used and therefore the total size of the cache has also been reduced.

Solution with Independent Parameters

The authors of [4] aim to enable independent adjustment of both cache size and associativity through two separate circuits that operate in parallel. One circuit manages cache capacity, disabling unused sets to adjust size, while the other circuit dynamically selects cache ways to adjust associativity as needed.

In such work, the tag size is set to accommodate the configuration requiring the most tag bits (i.e., smallest cache size and largest associativity). However, the least significant bits of the tag are shared by both circuits to identify the correct cache line based on the current configuration.

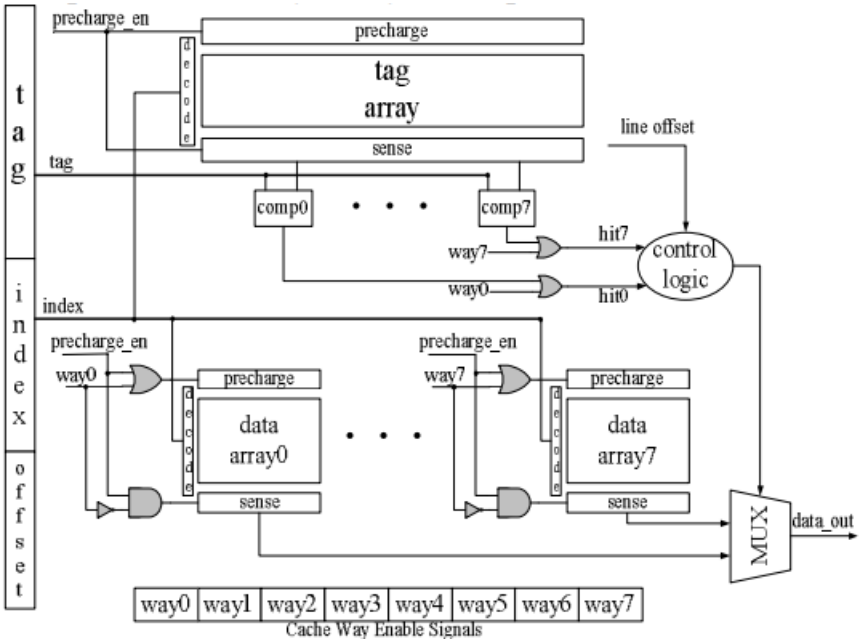


Figure 2.7: Associativity control logic from [39].

For cache size adjustment, it is necessary to control which bits are used for set selection (index). For that objective, the final tag bits help in deciding which sets are considered when hashing the desired block's address, as shown in the left circuit of Figure 2.8. A series of auxiliary selection bits can be turned on or off to adjust size. Each of these bits, when ANDed with the least significant tag bits, effectively change the number of index bits used. If all selection bits are off, only the default index bits are used. As more selection bits are enabled, additional tag bits contribute to indexing, altering the set count accordingly.

For associativity adjustment, it is necessary to control what ways are accessed in parallel or if only one at a time is used, simulating a direct-mapped cache. The second circuit, represented in the right side of Figure 2.8, uses two control bits and the two least significant tag bits to select the appropriate cache ways to be accessed, like the way concatenation method mentioned in Section 2.2.1. If any tag bit is already allocated to set selection, it is excluded from way selection, and the next available bits are used. Based on the control bit values, the cache can operate as direct-mapped, 2-way set associative, or 4-way set associative, as illustrated in Figure 2.8.

Unlike other approaches that disable cache ways to alter associativity, resulting in fewer active lines and effectively reducing cache size, this design simply reorganizes the ways without disabling them. Cache capacity can still be increased or reduced by changing the overall cache size. This approach show an average reduction in energy consumption of 17% in the data cache and 34% in the level-2 cache, with an overall performance degradation of less than 2% compared with a baseline statically-configured cache.

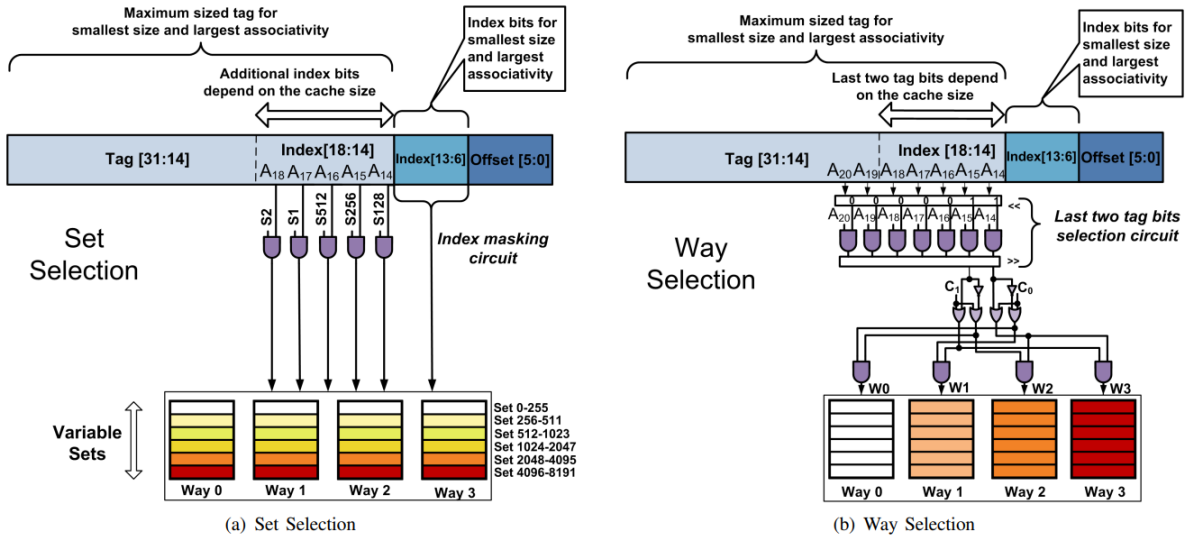


Figure 2.8: Size and Associativity addressing logic from [4]

Hybrid Cache Way Size

While no solution can fully prevent cache flushing or the additional misses caused by cache reconfiguration, there are methods to minimize these effects—particularly by carefully choosing the order in which cache parameters are adjusted [43]. Collected statistics from the Spec2000 benchmark suite simulations revealed that for the L1 data cache, 90% of the time, the utilization of the ways is distributed as 81% for the first way, 50% for the second one, 23% for the third one, and 6% for the fourth one, considering a four way set associative configuration. Which let them to the conclusion that only a small fraction of sets require some associativity most of the time.

To improve cache performance by changing the resource allocation of a cache they developed an heterogeneous way size cache, with this architecture more lines would be allocated to the first way and all subsequent ways would have less lines, but only the sets that were consistently being accessed would need to use all the available ways.

A slightly different mechanism of line indexing was developed to maintain coherence when accessing a value and continue to be able to map the index for each way in parallel. An example is shown in Figure 2.9, where there are four ways, one with eight lines, another with 2 lines and the remaining with four lines. The bits represented in the Address are only of the index bits since are the only ones that suffer any change, because with fewer lines less bits are required for a correct mapping.

2.2.2 Change in Line Size for Adaptable Caches

The size of a cache line is directly related to the cache's capacity to exploit spatial locality, resulting in increased performance, particularly in scenarios where an application exhibits a sequential access pattern. However, this advantage is not universally necessary. In cases where accesses are more dispersed, opting for smaller but more numerous lines can prove beneficial, all while maintaining the total cache size. Consequently, the capability to dynamically adjust the size of cache lines becomes a

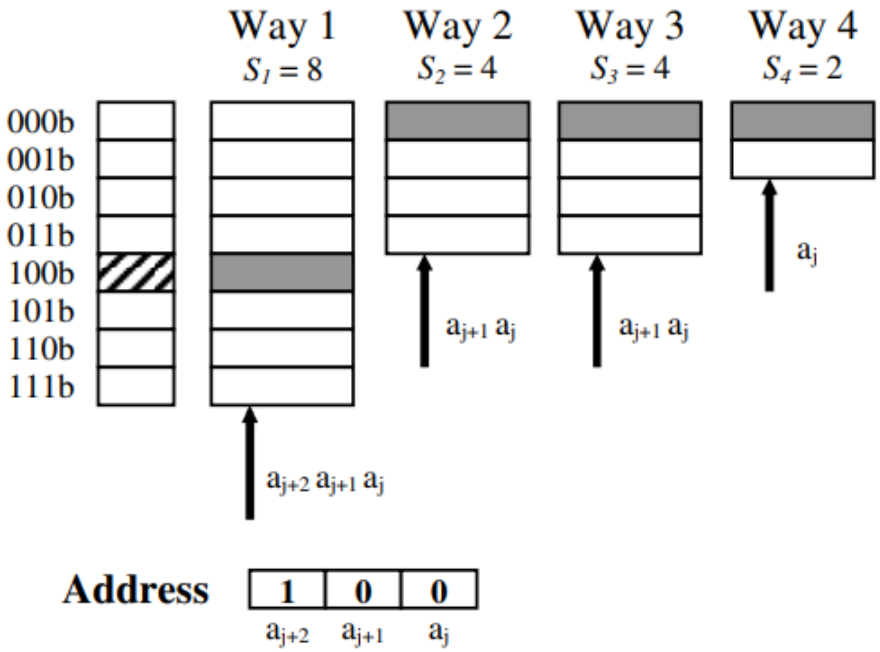


Figure 2.9: Indexing example on heterogeneous way size cache.[43]

worthy field for exploration.

A straightforward and efficient solution is line concatenation [38, 39]. The concept involves having smaller physical lines, such as 8B or 16B, but utilizing a larger ”virtual line” based on a variable register value. This virtual line can be of size 32B, 64B, 128B, or more. The specification of the desired virtual line size is only essential in two instances: when retrieving the required number of values from main memory or a higher-level cache to fill the line, and when a value is requested by the CPU, requiring consideration of all associated physical lines when determining the offset.

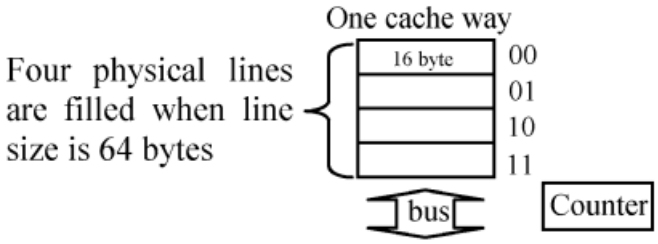


Figure 2.10: Illustrative representation of a Virtual cache line.[38]

2.3 Cache reconfiguration methodology

Changing the structure of a cache during runtime introduces some drawbacks and overheads. Primarily, the cache must be able to be configured for the worst-case scenario in terms of capacity, associativity, and line size. Consequently, there will be instances where not all cache resources are fully utilized. Therefore, a dynamic cache approach proves more beneficial in cases where an application or a set of applications have a diverse variety of memory access patterns that change during runtime.

Any decision mechanism will also have an associated latency that should be minimized in order to not significantly impact the overall performance.

Finally, if any alteration is made to a cache structure, such as reducing its total size, certain lines must be flushed back to a higher-level memory to maintain coherence. Additionally, the address mapping of data in the cache can suffer changes, leading to scenarios where values already present in the cache need to be reloaded, similar to compulsory misses.

2.3.1 Minimal Cache Flushing Solutions

Although no solution totally prevents cache flushing and the extra misses associated with cache reconfiguration there are ways to minimize them, especially when choosing the order by which the values of the cache parameters are changed [38]. A key insight is to always save the full tag and, for configurations requiring additional index bits, to share the least significant bits of the tag. This approach helps reduce cache miss mapping errors when reconfiguring the cache. As the authors point out, adjusting the capacity, associativity or line size of a cache can create different overheads depending on the order these changes are applied. Each of this scenarios is explained in the following paragraphs.

Total size

When adjusting cache size, it's preferable to start small and expand as needed. By increasing the number of lines, more data can be stored simultaneously in the cache, reducing the chance of evicting currently cached data. Naturally, the working set should be considered in this process.

Conversely, if a cache starts at maximum capacity and needs to be reduced for an incoming access pattern, fewer index bits are available, causing multiple data blocks to map to the same cache line and create conflicts. As a result, one of these blocks may need to be flushed to a higher memory level, which can be costly.

Associativity

Similar to adjusting cache size, modifying cache associativity can range from a fully direct-mapped configuration to the maximum associativity allowed by the architecture. When only the associativity is changed, the number of index bits is adjusted while keeping the total number of cache lines constant, only the number of sets and ways changes. This avoids introducing additional conflict misses. However, due to shifts in address mapping, some cache lines may relocate to different banks or partitions. When accessed for the first time after reconfiguration, these lines will generate coherence misses, since the coherence protocol in use was not notified.

Line Size

If the implementation uses line concatenation as described in Section 2.2.2, adjusting the line size parameter, whether increasing or decreasing, incurs no additional misses or flushing. Instead, the physical lines are simply remapped to the virtual line as needed.

2.3.2 Configuration Decision Algorithm

Designing an adaptable cache involves tuning various parameters, such as size and associativity. With increased cache flexibility, the parameter search space expands exponentially, making it essential to employ a procedure that minimizes impact on overall performance. Two key considerations are ensuring that the new configuration offers clear advantages over the previous one and carefully calibrating the intervals at which changes are made. Poor calibration can lead to high reconfiguration overhead or make adjustments ineffective if data access patterns shift before reconfiguration takes place.

Multiple decision mechanisms have been proposed to improve the behavior or configuration of a cache based on collected statistics [5, 4, 44, 45], they will be discussed in future subsections. Having an adaptable cache can give control to a developer to tailor its parameters for a specific application. However, the ability for the cache to change its configuration during runtime will not only reduce manual labor and development time, but also improve performance of a single application compared to a fixed configuration since the access patterns in many computational heavy applications are not constant.

Application Profiling with Machine Learning

A creative solution was presented by S. Charles et al [5]. They use a generic approach that takes advantage of machine learning to fill an application profile table and minimize the time required to simulate an application with every possible configuration. With a large search space, testing every configuration for a given application would not be feasible. Therefore given an application, by simulating only a few cases and using the results to train a neural network the authors showed it was possible to predict the performance of all other parameter combinations with that application. With this subset of combinations it is possible to determine the parameters of a machine learning model that minimizes an error function.

The described method, although useful for improving a developer productivity, is too much application specific, meaning a new training session was needed for all different applications, and does not have in consideration a reconfiguration in the middle of an application execution.

Reconfiguration Based on Runtime Statistics

By collecting statistics during the runtime of an application, it is possible to make an educated guess of the best cache configuration for future instructions [4]. This approach analyzes the collected statistics at fixed intervals to ensure that the overheads associated with reconfiguration are not significant. The two main statistics to be tracked are stack distance and dead set count.

1. **Stack Distance**: This metric keeps track of which line in a set has been accessed. If it was the LRU or a miss occurred, the cache may benefit from a higher associativity. Contrarily, if it was the most recently used (MRU), the cache can maintain a good performance with lower associativity.

2. **Dead Set Count**: Using a saturation counter for each set, it is possible to determine which sets are not being used over a defined interval of instructions. These sets are considered dead, and if

they are numerous, it is possible to conclude that the cache can function properly with fewer active lines.

By collecting these statistics and feeding them to a machine learning algorithm, it becomes possible to determine if the cache needs any reconfiguration and what parameters it should have.

Since [4] does not employ a line size change mechanism, no statistics were used to determine if the cache should use bigger or smaller lines. An hypothetical statistic for this parameter should be capable of assessing whether the cache would benefit from higher spatial locality (bigger lines) or temporal locality (more lines but smaller). Additionally, there are some edge cases where the stack distance would not predict that the number of ways should be reduced to improve performance.

Another important approach in cache optimization is identifying the types of misses that occur, as this can guide decisions on whether to adjust the cache configuration. Compulsory and capacity misses, for instance, may indicate that a larger or smaller cache could be beneficial. However, conflict misses are particularly crucial, as they directly suggest whether the cache's associativity should be modified.

One effective method for distinguishing conflict misses from non-conflict misses involves storing a portion of the tag from the most recently evicted line in a cache set [46]. When a subsequent miss occurs, the tag of the new access is compared with the stored tag from the previous eviction. If the tags match, it is highly likely that the miss was due to a conflict rather than another type of miss.

Storing the entire tag for every line would require significant additional space, which is impractical. To address this, tests were conducted to determine the minimal number of tag bits needed to maintain accuracy in predicting conflict misses. The results, as illustrated in Figure 2.11, show that after storing the 10 least significant tag bits, further increases to this number do not yield significant improvements in accuracy. Therefore, no more than the 10 last tag bits are stored for each line.

This method of miss identification is particularly useful when developing algorithms for dynamic cache reconfiguration. By accurately determining the type of miss, the cache can be more effectively tuned, ensuring optimal performance while minimizing unnecessary changes.

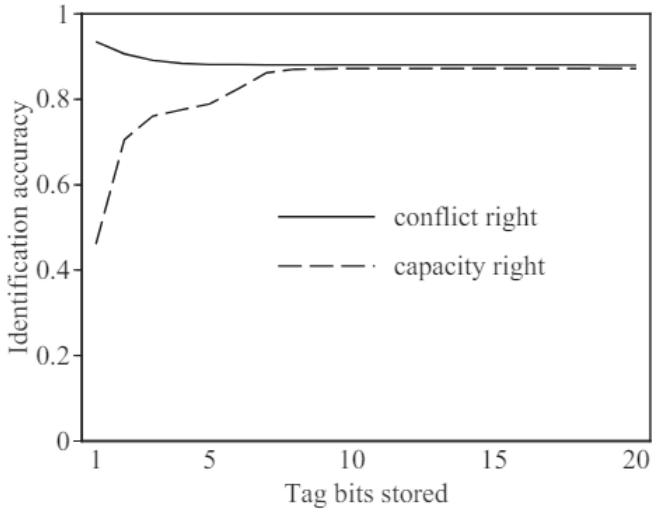


Figure 2.11: Miss identification accuracy over different sizes of stored tags. [46]

Finally, this methods makes the assumption that future instructions are related to previous ones, which is not always the case since access patterns can vary drastically during the runtime of an application. Reconfiguring the cache right before a program behavioral change can render it useless and only result in a performance decrease.

To counteract this issue there is the option of allowing the developer to manually control the cache structure for a specific block of instructions or create a mechanism that can interpret the data stream at compiler level and automatically infer how the cache should adapt during runtime.

2.4 Discussion

This section served as a way to contextualize and explain the cache memory fundamentals as well as dynamic mechanisms that seek to improve the performance of caches by taking into account the demands of different multi-phase applications.

As shown in this chapter, trying to improve on cache memory designs is a difficult task because there are always the resource limitations of the real world. The best implementation is not always obvious and most of the time is associated with a compromise. Nonetheless, new designs and architectures should be studied and tested in an attempt to maximize the performance of cache memory.

The approaches explained in this section will serve as a basis for future development, where a alternative runtime adaptable cache will be implemented and have its performance and power efficiency evaluated. Before trying to automatize the reconfiguration process, a possibility for the developer to choose the desired cache parameters should be created, and only after that will a decision mechanism based on the state of the art be employed and tested.

Chapter 3

Runtime-adaptable Cache Architecture

This chapter will motivate and describe the runtime adaptable cache architecture proposed by this thesis. Starting from a basic cache structure, different components will be explained and attached in order to reach the final objective of a dynamic cache with coherence preservation and minimal flushing. Automatic reconfiguration mechanisms will also be target of discussion.

3.1 Motivation

Choosing the ideal size for a cache is a complicated decision that requires consideration of the working set's size, the group of data to be used and accessed within a specific interval of instructions [47]. When the working set is very small, it is preferable to go for the smallest possible cache, minimizing energy consumption while maintaining great performance.

For a large working set, a larger cache is more suitable, even though it comes at the cost of increased energy consumption. It is essential to note that in conventional cache memory systems, a larger cache occupies more space that could be utilized for other resources. However, in the context of adaptable caches, the architecture accounts for the worst-case scenario, and consequently there is no difference in size independently of the configuration active at the moment.

The less obvious decision arises when the working set has a medium size. In such cases, a delicate balance must be weighted between performance, energy consumption and resource usage. Using more resources tends to improve performance, reduce execution time and can, therefore, also reduce energy consumption, but the delimitation between when to switch to a small, medium or large cache can be hard to make even with a lot of memory and processor statistics.

Finally, there is the scenario where the working set almost or completely exceeds the cache capacity. In such cases, the ideal approach is to abstain from storing the data in the cache. This is not only because it becomes useless, due to internal conflicts and capacity misses, but also because previously present data, potentially useful in the near future, would be trashed, leading to more misses when a

demand requires it to be reloaded. This phenomenon is commonly known as cache pollution.

The factors already mentioned only consider a single application running at a time. However, when multiple applications run concurrently, the decision process becomes considerably more complex. Working sets will be competing for space in the cache, and the chosen configuration must account for their size and behavior.

A group of simulations using algorithm 1 were conducted with the same access pattern over differently sized working sets, vectors of 8192, 4096, 2048 and 1024 4-Byte integers, and tested with static caches of also different size. The results of the simulations are represented in Figure 3.1.

Algorithm 1 Capacity Algorithm

```
Size ← 8192                                  ▷ [8192, 4096, 2048, 1024] depending on working set
N ← 0
A[Size], B[Size], C[Size] ← 1                ▷ 32-bit integer arrays
while N < 100 do
    i ← 0
    while i < Size/16 do
        C[i × 16] ← A[i × 16] + B[i × 16]    ▷ ×16 to access only 1 value per cache line
        i ← i + 1
    end while
    N ← N + 1
end while
```

Table 3.1: Results of the cache size benchmark

Cache Size (KBytes)	Small Set		Medium Set	
	Miss Rate	Number of Eviction	Miss Rate	Number of Eviction
16	0,008	156	0,670	25726
32	0,008	156	0,278	10675
64	0,007	130	0,009	271
Cache Size (KBytes)	Large Set		Huge Set	
	Miss Rate	Number of Eviction	Miss Rate	Number of Eviction
16	0,999	76798	0,999	153598
32	0,669	51450	0,999	153594
64	0,010	645	0,669	102812

As expected, there are scenarios where even the largest cache barely keeps up with the demands of an extremely large working set and, contrarily, scenarios where even the cache with lowest capacity can easily contain the entirety of the working set. This simple simulation highlights the disadvantage of utilizing a static cache over applications with very different memory requirements.

When deciding on the ideal cache associativity, it is a common practice to increase the number of ways to reduce the number of conflict misses and improve miss rate [48]. However, there are instances where a lower associativity or even a direct-mapped cache can yield better results [49].

This situation occurs when there are numerous different accesses to the same set, leading to conflict misses. Algorithm 2 tries to replicate this phenomenon by strategically accessing the same set in a series of cache memory requests. With a smaller number of ways and consequently more sets, the lines of that set would be more dispersed throughout the cache, potentially avoiding some conflicts and consequently evictions.

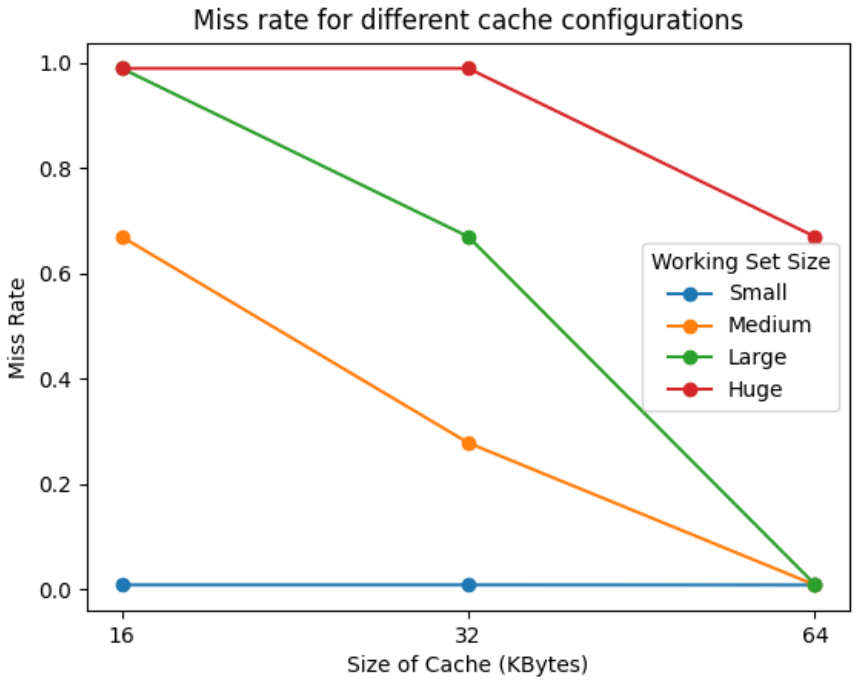


Figure 3.1: Simulation of caches with varying size with the different working sets mentioned if algorithm 1.

An example of this scenario was simulated with caches of different associativity and the results are shown in Figure 3.2. As predicted a lower associativity results in better performance for this benchmark.

Algorithm 2 Associativity Algorithm

$Sum \leftarrow 0$ ▷ Saved in a register
$N \leftarrow 0$ ▷ Saved in a register
while $N < 100$ **do** ▷ Arbitrary amount of repetitions
  $Sum \leftarrow Sum + Array[0]$ ▷ An array of 32-bit integers
  $Sum \leftarrow Sum + Array[512]$
  $Sum \leftarrow Sum + Array[1024]$
  $Sum \leftarrow Sum + Array[1536]$
  $Sum \leftarrow Sum + Array[2048]$
  $Sum \leftarrow Sum + Array[2560]$
  $Sum \leftarrow Sum + Array[3072]$
  $Sum \leftarrow Sum + Array[3584]$
  $Sum \leftarrow Sum + Array[6144]$
  $N \leftarrow N + 1$
end while

Number of ways	Miss Rate	Number of Evictions
1	0,114	206
2	0,180	305
4	0,279	503
8	0,498	899

Table 3.2: Results of the associativity benchmark

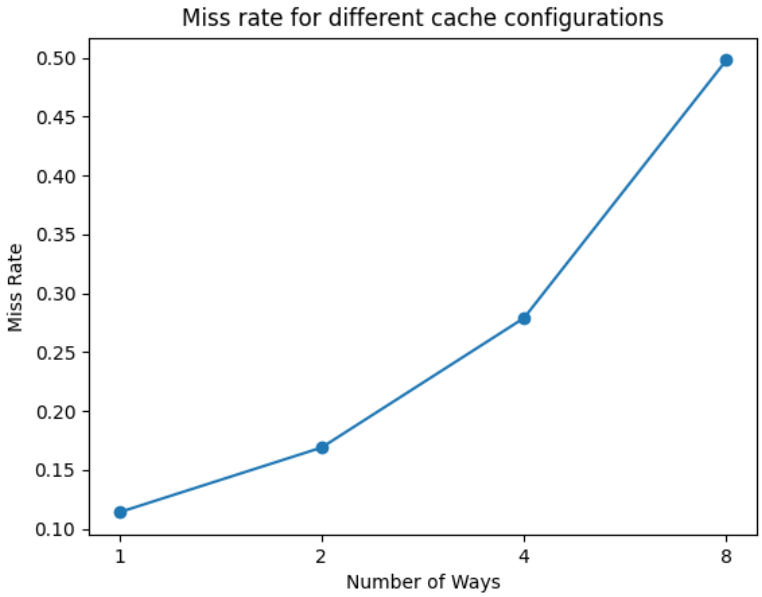


Figure 3.2: Simulation of caches with varying associativity.

Once more the importance of cache architecture flexibility can be extrapolated from this simple simulation. Where a static cache would not have an alternative when dealing with an access pattern with low compatibility, a dynamic cache can have a chance to change its structure during runtime and hopefully either improve performance or reduce energy consumption.

3.2 Starting Point

The purpose of a runtime-adaptable cache architecture is to dynamically adjust access behavior based on an application's demands, unlike a static cache. Starting from a basic cache structure, shown in Figure 3.3, this thesis aims to develop an adaptable cache architecture for the L1 data cache and L2 cache, offering way-associativity and size flexibility while maintaining coherence and minimizing reconfiguration overhead.

The adaptable architecture proposed in this thesis is designed with certain system constraints. It uses a write-back policy along with a LRU replacement policy, having snooping enabled with the MOESI coherence protocol. The maximum associativity is set to four ways for the L1 cache and eight ways for the L2 cache.

The first step is to develop a tool that can dynamically adjust how addresses are mapped to each cache way, allowing associativity to be modified quickly and efficiently rather than always maintaining maximum associativity.

Next, a mechanism is needed to control which cache ways are active and how future accesses respond to fewer active ways. This will enable size adaptability within the cache architecture.

Finally, since data may be mapped differently based on past and current cache configurations, coherence issues are expected to arise. To address this, an additional cache control method will be

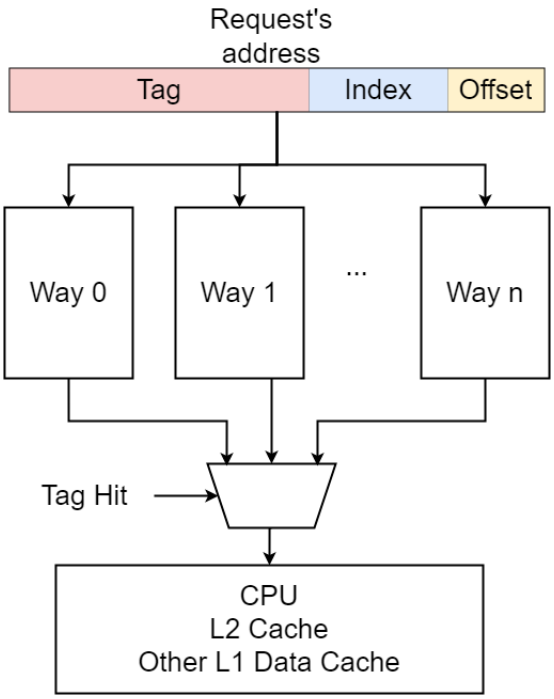


Figure 3.3: Foundation for a runtime-adaptable cache architecture.

implemented to maintain complete coherence while minimizing or even avoiding high-latency flush operations.

As the adaptable cache structure may not always support associative behavior, each subdivision of cache lines will be referred to as a cache bank. All reconfigurations will therefore involve adjusting how banks are mapped during each access. An overview of the desired behavior for the architecture to be developed is represented in Figure 3.4.

While any cache reconfiguration can be done manually, an algorithm for automatic runtime adaptability will be proposed to enable a more dynamic and comprehensive implementation. Cache behavior can be adjusted automatically, based on collected runtime statistics, or explicitly, by allowing a programmer or compiler to write to a designated address reserved for cache control.

To conclude this chapter, an analysis of adaptability overhead will demonstrate how this architecture can function without significantly impacting cache request latency compared to a standard cache.

3.3 Bank Concatenation

Each bank within the adaptable L1 cache functions as a distinct way, with a minimum associativity of one way and a maximum of four ways. In the L2 cache, due to a greater number of banks organized in pairs, the minimum associativity is two ways, and the maximum is eight. By using bank concatenation, selected banks can be accessed in parallel or sequentially, allowing the cache to operate with lower associativity while still providing more available lines per way.

With higher associativity, fewer index bits are required, as the number of cache lines remains constant, but the number of ways increases, which reduces the number of sets. The index size is therefore

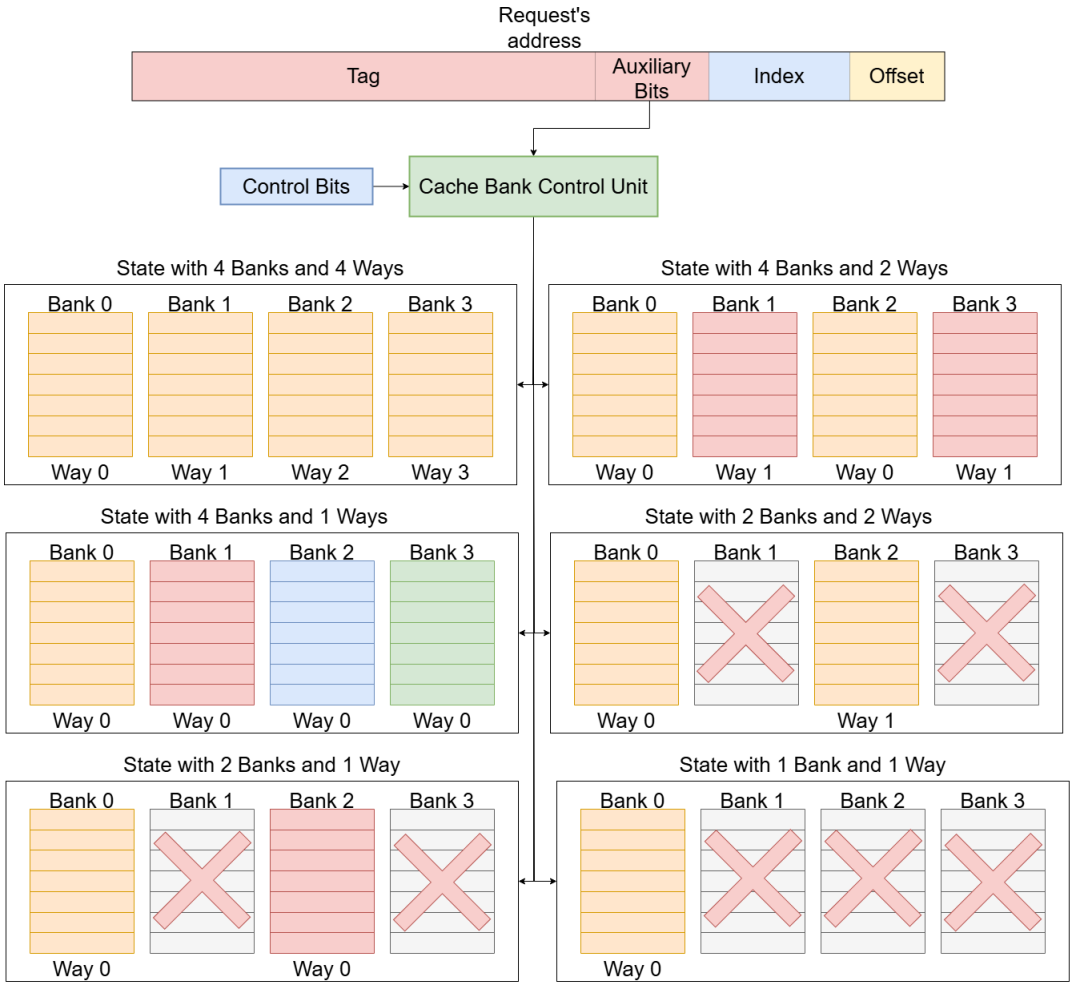


Figure 3.4: Possible adaptable cache configurations, controlled by a cache bank control unit.

configured for the maximum associativity when all banks are enabled. By using the two least significant tag bits along with two control bits in a logic circuit called the cache bank selector, shown in Figure 3.5, it is possible to determine which banks the requested address will map to.

The behavior of the control circuit is governed by two associativity control bits, labeled A0 and A1, which yield three distinct states. In state 00, cache banks are accessed sequentially, and the last two tag bits, T0 and T1, act as additional index bits. This configuration provides fully direct-mapped indexing in L1 and two-way associativity in L2. In state 11, maximum associativity is enabled, allowing indexing to rely solely on the index bits without the last tag bits. State 01 enables half the maximum associativity, with the T1 bit acting as an additional index bit; when T1 is 0, only half of the banks are accessed; when T1 is 1, the other half are accessed. The 10 combination is invalid, as a fourth state is undefined for this architecture, so this combination should not be allowed.

The control bits A0 and A1 are stored in a designated cache control address and can be modified automatically by a runtime reconfiguration algorithm or manually with an explicit write instruction.

Increasing associativity poses no coherence issues, as the larger set always contains all smaller sets within it. However, reducing associativity splits a set into smaller sets, which can lead to address mismapping. In this case, misses may be incorrectly assigned, as the requested data remains in the

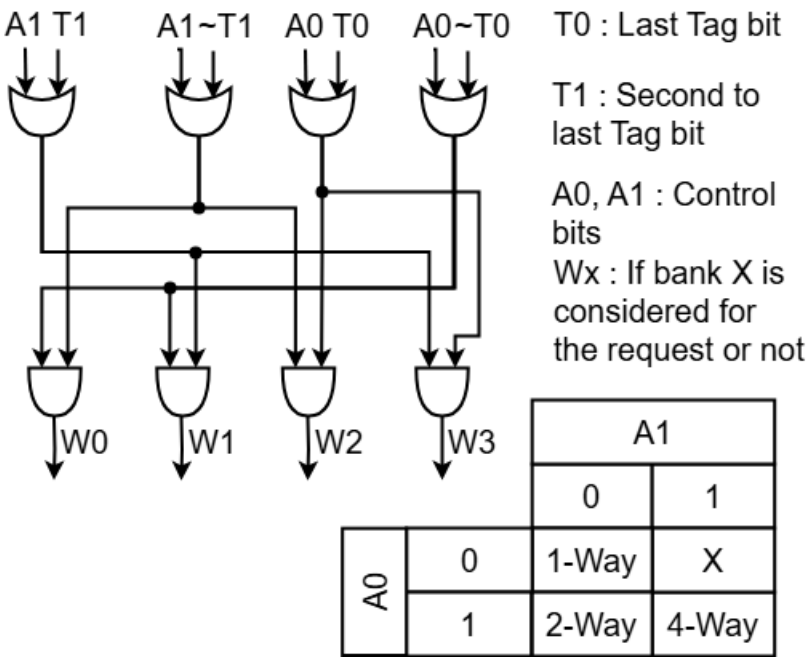


Figure 3.5: Cache Bank Selector circuit. Responsible for associativity control.

cache but is mapped to a different bank. Additionally, coherence issues can arise if no corrective measures are in place, which would be intolerable as they could invalidate program execution entirely.

3.4 Bank Allocation for Size and Energy Control

During certain phases of an application, it may be unnecessary for all cache banks to be active, as most reads and writes can often be handled by only one or two banks. By disabling half of the banks, the cache size effectively reduces by half; similarly, by disabling all but a single bank (or a single pair for the L2 cache), only a quarter of the cache is active, leading to an approximate 75% reduction in cache power consumption.

The Bank Selector circuit must adjust for active bank changes, ensuring that requests are mapped only to available banks. Figure 3.6 illustrates the modifications required for the selector circuit. To control this functionality, two additional control bits, S0 and S1, manage the number of active banks and prevent indexing to disabled banks. Each bit configuration results in a distinct state: 00 enables all banks; 01 enables half of the banks; and 11 activates only one bank for the L1 cache or one pair of banks for the L2 cache. The 10 combination is unused in the current design and is thus restricted, though future implementations could leverage it to enable selection of a single bank in the L2 cache, for example.

Reducing cache capacity, achieved by disabling specific banks, requires all valid lines in the disabled banks to be written back to a higher memory level, assuming a write-back policy. Conversely, increasing the cache capacity does not require flushing; however, requests may map to different lines than those where the desired data is stored. This can lead to coherence errors if precautions are not taken. As previously noted, coherence errors must be prevented to maintain the correctness of the executed application.

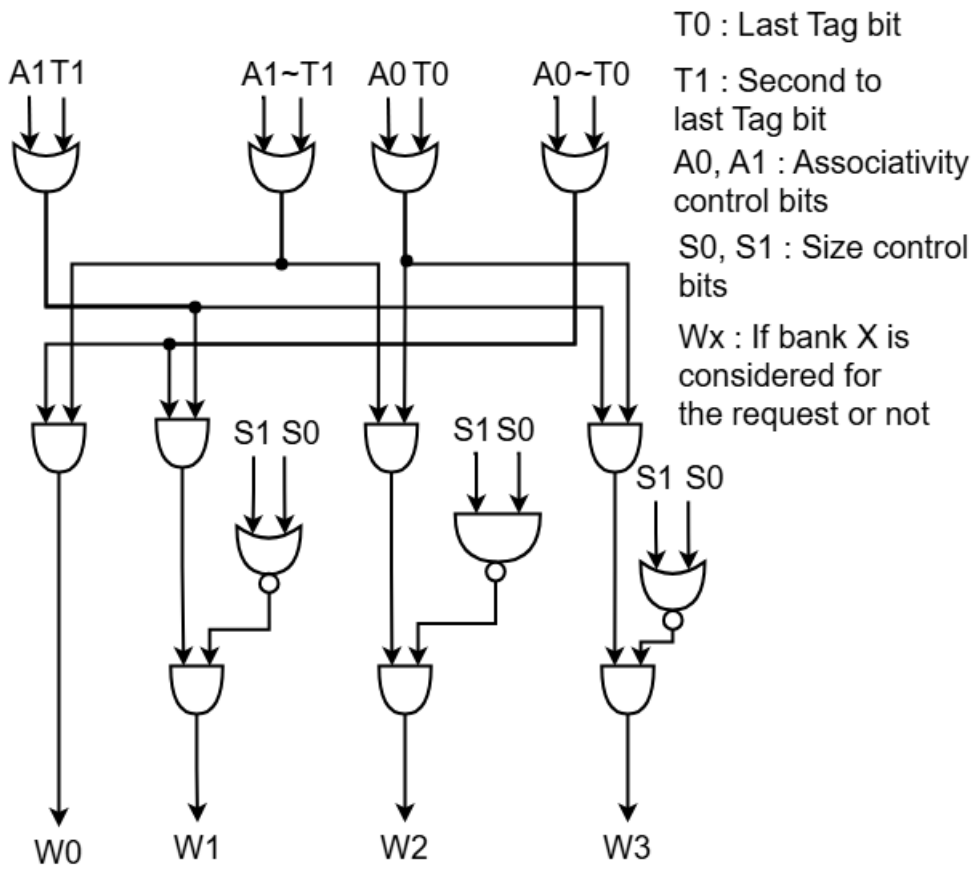


Figure 3.6: Cache Bank Selector circuit. Adjusted for size control.

3.5 Coherence Preservation

As previously noted, cache reconfiguration primarily involves remapping the request's address to a different cache line without prior flushing, which can lead to incorrectly assigned misses and coherence errors.

To address this, an additional bit, called the Correct (C) bit, is added to each cache line. This coherence-preserving approach ensures that any line has a valid location in at least one of the four bank groups under the same index.

In any set, if a line has its C bit set to 0, its tag is always compared as in a standard associative static cache. Several scenarios are possible: for example, if a request to a line marked with a 0 (indicating "incorrect") results in a hit in the correct bank, the C bit is set to 1 ("correct"). Conversely, if the line is in an unexpected bank, a correction policy is applied to move the block to the appropriate bank based on the current configuration, after which the C bit is set to 1.

Thus, any cache line that is loaded correctly should have its C bit set to 1. Each time a reconfiguration reduces associativity or re-enables a previously disabled bank, all C bits must be reset to 0 to ensure coherence. This reset operation has minimal overhead, which will be discussed further in Section 3.11.

3.5.1 Correction Policy

The correction policy ensures that any target line is placed in the expected bank as determined by the Cache Bank Selector. Two main approaches are considered for this policy: the Place policy and the Swap policy.

1. **Place Policy**: The target line will be placed in the correct bank with the C bit set to a 1. Its original place will be left with a line with its Invalid bit and C bit set to 1, as to not interfere with any future snoop operations. If a valid line was in the destination it will be properly flushed before the placing is executed.

2. **Swap Policy**: The target line will swap places with the line present in the destination, and its C bit will be set to 1. The victim line will not suffer any change to its Correct or coherence bits. As shown in Figure 3.7 the requested line is immediately sent to the requester before the swap is completed, hiding any additional latency. On SRAM cache this operation needs at least two cycles to be completed since a read and write cannot be performed simultaneously in the same memory cell.

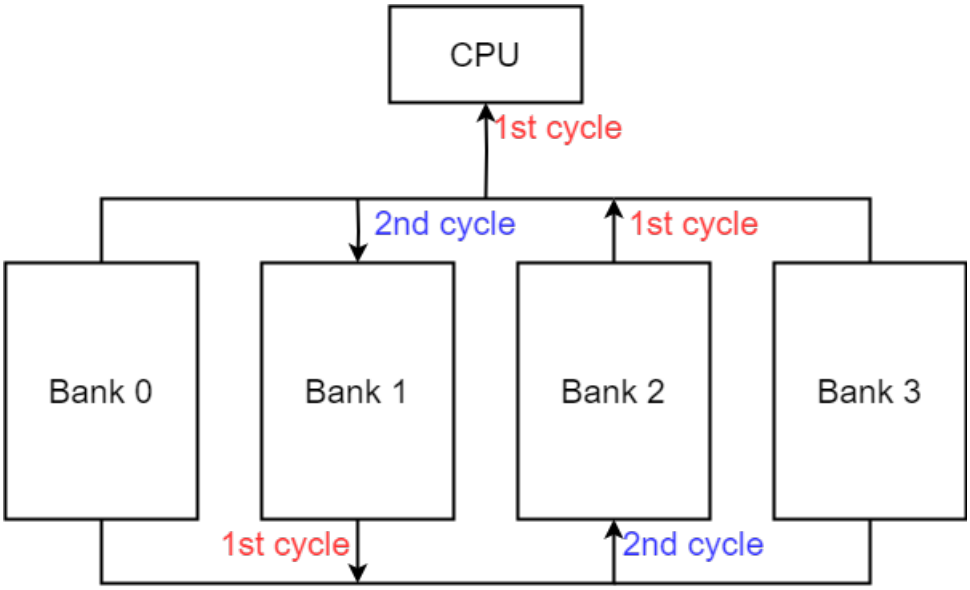


Figure 3.7: Swap policy, steps per cycle.

Among the two options, the Swap policy allows data to remain in the cache longer and introduces no additional latency from the CPU's perspective. This is possible because the line is returned to the CPU before the correction operation completes, ensuring all lines are correctly positioned in their respective banks before the next request's address is decoded. Consequently, the Swap policy was selected for this implementation.

This coherence preservation method gradually transitions the cache from a higher associativity configuration to a direct-mapped structure, rather than making an abrupt change. It maintains coherence effectively with minimal overhead.

3.6 Complete Cache Bank Selector Circuit

The control circuit for managing the adaptable cache's size and associativity must also account for the coherence-preserving Control bit. This bit should override any bank selection if it is set to 0, indicating that the corresponding line might reside in an incorrect bank and requires correction.

The circuit's essential features—such as the ability to control which banks are accessed in parallel via the last tag bits and auxiliary control bits—should remain intact to simulate varying associativities. Any disabled bank should be excluded from access; its data is flushed to a higher memory level, with future accesses remapped to active banks accordingly.

The intended control behavior is illustrated in Figure 3.8, showing the upgraded circuit design. Since control bits are fetched only after set indexing, this circuit is applied when selecting tags for hit or miss checks. As a result, all banks should perform tag comparisons, similar to a static associative cache, to account for any misplaced lines. By placing the cache bank selector circuit after tag matching, the design avoids additional latency from reading all Correct bits. Due to the minimal gate count in this circuit, it does not impact the critical path of normal cache access operations.

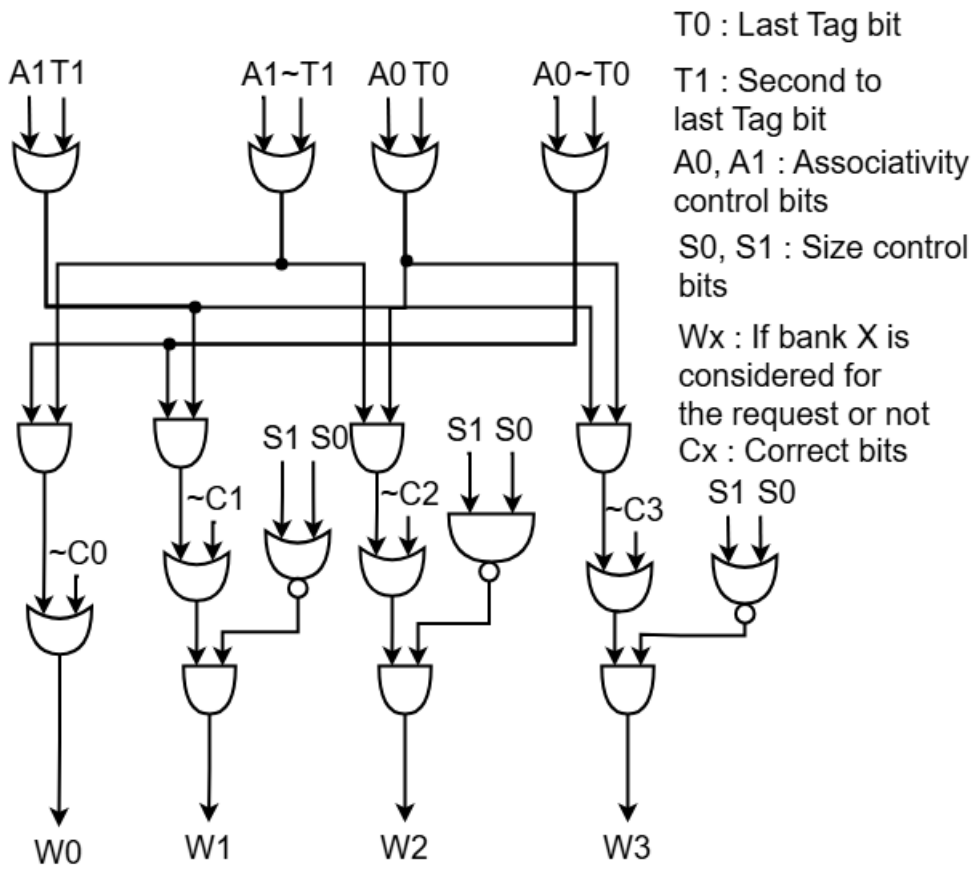


Figure 3.8: Cache Bank Selector circuit. Responsible for access remapping

3.7 Cache Reconfiguration Algorithm

The runtime adaptable cache can be reconfigured either manually or automatically. In both cases, modifying the cache behavior only requires adjusting four control bits: two for associativity and two for size. These control bits are stored in a physical address specifically reserved for cache control.

3.7.1 Manual Cache Reconfiguration

Enabling manual cache control, rather than limiting it solely to automatic adaptability, opens possibilities for external tools and heuristics to dynamically set cache behavior during execution. This flexibility is beneficial in scenarios where an automatic algorithm may yield inaccurate predictions, allowing a fallback to maintain cache functionality while addressing any issue or bug.

To manually adjust the cache configuration at runtime, a developer can issue instructions that write directly to the address storing the control bits, tailoring cache behavior to specific application phases. These control bits include A0 and A1, responsible for associativity selection, and the S0 and S1 bits, that control cache capacity.

Similarly, if the compiler can accurately predict runtime access patterns and the necessary cache configurations for different phases, it may also issue write instructions to the control bits.

3.7.2 Automatic Cache Reconfiguration

Automatic cache reconfiguration relies on collecting runtime statistics to predict future access patterns based on past requests accurately. The selection and weighting of these statistics are critical, as incorrect reconfigurations can create unnecessary loops, adding avoidable overhead.

A runtime-adaptable cache is only beneficial over a static one if changes to the cache state improve execution performance or reduce power consumption enough to offset reconfiguration overhead. This balance is essential for achieving a net positive impact on performance.

To avoid frequent, disruptive adjustments, the reconfiguration frequency must be carefully calibrated. A statistic collection period based on fixed intervals of cache accesses allows for consistent data gathering. The first three periods following a reconfiguration are discarded to let the cache stabilize. Additionally, when a key statistic exceeds a set threshold, suggesting a need for cache adjustment, it must surpass this threshold several times consecutively to ensure a confident prediction before initiating reconfiguration.

3.7.3 Conflict Miss Identification

To decide whether a change in associativity is needed, it's essential to examine not just the overall miss rate but, more importantly, the types of misses. Adjusting associativity addresses conflict misses specifically, while compulsory or capacity misses remain unaffected. Thus, identifying and focusing on conflict misses can improve adaptability.

For conflict miss identification, as suggested by [46], the cache stores the 10 least significant tag bits of the last evicted block in each set. If a subsequent miss in the same set has matching tag bits, there is nearly 90% confidence that it is a conflict miss.

Since the cache must support a fully direct-mapped structure, each line requires space for these additional 10 bits. On a 64-bit system, the L1 data cache holds 64kB of data with 64B per line, plus about 5kB for tag and control bits. Allocating space for this 10-bit addition in each line requires approximately 1.28kB more, a manageable increment.

3.7.4 Dead Set Identification

The necessary cache capacity can be gauged by assessing the percentage of recently used versus unused sets. When this percentage is low, fewer active lines may suffice without impacting performance. On the other hand, if usage is close to 100%, enabling more lines can help prevent capacity misses.

This percentage is derived by identifying dead sets, those accessed fewer than three times during a collection period. The threshold of three accesses was chosen based on a 2-bit saturation counter, which proved more stable than a 1-bit counter (prone to outlier influences) and more efficient than a 3-bit counter, which offered no added benefit.

3.8 Cache Request Differences

In a standard static cache architecture, the index determines the set, and all tags in that set are compared with the request's tag. On a cache hit, data from the selected line is returned to the CPU or a different cache requesting the data, completing the operation in about 4 cycles. In the event of a miss, the request is sent to a higher memory level, resulting in a significantly longer wait to retrieve the required data.

In the proposed adaptable cache, additional steps, specifically cache bank selection and checking the Correct bit for coherence, are incorporated. However, as these processes only impact the selection of the correct cache line and can occur in parallel with other cache operations, they add no extra cycles to each cache access.

3.9 Reconfiguration State Diagram

The size and associativity of each cache can be varied simultaneously, but these changes are not always compatible. For instance, with only one or two banks enabled, it is impossible to support a four-way set-associative configuration. Figure 3.9 illustrates a state diagram that maps the transitions between configurations.

The adaptable cache begins execution with maximum size and associativity enabled. This design choice was primarily made to demonstrate the feasibility of remapping cache lines with minimal overhead while still maintaining coherence. At fixed intervals, the reconfiguration algorithm assesses the

percentage of conflict misses and dead sets. Based on these statistics, it determines whether a state change is necessary and, if so, which state the adaptable cache should transition to. If the conflict percentage is low the adaptable cache will go from four ways to two ways and possibly to one single way and vice-versa. If the dead set percentage is high the adaptable cache will reduce its capacity to only two active banks and possibly only one active bank and vice-versa.

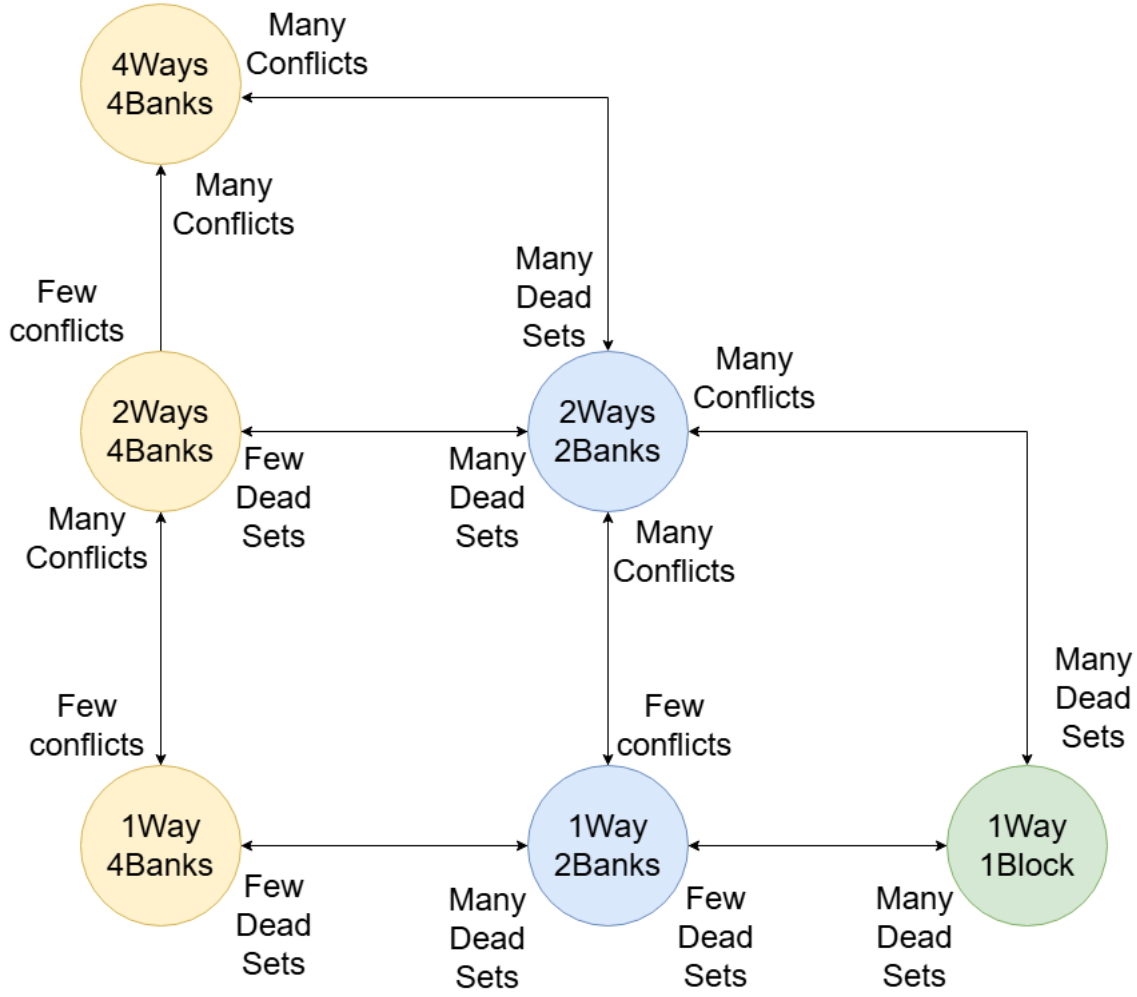


Figure 3.9: Reconfiguration State Diagram.

When a manual reconfiguration is initiated, any state can be immediately executed. In contrast, automatic reconfigurations are performed gradually to more effectively adjust the cache structure to the anticipated requirements of the current application phase.

3.10 Cache line composition

As previously discussed, each cache line must store additional information to ensure the desired functionality of the runtime-adaptable cache. This includes the Control bit for coherence preservation, two bits for the Dead Set count, and the last ten bits of the most recently evicted line from the set. Although the number of bits could be adjusted, research by J. D. Collins et al. [46] and simulation results indicate that ten bits are the minimum necessary for accurately predicting conflict misses. Consequently, this

results in a total of 13 extra bits per cache line compared to a standard cache. The distinction between configurations, along with a potential bit layout, is illustrated in Figure 3.10.

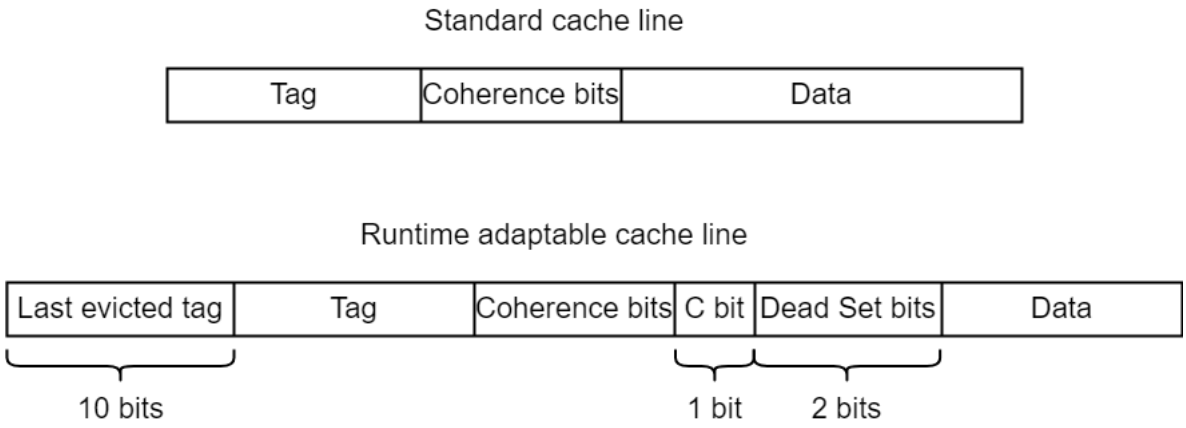


Figure 3.10: Cache line comparison.

3.11 Adaptability Overheads

Each change to cache parameters incurs an overhead that can range from negligible to significant. For instance, increasing the associativity primarily involves updating control bits, a process that can be completed within a single cycle. Although some cache lines may need to be relocated during future accesses, the CPU does not experience any noticeable latency increase since these remain cache hits.

In contrast, reducing associativity requires not only changing the control bits but also resetting the C bits of all cache lines. This process can be executed in one of two ways:

1. **Resetting One Line at a Time**: This method adds cycles equal to the number of lines being reset. While this approach is not ideal, its impact on performance can be mitigated if the resulting cache configuration compensates for this sporadic latency increase.

2. **Column Reset**: By organizing the C bits into a single column or grouping them by cache banks, a column reset can be performed, requiring only one cycle per cache bank.

The overhead associated with changing the number of active banks depends on the chosen methodology, such as completely shutting down a bank or placing it in a drowsy state. Regardless of the approach, increasing or decreasing the number of banks consumes additional energy, and it takes several cycles for a bank to become fully operational after reactivation. Moreover, when disabling a bank, its data must be written back and invalidated to maintain coherence before any future accesses.

3.12 Complete Runtime-adaptable Cache Architecture

To achieve the desired behavior, the L1 data cache is divided into four banks, which can be mapped either in parallel, functioning like an associative cache, or sequentially, simulating a fully direct-mapped

cache, as illustrated in Figure 3.11. In the case of the L2 cache, there are eight banks organized into pairs, allowing for configurations with a minimum associativity of two ways and a maximum of eight ways.

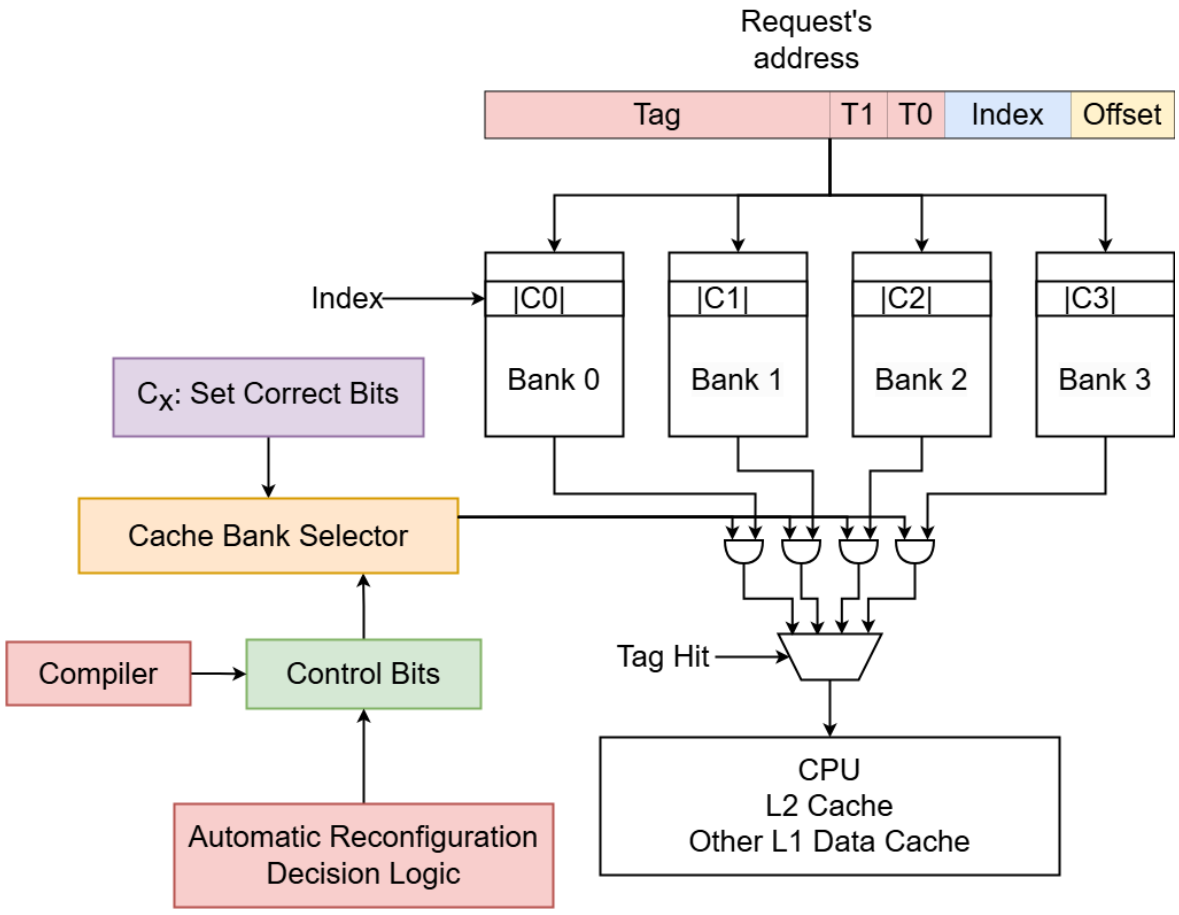


Figure 3.11: Basic structure of a Runtime-Adaptable Cache Architecture.

This architecture for both L1 and L2 caches requires a non-inclusive policy. To justify this, consider the scenario where the L1 cache has four ways and the L2 cache has only two ways. If there are four entries that conflict with each other and always map to the same set, the L1 cache can store all of them simultaneously, whereas the L2 cache cannot.

A cache bank selector is responsible for correctly remapping the address based on the desired cache parameters. Its design maintains a low level of complexity, ensuring it does not affect the critical path of standard cache requests.

For power efficiency, each bank can be turned off or placed in a 'drowsy' state, reducing its frequency to save power when the application's current phase does not require extensive cache capacity to maintain performance. This mechanism enables a reduction in power consumption without significantly sacrificing performance.

Additionally, all cache lines include extra bits: a Correct bit that helps preserve coherence across the cache hierarchy, ten bits to track the ratio of conflict misses, and two bits to monitor how frequently each line has been accessed recently.

3.13 Runtime Procedure

The cache parameters for each layer of the final version of the proposed architecture are listed in Table 3.3. As expected, each cache has a fixed maximum size and will always occupy the area required for the largest possible configuration, even if some resources are not utilized to their full potential.

L1 Total Size	64kB
L1 Size Options	[16, 32, 64]kB
L1 Associativity Options	[1, 2, 4] Ways
L1 Line Size	
L2 Total Size	512kB
L2 Size Options	[128, 256, 512]kB
L2 Associativity Options	[2, 4, 8] Ways
L2 Line Size	64 Bytes

Table 3.3: L1 and L2 caches parameters.

An example of how the runtime adaptable cache pipeline works compared to a static one is illustrated in Figure 3.12. With the simple design of the proposed architecture additional operations are made in parallel with the already standard ones, but no bottleneck is created and, therefore, the critical path and cycles required for a cache request operation remains the same.

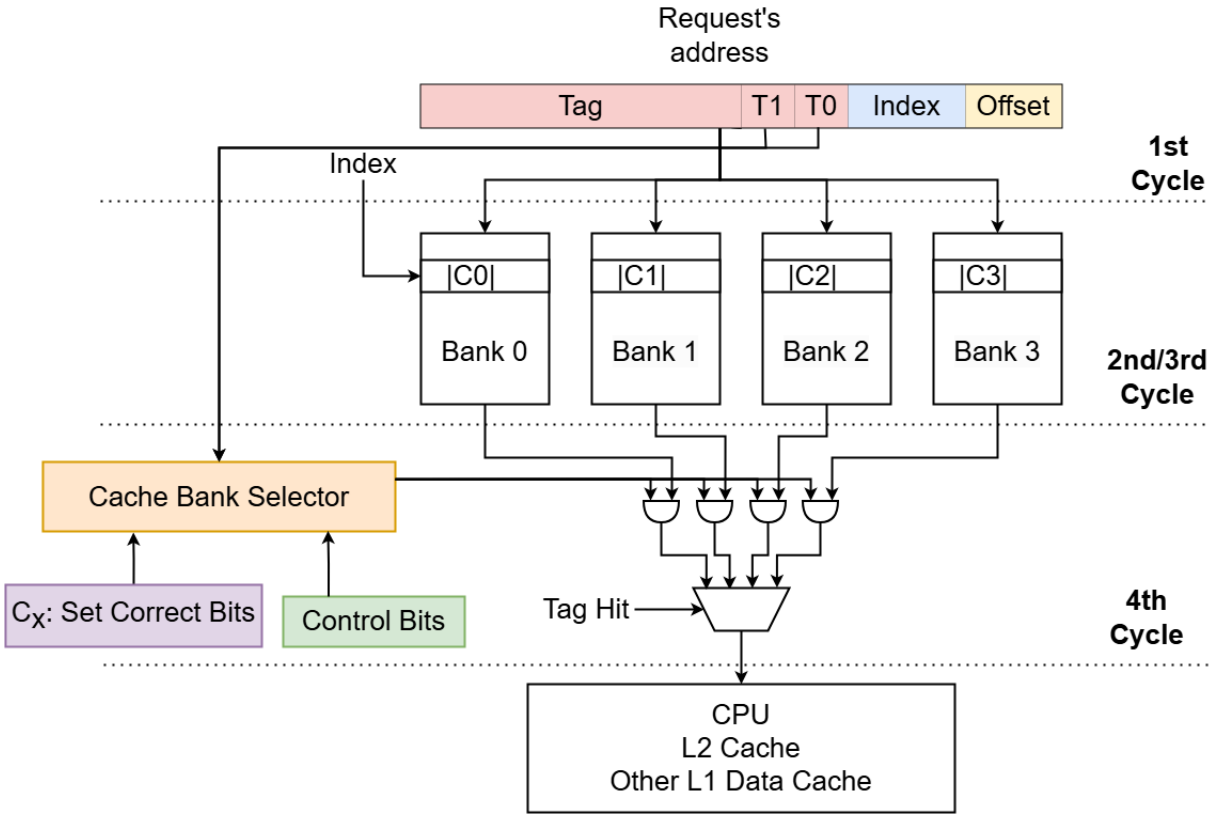


Figure 3.12: Execution pipeline of a Runtime-Adaptable Cache Architecture.

During runtime, the percentages of dead sets and conflict misses are collected to inform decisions on how and when to reconfigure each cache. Algorithm 3 presents pseudo-code illustrating the reconfiguration logic operating behind the scenes. Each statistic has upper and lower limits that can be

independently adjusted. When one of these limits is crossed, a level of confidence is assigned to a corresponding action based on the statistic that crossed the limit. This confidence parameter is recommended to prevent premature reconfigurations.

Algorithm 3 Runtime Reconfiguration Algorithm

```
newCacheAccess()
if numberOfCacheAccesses % 100000 = 0 then
    conflictPercentageUpdate()
    deadSetPercentageUpdate()
    if deadSetPercentage > sizeUpperLimit then
        reduceSizeConfidence ← reduceSizeConfidence + 1
    end if
    if deadSetPercentage < sizeLowerLimit then
        increaseSizeConfidence ← increaseSizeConfidence + 1
    end if
    if conflictPercentage > associativityUpperLimit then
        reduceAssociativityConfidence ← reduceAssociativityConfidence + 1
    end if
    if conflictPercentage < associativityLowerLimit then
        increaseAssociativityConfidence ← increaseAssociativityConfidence + 1
    end if
    if reduceSizeConfidence > confidenceLimit then
        handleCacheReconfiguration(reduceSize)
    else if increaseSizeConfidence > confidenceLimit then
        handleCacheReconfiguration(increaseSize)
    else if reduceAssociativityConfidence > confidenceLimit then
        handleCacheReconfiguration(reduceAssociativity)
    else if increaseAssociativityConfidence > confidenceLimit then
        handleCacheReconfiguration(increaseAssociativity)
    end if
end if
```

While this first iteration of the runtime reconfiguration algorithm is functional, it lacks robustness and generates significant unnecessary overhead. During the implementation phase of this architecture, the disadvantages of this algorithm were identified and addressed through simulations.

Firstly, after a reconfiguration, many lines may end up in incorrect positions. In cases where banks were disabled, this can lead to additional misses. Therefore, it is unreasonable to attempt further reconfigurations immediately, as they may not be based on reliable statistics. To resolve this, a waiting period should be added after each cache state change, allowing the cache time to adjust to the previous changes and ensuring that any future reconfiguration attempts have a higher accuracy.

Secondly, incorporating the current miss rate as an additional variable limit can help prevent undesirable behaviors from the adaptable cache. For instance, if the miss rate is high, reducing cache size could hinder performance, making any potential power savings negligible.

Finally, altering the cache architecture can lead to significant changes in the percentages of conflicts or dead sets, potentially dropping from 60% to nearly 0%. While this effect is beneficial, it can mislead the cache into reconfiguring when it has already achieved the optimal configuration for the current access pattern. Moreover, the cache lacks a mechanism to verify whether the collected statistics originate from the same phase or from a different phase that the application may have transitioned into.

To facilitate more informed reconfigurations, a list of recent program counters (PCs) is maintained

in a table and compared with the PCs collected before the last reconfiguration. If they share entries beyond a certain threshold, the cache assumes the application remains in the same phase and refrains from reverting in its reconfiguration path. Through simulation, it was determined that saving 32 PCs is sufficient to accurately predict phase changes.

The complete and final version of the runtime reconfiguration algorithm is outlined in Algorithm 4, addressing all the shortcomings of the initial algorithm.

3.14 Summary

This chapter presents a runtime-adaptable cache architecture that incorporates both automatic and manual reconfiguration capabilities, along with a novel coherence preservation mechanism. The architecture evolves from a basic static cache structure, gradually integrating components based on state-of-the-art implementations to achieve the proposed objectives.

Algorithm 4 Runtime Reconfiguration Algorithm

```
newCacheAccess()
if numberOfCacheAccesses % 100000 = 0 then
    currentMissRateUpdate()
    conflictPercentageUpdate()
    deadSetPercentageUpdate()
    currentPCsUpdate()
    if waitingPeriod > 0 then
        waitingPeriod ← waitingPeriod − 1
    else
        if deadSetPercentage > sizeUpperLimit & currentMissRate < MissRateLimit then
            reduceSizeConfidence ← reduceSizeConfidence + 1
        end if
        if deadSetPercentage < sizeLowerLimit & currentMissRate > MissRateLimit then
            increaseSizeConfidence ← increaseSizeConfidence + 1
        end if
        if conflictPercentage > associativityUpperLimit & currentMissRate < MissRateLimit
then
            reduceAssociativityConfidence ← reduceAssociativityConfidence + 1
        end if
        if conflictPercentage < associativityLowerLimit & currentMissRate > MissRateLimit
then
            increaseAssociativityConfidence ← increaseAssociativityConfidence + 1
        end if
        if reduceSizeConfidence > confidenceLimit then
            if currentPCs != oldPCs then
                handleCacheReconfiguration(reduceSize)
                oldPCs ← currentPCs()
                resetWaitingPeriod()
            end if
        else if increaseSizeConfidence > confidenceLimit then
            if currentPCs != oldPCs then
                handleCacheReconfiguration(increaseSize)
                oldPCs ← currentPCs()
                resetWaitingPeriod()
            end if
        else if reduceAssociativityConfidence > confidenceLimit then
            if currentPCs != oldPCs then
                handleCacheReconfiguration(reduceAssociativity)
                oldPCs ← currentPCs()
                resetWaitingPeriod()
            end if
        else if increaseAssociativityConfidence > confidenceLimit then
            if currentPCs != oldPCs then
                handleCacheReconfiguration(increaseAssociativity)
                oldPCs ← currentPCs()
                resetWaitingPeriod()
            end if
        end if
    end if
end if
```

Chapter 4

Evaluation

4.1 Simulator

Computer architecture simulation tools are essential for implementing and evaluating new ideas [50]. Several simulators have been used to model novel computer architectures, of which Sniper [51] and gem5 [52] stand out.

Sniper is a state-of-the-art parallel, high-speed x86 simulator. This multi-core simulator allows fast and accurate simulation, trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures.

The Sniper simulator allows one to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications at a high speed when compared to existing simulators. Sniper has been validated against multi-socket Intel Core2 and Nehalem systems and provides average performance prediction errors within 25% at a simulation speed of up to several million instructions per second.

The gem5 simulator offers a highly configurable simulation framework, supporting a wide range of ISAs (Instruction Set Architectures), CPU models, and a flexible memory system. This simulator was chosen to conduct preliminary experiments essential for shaping the direction of this thesis, largely due to its adaptability and the availability of extensive libraries tailored for cache memory systems. It enables the simulation of applications on custom, predefined hardware architectures and facilitates the collection of detailed statistics relevant to performance analysis.

Gem5 operates in two modes: full system mode, which provides a complete simulation of an operating system, and syscall emulation mode, which is optimized for simulating single applications. This dual-mode functionality allows for versatile experimentation, catering to both comprehensive system-level studies and focused application-level investigations.

By modifying the gem5 source code, it is possible to customize and introduce new statistics. For example, instead of relying solely on an overall eviction count, one can implement a custom statistic to track the number of evictions for specific cache sets and ways. Another crucial custom statistic is the number of accesses for each cache configuration when active. These newly added statistics offer a

more granular understanding of cache behavior, enabling a more precise determination of the optimal cache configuration for specific instruction blocks.

4.1.1 Simulation Methodology

The gem5 simulator was selected to implement the proposed runtime-adaptable cache architecture, primarily because of its versatility and flexibility, as previously discussed. Given the numerous tests required to debug the system and validate the performance of each implementation at every stage of development, adopting a well-structured methodology was essential for an efficient step-by-step implementation process.

In gem5, the behavior of every system component is defined across various C++ files. For this thesis, the most relevant files are BaseCache, Tags, and IndexingPolicy, as they play a critical role in defining how each cache request is managed and processed. These files govern key operations, such as cache data storage, retrieval, eviction victim selection, and indexing mechanisms, which are vital to implementing the adaptable cache architecture.

Additionally, these behaviors are configured and linked through Python scripts, allowing for dynamic adjustments based on default settings or specific parameter input. Among these scripts, *caches.py* is particularly important, as it configures the initialization parameters for each cache, while *two_level.py* outlines how the entire simulation is programmed, managing the interactions between different cache levels and the overall simulation flow. The integration of these scripts enables a modular and adaptable simulation environment, which is essential for accurately testing and refining the proposed cache architecture.

The development process was mainly divided into two main phases: implementation and adjustment. During the implementation phase, small, generalized benchmarks were created to test the accuracy of the cache functionalities and to debug each new feature as it was added. This step-by-step approach ensured that each component of the adaptable cache worked properly before moving on to more complex evaluations.

In the adjustment phase, larger and more demanding benchmarks were employed to refine the cache reconfiguration algorithm. These benchmarks provided a diverse range of access patterns and workload intensities, which were crucial in determining the optimal parameters for cache reconfiguration. This phase focused on fine-tuning the algorithm to achieve the best balance between performance, power consumption and adaptability, ensuring that the cache could effectively respond to varying workload demands.

In the context of analyzing the performance of an adaptable cache, it is ideal to use an application that exhibits multiple and varied phases with different access patterns. To achieve this, source files from various benchmarks from the PolyBench Suite [53] were combined into a single C script, ensuring that the cache would be challenged to adapt correctly to the demands of each distinct phase. This approach creates a dynamic testing environment in which the cache must continuously reconfigure itself to maintain optimal performance across changing workloads. Furthermore, in the evaluation phase of

this project, the application being executed in the simulation had its phases mixed into twenty different seeds. The results were averaged in order eliminate or at least reduce any bias in parameter adjustment and performance evaluation phases.

Given that only one simulation could be executed at a time, an additional Python script was developed to manage this process efficiently. This script was designed to queue multiple simulations, each initialized with the desired parameters and configurations. After the completion of each simulation, the selected statistics were automatically collected and stored for future analysis. This automation not only streamlined the testing process but also ensured that a consistent and comprehensive set of data was gathered to evaluate the cache's adaptability under varying conditions.

4.2 Evaluation Criteria

Reaching a final architecture and proving that it is faster than other implementations does not always tell the whole truth. The resources utilized, the area necessary, and the power consumption cannot be blindly ignored when performing a complete evaluation of a cache performance. Being an adaptable cache has the benefit of flexibility when weighing the optimization priority.

Modern computing systems execute diverse applications that have significantly different requirements and priorities [2], therefore, in certain scenarios it may be preferable to have a slower but more power efficient memory system or, contrarily, a faster architecture that requires a higher potency. Although the perfect solution would be an all-round better cache, it is not always the case.

The reconfiguration algorithm has different parameters that work as thresholds for when the structure of the cache should change and to what state. By trying different parameter combinations, it is possible to force the cache to go in a specific direction that satisfies any user's demands.

The two main criteria that this evaluation will focus on are overall performance and power consumption.

4.2.1 Performance Evaluation Metrics

Evaluating the performance of a cache over a group of applications or phases is not always trivial. There are a lot of factors that need to be taken into account so that the simulation results can more accurately predict the behavior of the cache if it were implemented on hardware. For instance, a cache can achieve a very low miss rate, but if it comes at the cost of many evictions or high access times, the overall performance is not necessarily better.

The statistic that carries the most weight in this analysis is Cycles per Instruction (CPI), as it serves as a key indicator of overall performance when comparing the execution of the same group of instructions across different architectures. A lower CPI indicates that the processor completes more instructions per cycle, making it a clear measure of which configuration is the fastest and most efficient overall.

Another crucial parameter to consider is the cache miss rate. As previously discussed, a higher miss rate suggests that a greater number of memory accesses require data retrieval from higher levels in the

memory hierarchy. Since memory latency increases exponentially as data is fetched from higher levels (such as L3 cache or main memory), reducing the miss rate is essential for optimizing performance. Minimizing the miss rate ensures faster data access and less time spent waiting for memory responses, directly impacting the overall efficiency of the system.

Finally, the total number of evictions provides insight into the effectiveness of the cache's memory management. Evictions occur when cache lines must be replaced, and this process is both time-consuming and costly in terms of performance. High eviction rates may indicate excessive cache conflicts or insufficient cache capacity, both of which can degrade performance. Therefore, reducing the number of evictions is critical, as it helps avoid unnecessary delays and improves the overall efficiency of the cache system over the course of one or multiple application executions.

4.2.2 Power Consumption Analysis

Power consumption should always be taken into account when dealing with computer technology. A good architecture is also an efficient architecture. Therefore when evaluating the proposed implementation for an adaptable cache, its power consumption also has to be analyzed.

The gem5 simulator was used to model the desired architecture, but unfortunately it does not have a mechanism that can quantify the energy being consumed during runtime. Nonetheless, a different method was used. The analysis was performed by a relative comparison between the power consumption of a static and an adaptable cache.

By collecting the interval of time each bank is enabled a ratio can be achieved by comparing it to a static cache were all the banks are continuously active.

It is important to note that this comparison only applies to the cache's power consumption. The whole system is affected by the behavior of the cache memory, any application can take longer to execute and the energy consumed by the rest of the processor's components will not be taken into account. Furthermore, it is assumed that an inactive bank will always consume less than an active one and that any overhead associated with a state change is negligible when compared to the execution of the whole or many applications.

The following equations define and show the relation between the energy consumption of an adaptable cache and a static one. With Δt representing the total execution time, P_x the potency with X banks enabled, the Δt_x the fraction of the total execution time with X banks enabled, and k the energy overheads associated with each bank state change.

$$E_{\text{static}} = \Delta t * P_4 \tag{4.1}$$

$$E_{\text{adaptable}} = \Delta t_4 * P_4 + \Delta t_2 * P_2 + \Delta t_1 * P_1 + k \tag{4.2}$$

Considering that $P_4 = 2P_2 = 4P_1$, and that k is insignificant when compared to the scale of the whole application execution it is possible to further simplify:

$$E_{\text{adaptable}} = (\Delta t_4 + \frac{\Delta t_2}{2} + \frac{\Delta t_1}{4}) * P_4 \tag{4.3}$$

$$\frac{E_{\text{adaptable}}}{E_{\text{static}}} = \frac{\Delta t_4 + \frac{\Delta t_2}{2} + \frac{\Delta t_1}{4}}{\Delta t} \tag{4.4}$$

When evaluating the L2 cache, the same ratio can be used since the banks are grouped in pairs and each pair is controlled exactly the same ways as the ones in the L1 cache.

4.3 Different Evaluation Environments

The proposed architecture for a runtime adaptable cache architecture has a fairly large level of complexity, for instance both the L1 and L2 caches keep track of their own statistics and can independently be reconfigured. In such reconfiguration either the size or the associativity can be changed, or even both at the same time. Therefore, it is easier to isolate this described behaviour in order to confirm the impact they have on the resulting performance of the executed application.

With this in mind, the evaluation stage will be separated into different environments. The first one will serve as a basis and all the adaptability mechanisms will be enabled, subsequently, one where only the size of the caches can change, another where only the associativity can be altered, another where only the L1 cache is allowed to be adaptable and another where only the L2 cache has its adaptability enabled.

Finally, comparing the results of the adaptable cache with and without a prefetcher can yield some insight in how the two mechanisms work together. A prefetcher tries to preemptively load data from higher memory levels based on predictions about future accesses based on the patterns of previous ones, and no other state-of-the-art adaptable cache implementation mentioned the advantages or disadvantages of having both prefetching and runtime cache reconfiguration enabled at the same time.

4.3.1 Fully Enabled Adaptability

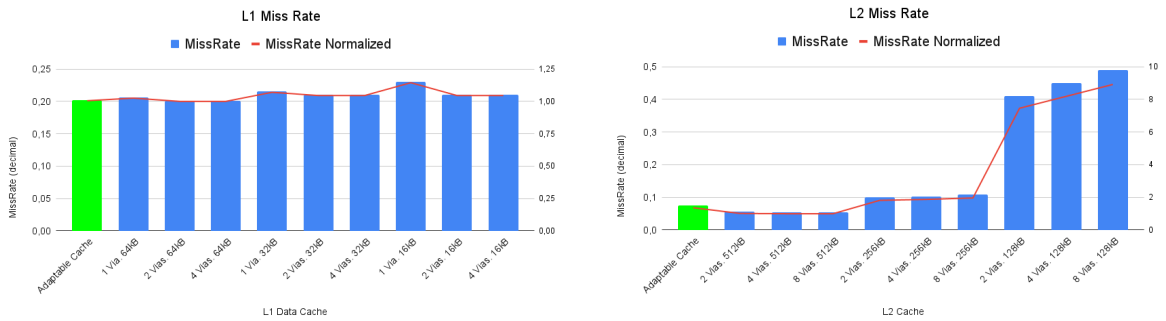
To consolidate all the different mechanisms in a single robust adaptable memory system, this evaluation environment will highlight the overall performance and relative power savings of the adaptable caches will all the its functionalities enabled. Both caches will have its automatic adaptability enabled and be allowed to change their size and associativity as the reconfiguration algorithm demands it.

The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.1, and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results are also represented in Figure 4.1. The comparison of the respective CPIs is shown in Table 4.2 and represented in Figure 4.2.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2, the L1 and L2 adaptable caches consumed, respectively, 10% and 6% less energy when compared to

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,202	1,005	Adaptable	0,075	1,364
1 Via. 64kB	0,206	1,025	2 Vias. 512kB	0,056	1,018
2 Vias. 64kB	0,201	1	4 Vias. 512kB	0,055	1
4 Vias. 64kB	0,201	1	8 Vias. 512kB	0,055	1
1 Via. 32kB	0,215	1,07	2 Vias. 256kB	0,1	1,818
2 Vias. 32kB	0,21	1,045	4 Vias. 256kB	0,103	1,873
4 Vias. 32kB	0,21	1,045	8 Vias. 256kB	0,108	1,964
1 Via. 16kB	0,23	1,144	2 Vias. 128kB	0,41	7,455
2 Vias. 16kB	0,21	1,045	4 Vias. 128kB	0,45	8,182
4 Vias. 16kB	0,21	1,045	8 Vias. 128kB	0,49	8,909

Table 4.1: Miss Rate evaluation of a fully adaptable cache system.



(a) (b)

Figure 4.1: Miss rate evaluation of both L1 and L2 adaptable caches.

the static configuration with maximum size and associativity.

It is clear that, on average, the runtime adaptable cache system can keep up in performance with the static cache configurations with maximum capacity, and is strictly better than any architecture with less than the maximum possible size. Furthermore, by dynamically adapting to any phase's access pattern, the adaptable caches can achieve a reduction in power consumption of up to 10% in case of the L1 and up to 6% in case of the L2. Considering this facts, it is possible to conclude that the proposed architecture for a runtime adaptable cache can achieve equal or greater performance, and consume less power, that any static configuration that would require approximately the same hardware requirements.

The average percentage of execution time each possible cache configuration is active for both the

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	5,86	1,026
1 Via 64kB. 2Vias 256kB	5,79	1,014
2 Vias 64kB. 4Vias 256kB	5,71	1
4 Vias 64kB. 8Vias 256kB	5,71	1
1 Via 32kB. 2Vias 128kB	6,16	1,079
2 Vias 32kB. 4Vias 128kB	6,1	1,068
4 Vias 32kB. 8Vias 128kB	6,08	1,065
1 Via 16kB. 2Vias 64kB	8,38	1,468
2 Vias 16kB. 4Vias 64kB	8,23	1,441
4 Vias 16kB. 8Vias 64kB	8,51	1,49

Table 4.2: CPI evaluation with a fully adaptable cache system.

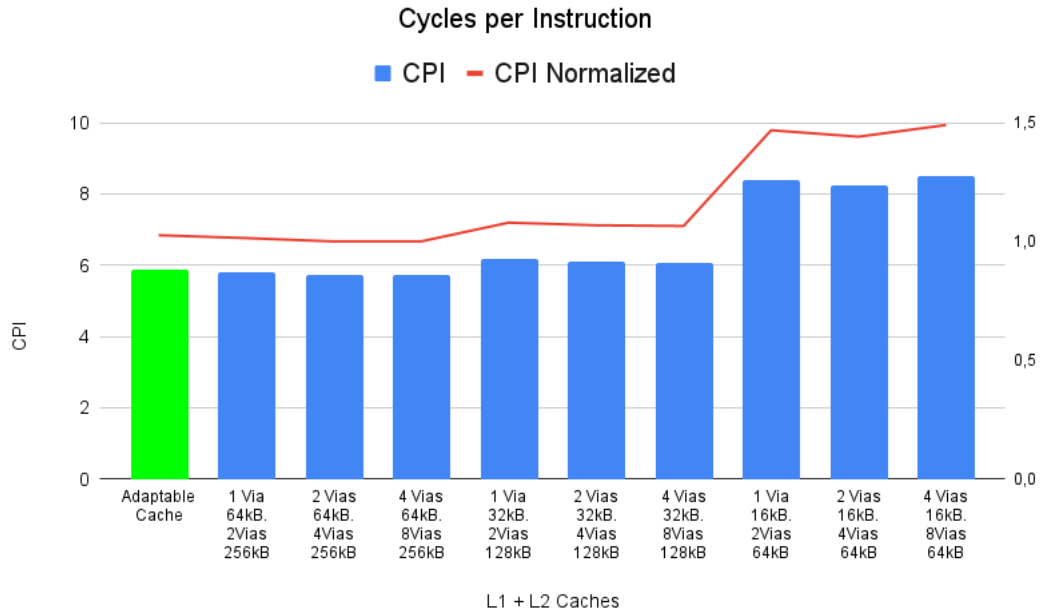


Figure 4.2: CPI evaluation of a fully adaptable cache system.

L1 and L2 caches is represented in Figure 4.3.

Every possible cache configuration has an up-time, for both the L1 and L2 caches, but there is clearly a preference for two configurations in both cases. This bias in not necessarily negative since the benchmarks use for this evaluation can request an unbalanced frequency for each cache state. In contrast, the manual adjustment of reconfiguration parameters can be imperfect, or be biased towards a limited set of adjustment trials. In both scenarios, a larger and more diverse group of benchmarks should be considered in any continuation of this work.

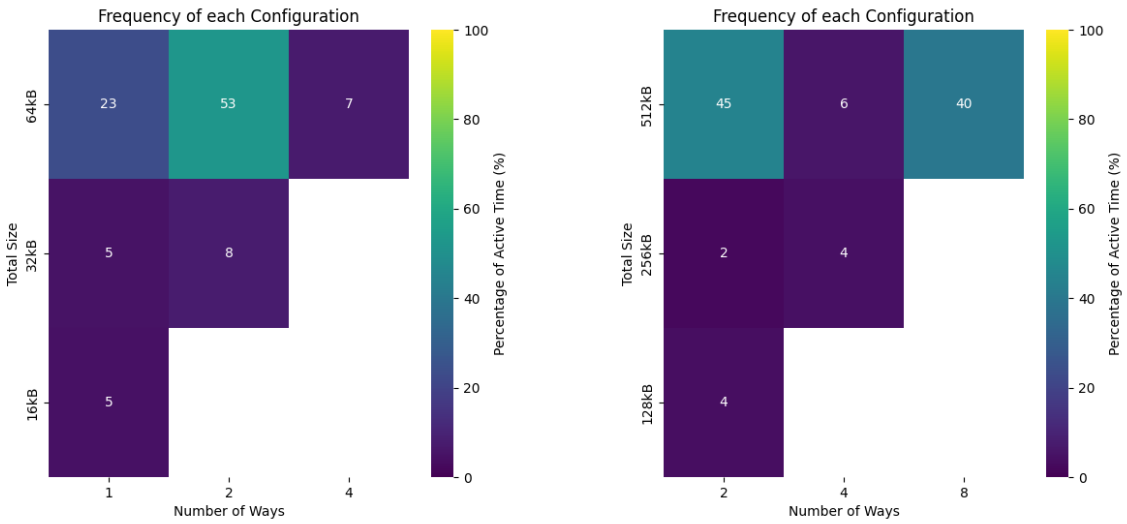


Figure 4.3: State frequency for both adaptable caches.

During runtime the percentage of conflict misses and dead sets is constantly kept tracked and feed to the reconfiguration algorithm so that it can decide if a state change should be made and to what configuration. Since this statistics are directly dependent on the current access pattern and the current cache configuration, it can be useful to visualize an example of how many of each percentage is detected during runtime.

A group of histograms of how frequent each percentage of conflict misses is for each configuration of the L1 cache is represented in Figure 4.4.

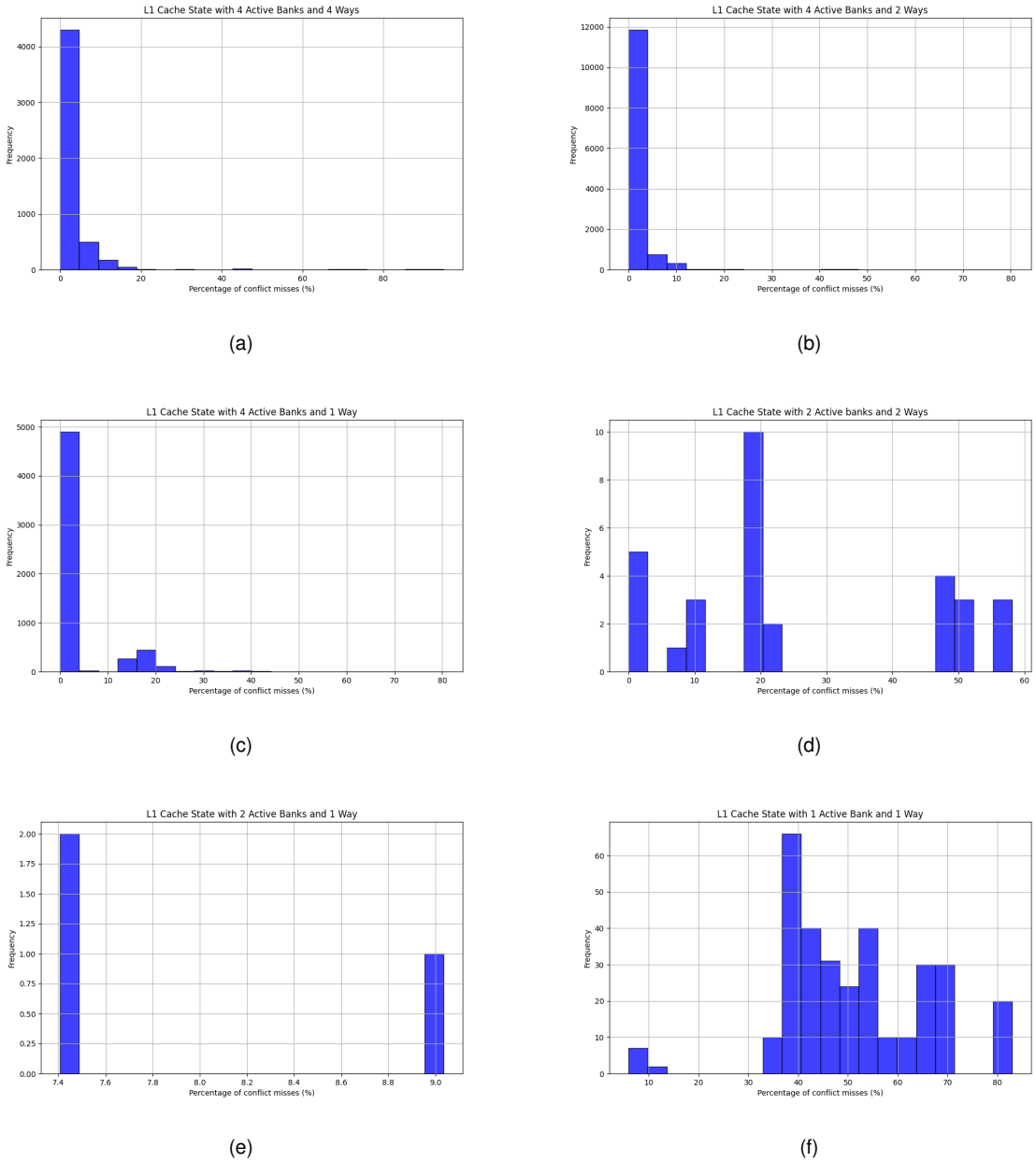


Figure 4.4: Frequency of each percentage of conflict misses collected during runtime for each state of the L1 cache.

A group of histograms of how frequent each percentage of conflict misses is for each configuration of the L2 cache is represented in Figure 4.4.

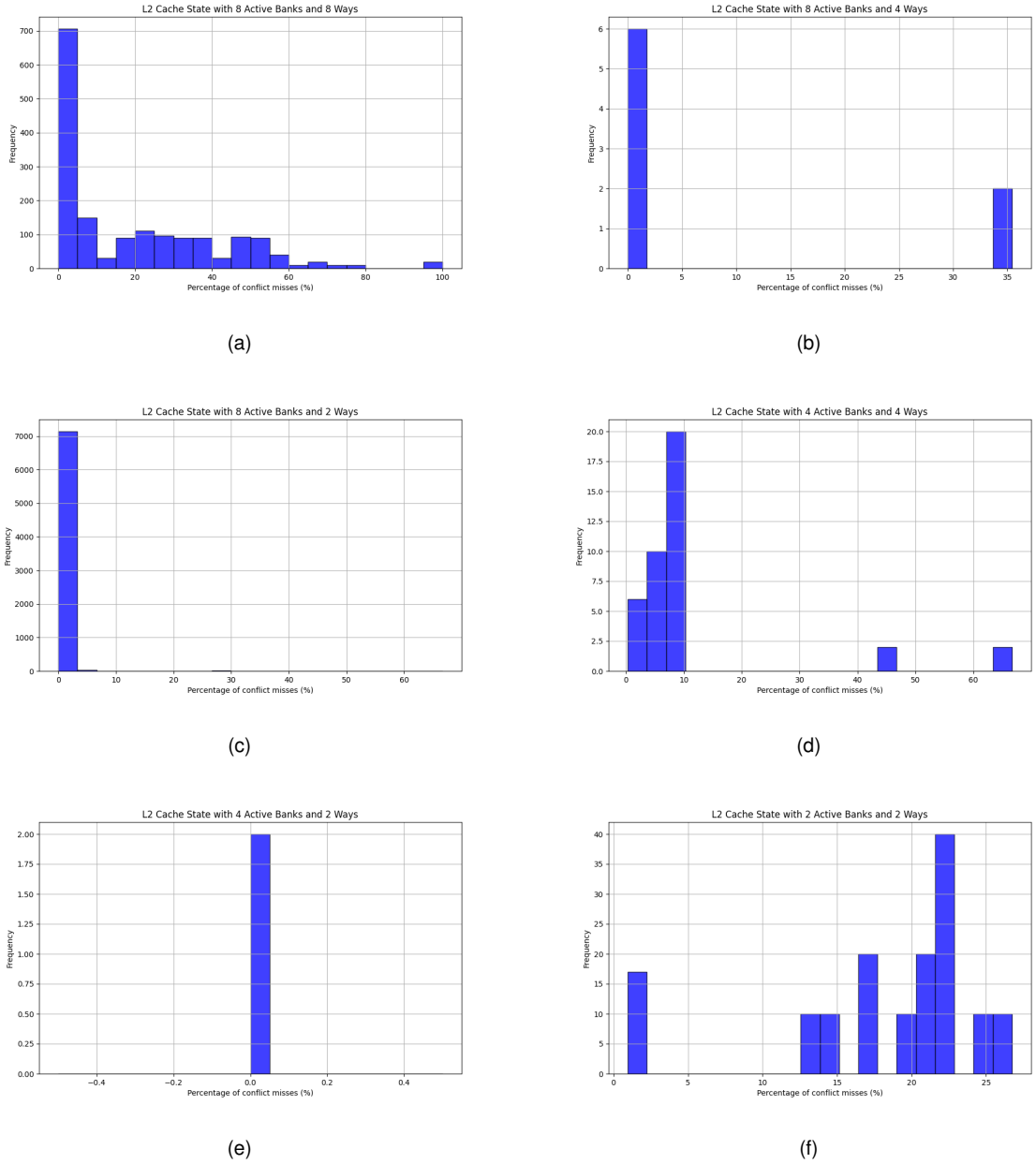


Figure 4.5: Frequency of each percentage of conflict misses collected during runtime for each state of the L2 cache.

It is clear that for a configuration with high associativity, the percentage of misses generated by a conflict are considerably low, but for configurations with a lower number of ways this percentage starts to rise. Ideally, the runtime adaptable cache would try to maintain the lowest associativity possible unless the percentage of conflicts surpassed a desirable level. In that case, it will change to a state with higher associativity if possible for the current number of active banks.

A group of histograms of how frequent each percentage of conflict misses is for each configuration of the L2 cache is represented in Figure 4.4.

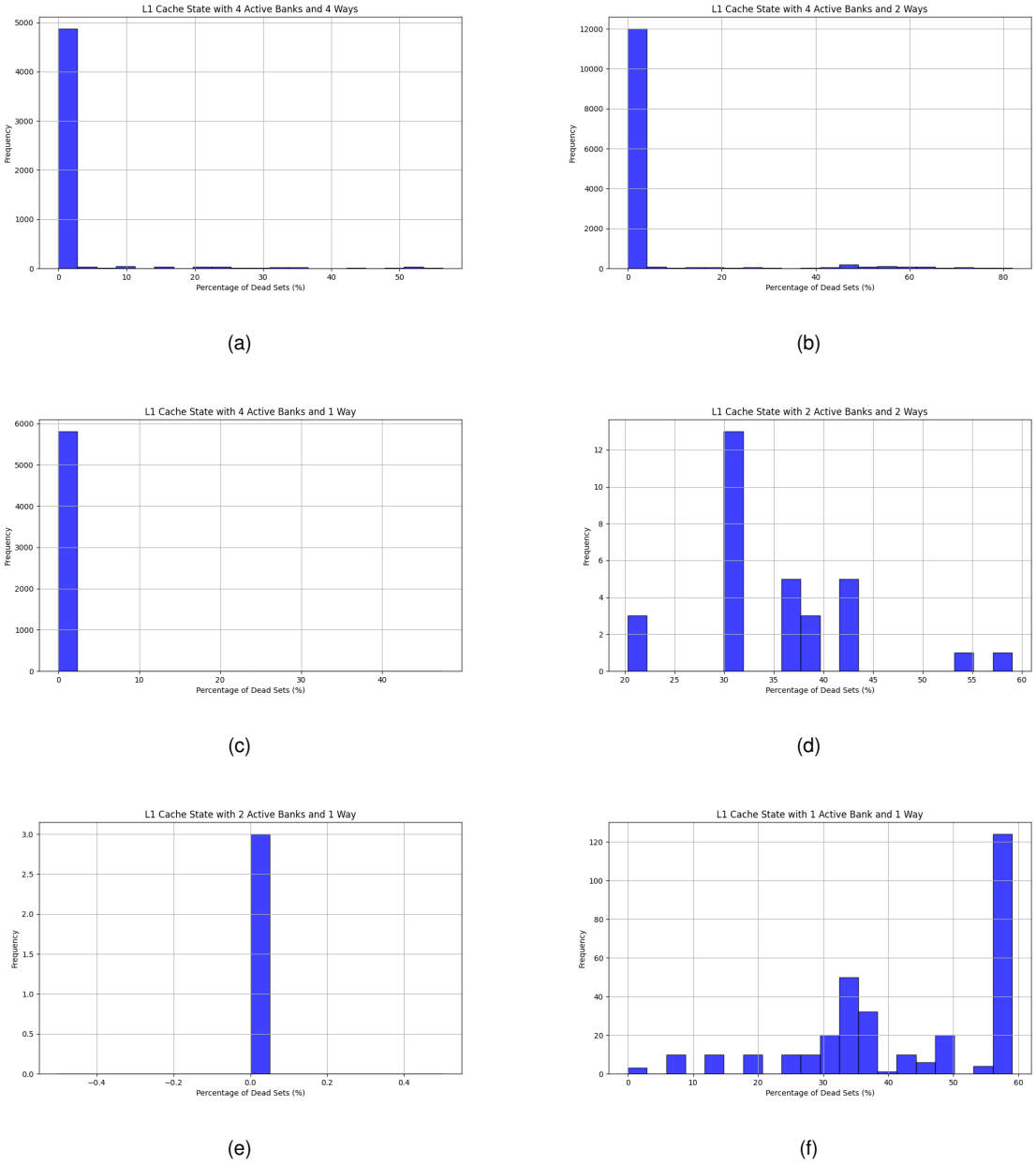


Figure 4.6: Frequency of each percentage of dead sets collected during runtime for each state of the L1 cache.

A group of histograms of how frequent each percentage of dead sets is for each configuration of the L1 cache is represented in Figure 4.4.

If some cache lines are not being used it is possible to save some energy while preserving performance. Whenever the percentage of of dead sets rises, the algorithm knows that the cache can continue to function properly with less active banks and, therefore, transitions into a state of less cache capacity and remains in that configuration unless or until the registered percentage of dead sets is lower than desirable.

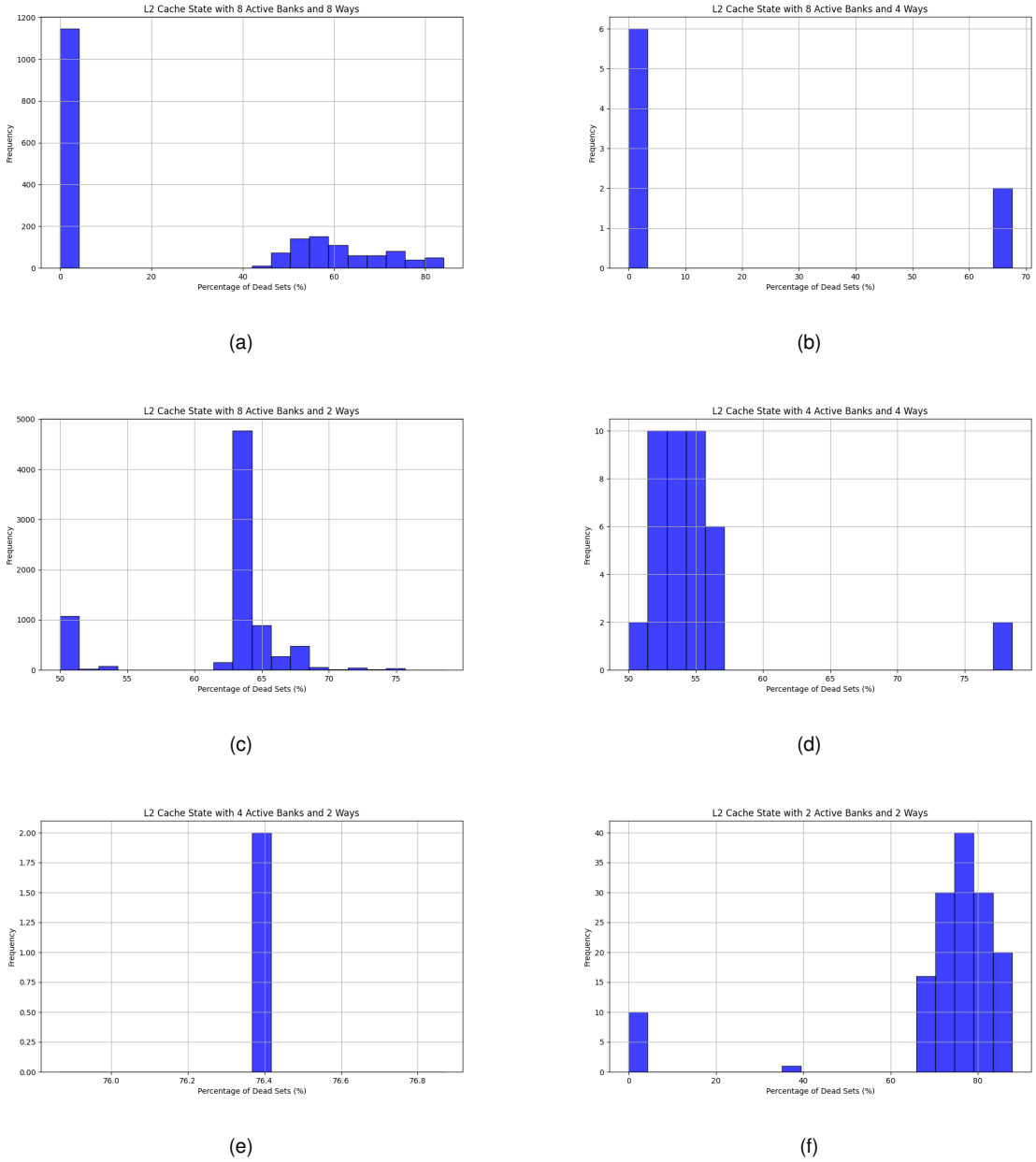


Figure 4.7: Frequency of each percentage of dead sets collected during runtime for each state of the L2 cache.

4.3.2 Fixed Size Evaluation

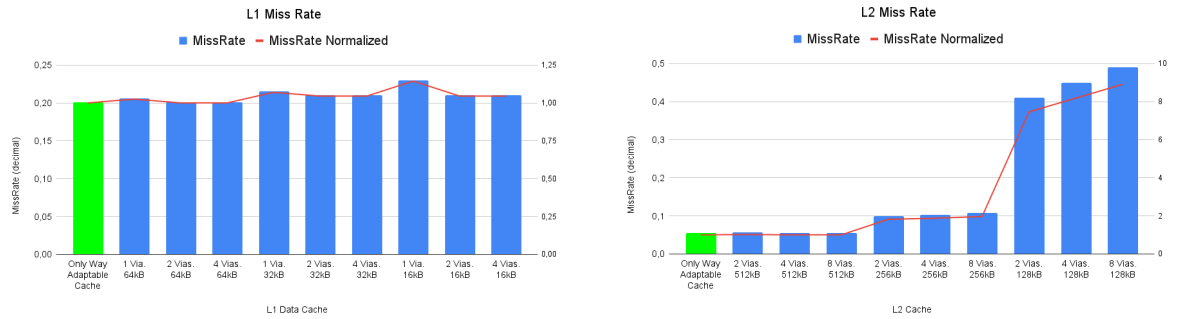
In this environment only the size of the adaptable caches will be allowed to automatically change during runtime. The objective of this evaluation is to access if the reconfiguration algorithm identifies scenarios where changing the available size of the cache can have any benefit either in performance or in power saving.

Due to the structure of the proposed architecture, the total number of available banks is linked to the associativity, therefore in any cache state the associativity will be set to the maximum possible for the current number of enabled banks.

The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.3, and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results are also represented in Figure 4.8. The comparison of the respective CPIs is also shown in Table 4.4 and represented in Figure 4.9.

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,201	1	Adaptable	0,055	1
1 Via. 64kB	0,206	1,025	2 Vias. 512kB	0,056	1,018
2 Vias. 64kB	0,201	1	4 Vias. 512kB	0,055	1
4 Vias. 64kB	0,201	1	8 Vias. 512kB	0,055	1
1 Via. 32kB	0,215	1,07	2 Vias. 256kB	0,1	1,818
2 Vias. 32kB	0,21	1,045	4 Vias. 256kB	0,103	1,873
4 Vias. 32kB	0,21	1,045	8 Vias. 256kB	0,108	1,964
1 Via. 16kB	0,23	1,144	2 Vias. 128kB	0,41	7,455
2 Vias. 16kB	0,21	1,045	4 Vias. 128kB	0,45	8,182
4 Vias. 16kB	0,21	1,045	8 Vias. 128kB	0,49	8,909

Table 4.3: Miss Rate evaluation of a fixed size adaptable cache system.



(a) (b)

Figure 4.8: Miss rate evaluation of both L1 and L2 adaptable caches with fixed size.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2, the L1 and L2 adaptable caches consumed both 0% less energy when compared to the static configuration with maximum size and associativity.

It comes as no surprise that a fixed sized adaptable cache will show no improvements in energy conservation, since all cache banks are completely enabled at all times. Consequently, the permanent large capacity and full associativity flexibility of this scenario allow the adaptable caches to outperform any static cache structure in terms of both miss rate and CPI.

Having an architecture that achieves good performance but has no way of conserve any unnecessary energy is not a good practice, since during certain phases of execution the maximum performance can be achieved without utilizing half of the available resources. Therefore, limiting the adaptable cache to a fixed size should be avoided.

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	5,71	1
1 Via 64kB. 2Vias 256kB	5,79	1,014
2 Vias 64kB. 4Vias 256kB	5,71	1
4 Vias 64kB. 8Vias 256kB	5,71	1
1 Via 32kB. 2Vias 128kB	6,16	1,079
2 Vias 32kB. 4Vias 128kB	6,1	1,068
4 Vias 32kB. 8Vias 128kB	6,08	1,065
1 Via 16kB. 2Vias 64kB	8,38	1,468
2 Vias 16kB. 4Vias 64kB	8,23	1,441
4 Vias 16kB. 8Vias 64kB	8,51	1,49

Table 4.4: CPI evaluation with a fixed size adaptable cache system.

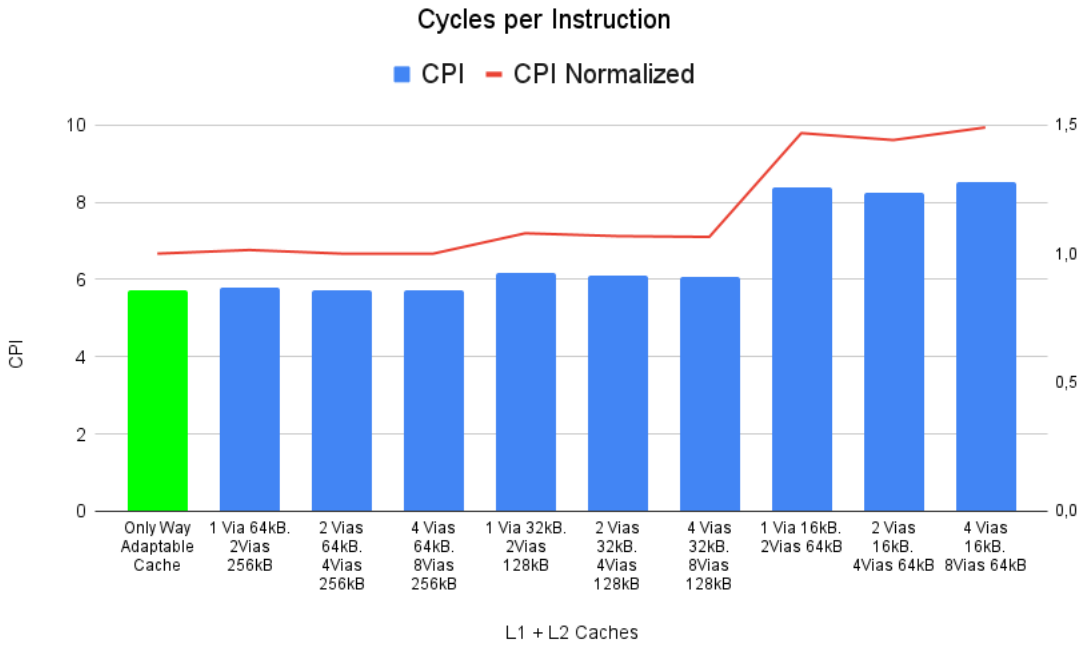


Figure 4.9: CPI evaluation of a fixed size adaptable cache system.

4.3.3 Fixed Associativity Evaluation

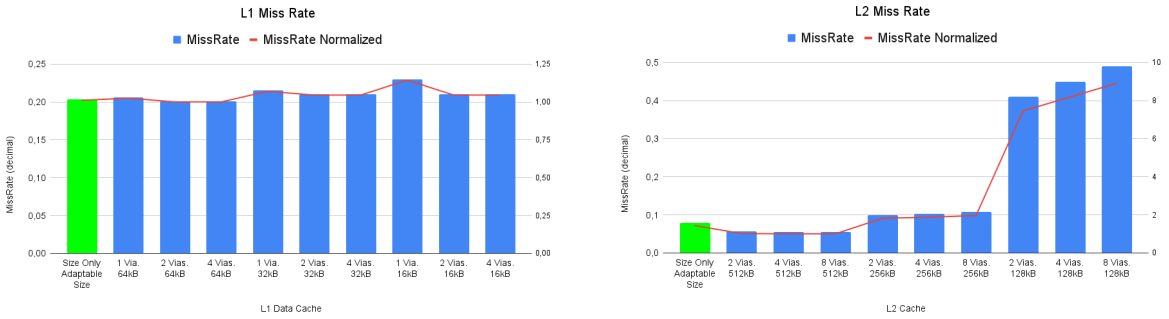
In this environment only the associativity of the adaptable caches will be allowed to automatically change during runtime. The objective of this evaluation is to access if, with the maximum number of lines available, the reconfiguration algorithm identifies scenarios where changing associativity can reduce the number of conflicts or take advantage of a low conflict phase to increase the number of sets and consequently improve performance.

The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.5 and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results represented in Figure 4.10. The comparison of the respective CPIs is shown in Table 4.6 and represented in Figure 4.11.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2,

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,203	1,01	Adaptable	0,079	1,436
1 Via. 64kB	0,206	1,025	2 Vias. 512kB	0,056	1,018
2 Vias. 64kB	0,201	1	4 Vias. 512kB	0,055	1
4 Vias. 64kB	0,201	1	8 Vias. 512kB	0,055	1
1 Via. 32kB	0,215	1,07	2 Vias. 256kB	0,1	1,818
2 Vias. 32kB	0,21	1,045	4 Vias. 256kB	0,103	1,873
4 Vias. 32kB	0,21	1,045	8 Vias. 256kB	0,108	1,964
1 Via. 16kB	0,23	1,144	2 Vias. 128kB	0,41	7,455
2 Vias. 16kB	0,21	1,045	4 Vias. 128kB	0,45	8,182
4 Vias. 16kB	0,21	1,045	8 Vias. 128kB	0,49	8,909

Table 4.5: Miss Rate evaluation of a fixed associativity adaptable cache system.



(a) (b)

Figure 4.10: Miss rate evaluation of both L1 and L2 adaptable caches with fixed associativity.

the L1 and L2 adaptable caches consumed, respectively, 16% and 12% less energy when compared to the static configuration with maximum size and associativity.

By restraining the associativity of the runtime adaptable cache and only allowing it to change its capacity, the limited flexibility resulted in a slightly worse performance when compared to a static cache with maximum capacity and associativity. In contrast, by letting the reconfiguration algorithm focus on possible capacity adjustments, the L1 cache can save up to 16% of consumed energy and the L2 can save up to 12%. From these results it is possible to confirm the importance of giving priority to dynamically change capacity for an adaptable cache, since with a small margin of inferior performance, a considerable amount of power can be saved.

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	5,9	1,033
1 Via 64kB. 2Vias 256kB	5,79	1,014
2 Vias 64kB. 4Vias 256kB	5,71	1
4 Vias 64kB. 8Vias 256kB	5,71	1
1 Via 32kB. 2Vias 128kB	6,16	1,079
2 Vias 32kB. 4Vias 128kB	6,1	1,068
4 Vias 32kB. 8Vias 128kB	6,08	1,065
1 Via 16kB. 2Vias 64kB	8,38	1,468
2 Vias 16kB. 4Vias 64kB	8,23	1,441
4 Vias 16kB. 8Vias 64kB	8,51	1,49

Table 4.6: CPI evaluation of a fixed associativity adaptable cache system.

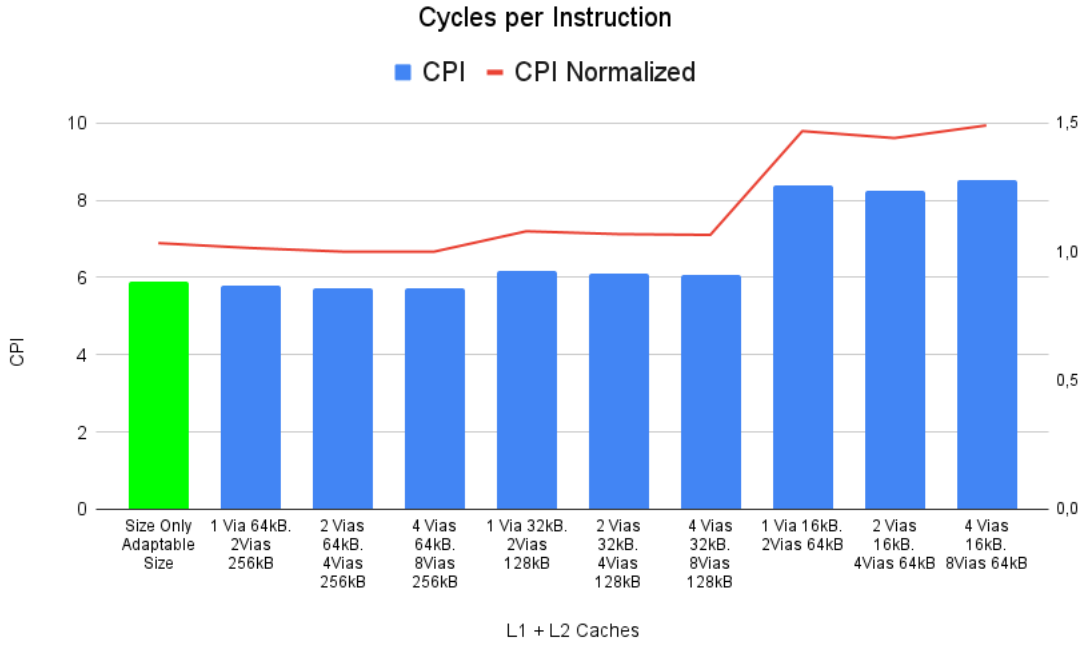


Figure 4.11: CPI evaluation of a fixed associativity adaptable cache system.

4.3.4 Only L1 Enabled Adaptability

In this environment only the L1 cache will have the automatic adaptability enabled. The L2 cache will be set to the maximum size and associativity possible in adaptable mode but, unlike the L1 cache, will be static throughout the whole execution.

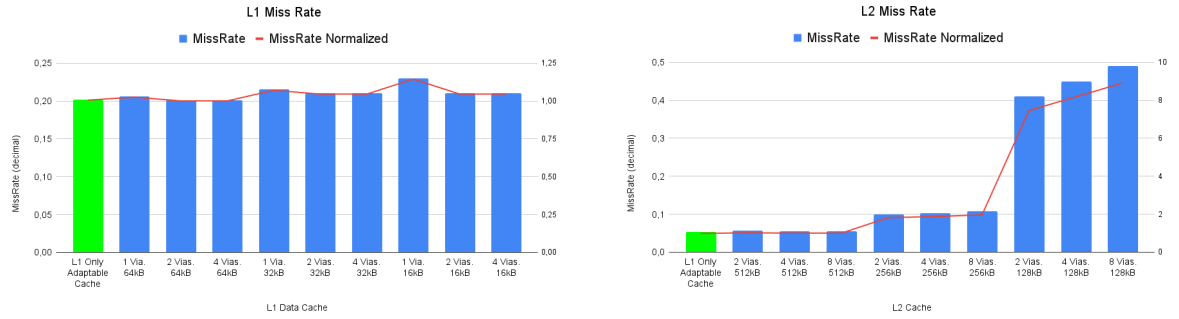
The objective of this evaluation scenario is to verify the impact the L1 cache can have as a runtime adaptable cache without the influence of the L2 cache also changing its structure. Since it is smaller and directly connected to the CPU it is expected to have a higher sensibility to access pattern changes and, therefore, to reconfigure itself more frequently and to more distinct states.

The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.7, and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results are also represented in Figure 4.12. The comparison of the respective CPIs is shown in Table 4.8 and represented in Figure 4.13.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2, the L1 and L2 adaptable caches consumed, respectively, 10% and 0% less energy when compared to the static configuration with maximum size and associativity.

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,202	1,005	Adaptable	0,054	0,982
1 Via. 64kB	0,206	1,025	2 Vias. 512kB	0,056	1,018
2 Vias. 64kB	0,201	1	4 Vias. 512kB	0,055	1
4 Vias. 64kB	0,201	1	8 Vias. 512kB	0,055	1
1 Via. 32kB	0,215	1,07	2 Vias. 256kB	0,1	1,818
2 Vias. 32kB	0,21	1,045	4 Vias. 256kB	0,103	1,873
4 Vias. 32kB	0,21	1,045	8 Vias. 256kB	0,108	1,964
1 Via. 16kB	0,23	1,144	2 Vias. 128kB	0,41	7,455
2 Vias. 16kB	0,21	1,045	4 Vias. 128kB	0,45	8,182
4 Vias. 16kB	0,21	1,045	8 Vias. 128kB	0,49	8,909

Table 4.7: Miss Rate evaluation of a L1 adaptable and L2 static caches.



(a) (b)

Figure 4.12: Miss rate evaluation of L1 adaptable and L2 static caches.

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	5,72	1,002
1 Via 64kB. 2Vias 256kB	5,79	1,014
2 Vias 64kB. 4Vias 256kB	5,71	1
4 Vias 64kB. 8Vias 256kB	5,71	1
1 Via 32kB. 2Vias 128kB	6,16	1,079
2 Vias 32kB. 4Vias 128kB	6,1	1,068
4 Vias 32kB. 8Vias 128kB	6,08	1,065
1 Via 16kB. 2Vias 64kB	8,38	1,468
2 Vias 16kB. 4Vias 64kB	8,23	1,441
4 Vias 16kB. 8Vias 64kB	8,51	1,49

Table 4.8: CPI evaluation with a L1 adaptable and L2 static caches.

It is clear that only having the L1 cache working as a runtime adaptable cache is enough to achieve an advantage over a fully static memory system. The overall performance with an adaptable L1 cache is essentially even with the performance of a completely static system but there are still instances where the L1 cache dynamically reduces its capacity, and therefore the resources being utilized, and saves up to 10% more energy than the L1 static cache.

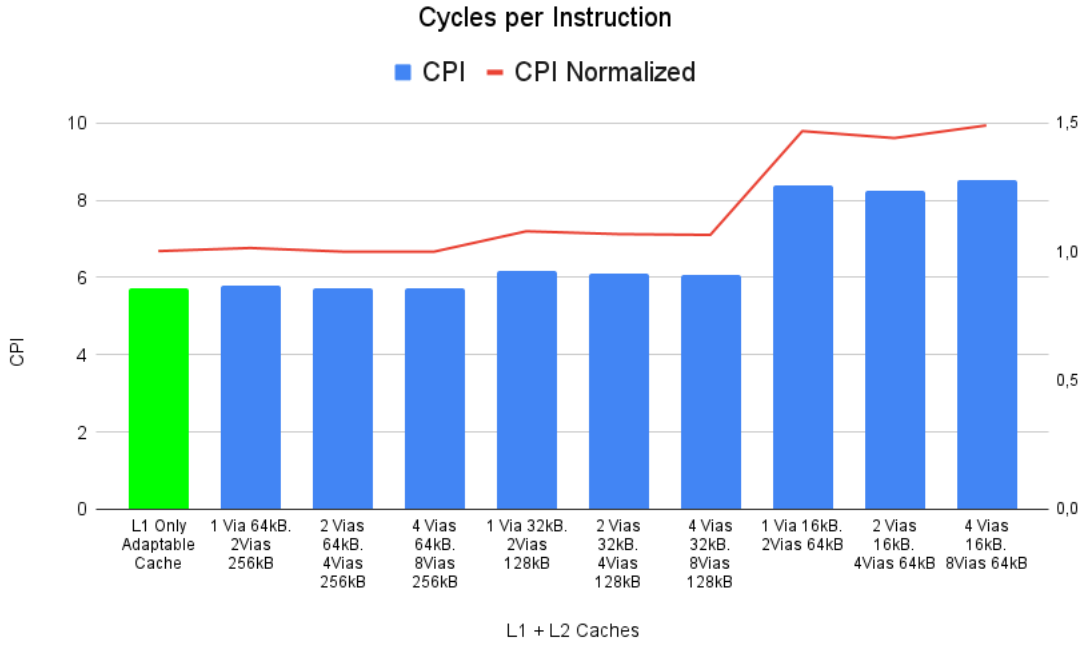


Figure 4.13: CPI evaluation of a memory system with an adaptable L1 and static L2 caches.

The L2 adaptable cache works exactly the same as the L2 static cache and, therefore achieves the same performance but does not economize energy or resources in any way.

4.3.5 Only L2 Enabled Adaptability

In this environment only the L2 cache will have the automatic adaptability enabled. The L1 cache will be set to the maximum size and associativity possible in adaptable mode but, unlike the L2 cache, will be static throughout the whole execution.

The objective of this evaluation scenario is to verify the impact the L2 cache can have as a runtime adaptable cache without the influence of the L1 cache also changing its structure. Since it is larger and unified, containing both data and instructions, each state change is expected to have a larger impact over the total performance of the application being executed.

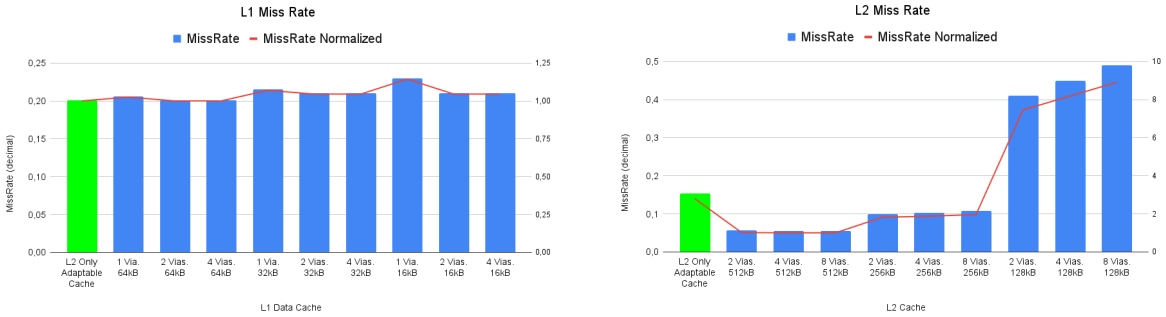
The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.9 and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results are also represented in Figure 4.14. The comparison of the respective CPIs is shown in Table 4.10 and represented in Figure 4.15.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2, the L1 and L2 adaptable caches consumed, respectively, 0% and 13% less energy when compared to the static configuration with maximum size and associativity.

It is clear that only having the L2 cache working as a runtime adaptable cache hinders the performance of the whole memory system by more than a desirable amount when compared to the completely

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,201	1	Adaptable	0,154	2,8
1 Via. 64kB	0,206	1,025	2 Vias. 512kB	0,056	1,018
2 Vias. 64kB	0,201	1	4 Vias. 512kB	0,055	1
4 Vias. 64kB	0,201	1	8 Vias. 512kB	0,055	1
1 Via. 32kB	0,215	1,07	2 Vias. 256kB	0,1	1,818
2 Vias. 32kB	0,21	1,045	4 Vias. 256kB	0,103	1,873
4 Vias. 32kB	0,21	1,045	8 Vias. 256kB	0,108	1,964
1 Via. 16kB	0,23	1,144	2 Vias. 128kB	0,41	7,455
2 Vias. 16kB	0,21	1,045	4 Vias. 128kB	0,45	8,182
4 Vias. 16kB	0,21	1,045	8 Vias. 128kB	0,49	8,909

Table 4.9: Miss Rate evaluation of a L1 static and L2 adaptable caches.



(a) (b)

Figure 4.14: Miss rate evaluation of L1 static and L2 adaptable caches.

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	6,2	1,086
1 Via 64kB. 2Vias 256kB	5,79	1,014
2 Vias 64kB. 4Vias 256kB	5,71	1
4 Vias 64kB. 8Vias 256kB	5,71	1
1 Via 32kB. 2Vias 128kB	6,16	1,079
2 Vias 32kB. 4Vias 128kB	6,1	1,068
4 Vias 32kB. 8Vias 128kB	6,08	1,065
1 Via 16kB. 2Vias 64kB	8,38	1,468
2 Vias 16kB. 4Vias 64kB	8,23	1,441
4 Vias 16kB. 8Vias 64kB	8,51	1,49

Table 4.10: CPI evaluation with a L1 static and L2 adaptable caches.

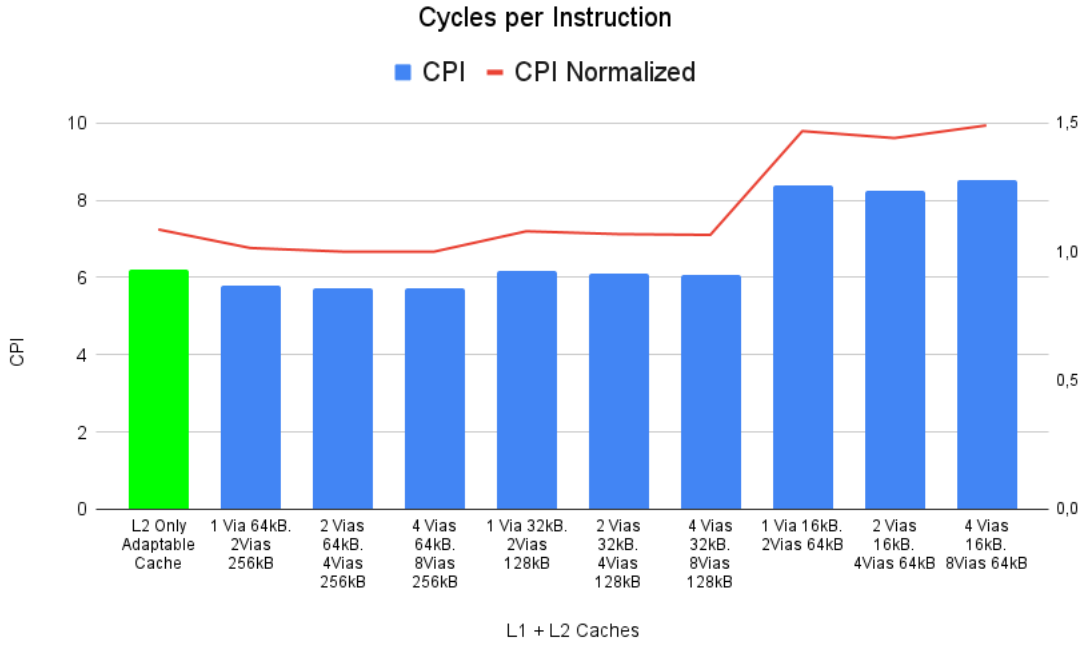


Figure 4.15: CPI evaluation of a memory system with a static L1 and an adaptable L2 cache.

static cache system, but a positive of this configuration is the 13% saving in power consumption of the L2 adaptable cache.

The L1 adaptable cache works exactly the same as the L1 static cache and, therefore achieves the same performance but does not economize energy or resources in any way.

4.3.6 Fully Enabled Adaptability With Prefetching

As previously stated, a scenario where prefetching is enabled can also be productive as it may complement the effort the adaptable cache makes in preventing accesses to higher memory levels as much as possible.

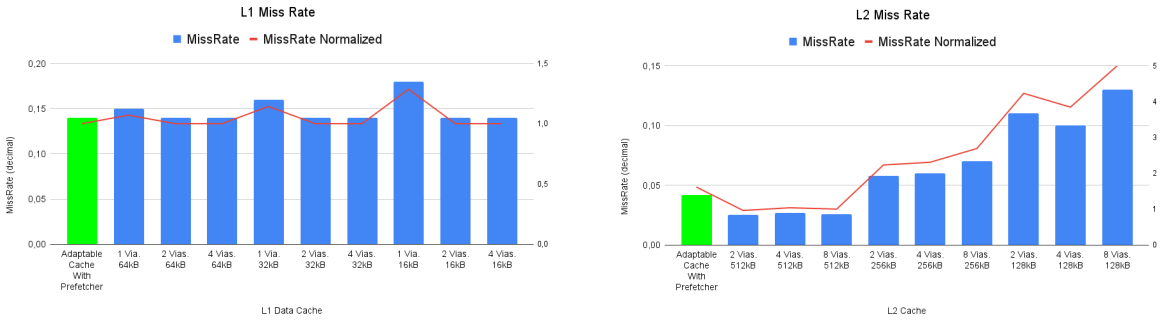
In this environment, two different prefetchers were selected from the ones available in the gem5 simulator, one for the L1 and another for the L2. The ones that showed the higher coverage and accuracy were the Signature Path Prefetcher V2 [54] for the L1 and the Signature Path Prefetcher [55] for the L2. The adaptable cache system and all the static cache configurations were evaluated with this prefetcher enabled as to create a fair comparison between performances.

The miss rates of both L1 and L2 adaptable caches are compared with the miss rate of every combination for a static cache as shown in Table 4.11 and normalized to the largest static configuration possible, 64kB with four ways to the L1 cache and 512kB with eight ways to the L2. This results are also represented in Figure 4.16. The comparison of the respective CPIs is shown in Table 4.12 and represented in Figure 4.17.

From the relative power consumption comparison method described in Equation 4.4, in Section 4.2.2, the L1 and L2 adaptable caches consumed, respectively, 7% and 8% less energy when compared to

L1 Cache	MissRate	MissRate Normalized	L2 Cache	MissRate	MissRate Normalized
Adaptable	0,14	1	Adaptable	0,042	1,615
1 Via. 64kB	0,15	1,071	2 Vias. 512kB	0,025	0,962
2 Vias. 64kB	0,14	1	4 Vias. 512kB	0,027	1,038
4 Vias. 64kB	0,14	1	8 Vias. 512kB	0,026	1
1 Via. 32kB	0,16	1,143	2 Vias. 256kB	0,058	2,231
2 Vias. 32kB	0,14	1,	4 Vias. 256kB	0,06	2,308
4 Vias. 32kB	0,14	1	8 Vias. 256kB	0,07	2,692
1 Via. 16kB	0,18	1,286	2 Vias. 128kB	0,11	4,231
2 Vias. 16kB	0,14	1	4 Vias. 128kB	0,1	3,864
4 Vias. 16kB	0,14	1	8 Vias. 128kB	0,13	5

Table 4.11: Miss Rate evaluation of L1 and L2 adaptable caches with prefetching enabled.



(a) (b)

Figure 4.16: Miss rate evaluation of both L1 and L2 adaptable caches with prefetching enabled.

L1 + L2 Caches	CPI	CPI Normalized
Adaptable	4,96	1.018
1 Via 64kB. 2Vias 256kB	5,04	1,035
2 Vias 64kB. 4Vias 256kB	4,88	1,002
4 Vias 64kB. 8Vias 256kB	4,87	1
1 Via 32kB. 2Vias 128kB	5,25	1,,078
2 Vias 32kB. 4Vias 128kB	5,03	1,033
4 Vias 32kB. 8Vias 128kB	5,04	1,035
1 Via 16kB. 2Vias 64kB	5,8	1,191
2 Vias 16kB. 4Vias 64kB	5,3	1,088
4 Vias 16kB. 8Vias 64kB	5,37	1,103

Table 4.12: CPI evaluation of L1 and L2 adaptable caches with prefetching enabled.

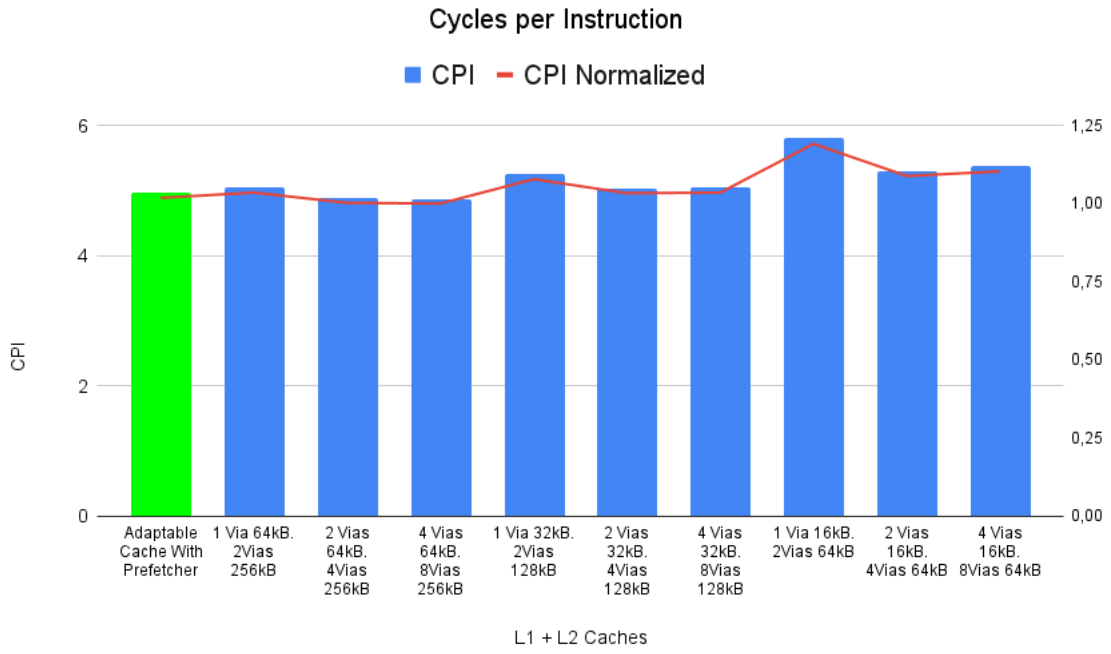


Figure 4.17: CPI evaluation of a fully adaptable cache system with prefetching enabled.

the static configuration with maximum size and associativity.

It is clear that, on average, the runtime adaptable cache system with prefetchers enabled can keep up in performance with the static cache configurations with maximum capacity and associativity. Furthermore, by dynamically adapting to any phase's access pattern, the adaptable caches can achieve a reduction in power consumption of up to 10% in case of the L1 and up to 6% in case of the L2.

Considering this facts, it is possible to conclude that a prefetcher can work in conjunction with a runtime adaptable cache, and both mechanisms together will not only improve or at least maintain a good performance while also identifying instances where energy and resources are not completely required and can be saved to improve overall efficiency.

4.4 Discussion

The architecture proposed in this thesis for a runtime adaptable cache memory system was evaluated using the benchmarks available in the PolyBench Suite. This benchmarks were concatenated into a single multi-phase application, and the average of the execution of twenty different phase orders was used to determine and compare the performance and power consumption of the adaptable cache.

Regardless of the results, it is important to take into account the additional hardware required to implement a runtime adaptable cache architecture. Firstly, each cache line needs to accommodate thirteen additional bits. One bit for the line correctness, two to keep track of dead sets and ten to identify conflict misses.

Continuously, a Cache Bank Selector Circuit is required in each cache in order to manipulate the indexing policy when mapping a request's address to a cache line or set. Finally, four bits of a reserved

register or reserved physical address block need to be allocated for the control bits responsible to identify the current state and configuration of each adaptable cache.

The main take away from the presented results is the flexibility accessible by the adaptable cache caches during runtime, instead of a limited configuration and behaviour oh a static memory system. This capacity to adapt to changes in access patterns during long and varied applications allow the cache to stay near the maximum performance possible for its hardware limitations while also identifying and taking advantage of phases where requests to the memory do not require a lot of lines and cache space. This dynamic behaviour can never be achieved with a static configuration.

Another proposed architectures for runtime dynamic caches like the Smart Cache [4] achieve a reduction of 17% in power consumption also without significantly hindering performance. Unfortunately, there are two factors that make a direct comparison between results unfair.

Firstly, the gem5 simulator [52] does not gather any energy related statistic due to how its simulation framework is created, therefore, the energy saved by the adaptable cache proposed in this thesis was extrapolated by applying a relative comparison to a static cache with all banks permanently enabled. This analysis can also only apply to the differences in potency of adaptable and static caches and to to the whole processor and memory system that are also, of course, dependant on the adaptable cache performance.

Secondly, only the Polybench benchmark suite was used for evaluation instead of other more execution intensive benchmark suites like SPEC CPU [56]. This decision was made since in the time frame this project was developed it was not viable to simulate those benchmarks on gem5, that considerably slows down execution to correctly simulate the applied architecture.

It is clear that the reconfiguration algorithm, although functional, is still not at its maximum potential, not only because it is constrained to only taking actions depending on the percentage of conflict misses and dead sets, but also because the adjustment of parameters was made manually and only based on the results from a single benchmark suite.

4.5 Summary

This section presented a comparison in performance and energy consumption between a runtime adaptable cache memory system and a static one. The evaluation was made with the Polibench Benchmark Suite and some observation were given at the end of each evaluation setup. It concludes with a final discussion about the developed architecture and the resuts it produced.

Chapter 5

Conclusion

This thesis starts by presenting a motivation for the work to be developed and introduces the objectives it will try to achieve in Chapter 1. The quantity and variety of different applications executed in the same machine is increasing. Consequently, various patterns of memory access requests are issued to the cache memory system, that cannot be equally satisfied by a static cache architecture. The concept of adaptability can yield many benefits in terms of performance and power consumption but has the disadvantage of requiring more hardware space and possibly having an additional latency associated with reconfiguration and remapped accesses when compared to conventional static cache memories.

To further support this projects objectives, Chapter 2 starts by explaining what a basis cache structure looks like nowadays and provides some clarification about how it functions and the impact that different design choices can have over its performance. Some previously developed adaptable cache architectures are then enumerated and explained, where the main focus of reconfiguration is the total size of the cache, its associativity and the size of cache lines. All of these parameters have a significant impact on performance and the ability to fit them to an application's needs can be very advantageous. Additionally, some reconfiguration methods are listed that showcase different statistics and priorities of when and how to reconfigure an adaptable cache. Furthermore, different attempts of minimizing flushing and compulsory misses are provided by different authors, for example the order that a reconfiguration path should take.

The focus of this thesis is in Chapter 3, where the proposed architecture for a runtime adaptable cache is introduced. Starting from a basic cache structure, different components are explained and attached to the architecture, each one serving a purpose. It becomes possible to remap a requests address in such a way that functionally the cache can behave as caches with different size and associativity. Furthermore, an extra bit responsible for coherence preservation was introduced alongside extra line access rules. To give the adaptable cache autonomy, an automatic reconfiguration algorithm reliant on runtime statistics was implemented, mainly taking into account the percentage of conflict misses and dead sets.

Finally, Chapter 4 introduces the chosen simulator where the proposed architecture was implemented, the gem5 simulator, and the evaluation method is explained. The kernels for evaluation were

taken from the Polibench benchmark suite and grouped in a multi-phase application. Different evaluation environments were created to test the L1 and L2 adaptable caches under different restrictions. The main evaluation criteria were the miss rate, CPI and relative energy consumption.

From this thesis it is possible to conclude that runtime adaptable cache memory systems should be target of investigation as there is still unexplored automatic reconfiguration mechanisms and algorithm. A robust adaptable cache architecture can bring benefits in controlling the balance between processing and data transferring. The proposed implementation will hopefully serve as a baseline for increasingly sophisticated architectures as it provides a solid and fast method of reconfiguration that can preserve coherence and not generate any additional latency noticed by the processing unit.

5.1 Future Work

The cache architecture proposed for this thesis leaves room for future development. Although the cache reconfiguration mechanism is concise and effective, the algorithm that decides what the correct cache state should be is limited and requires a more methodic adjustment, for example resorting to machine learning algorithms. A more complete method of evaluation, consisting of more varied and complex benchmarks, should also be applied.

Furthermore, to really confirm the effectiveness of the proposed architecture of an adaptable cache in real world scenarios, it will be relevant to implement it in hardware so that not only performance and energy consumption can be accurately measured but also a precise space requirement can be determined.

Bibliography

[1] A. Hankin, T. Shapira, K. Sangaiah, M. Lui, and M. Hempstead, "Evaluation of non-volatile memory based last level cache given modern use case behavior," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 143–154.

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

[3] A. Alsharef, P. Jain, M. Arora, S. R. Zahra, G. Gupta *et al.*, "Cache memory: an analysis on performance issues," in *2021 8th international conference on computing for sustainable global development (INDIACom)*. IEEE, 2021, pp. 184–188.

[4] K. T. Sundararajan, T. M. Jones, and N. Topham, "Smart cache: A self adaptive cache architecture for energy efficiency," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2011, pp. 41–50.

[5] S. Charles, A. Ahmed, U. Y. Ogras, and P. Mishra, "Efficient cache reconfiguration using machine learning in noc-based many-core cmps," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 6, pp. 1–23, 2019.

[6] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[7] J. Handy, *The cache memory book*. Morgan Kaufmann, 1998.

[8] K. R. Kaplan and R. O. Winder, "Cache-based computer systems," *Computer*, vol. 6, no. 3, pp. 30–36, 1973.

[9] S. Lavington, "The atlas story." in *Atlas Symposium*, vol. 6, 2012.

[10] R. Pawson, "The myth of the harvard architecture," *IEEE Annals of the History of Computing*, vol. 44, no. 3, pp. 59–69, 2022.

[11] H. Shukur, S. Zeebaree, R. Zebari, O. Ahmed, L. Haji, and D. Abdulqader, "Cache coherence protocols in distributed systems," *Journal of Applied Science and Technology Trends*, vol. 1, no. 2, pp. 92–97, 2020.

[12] K. Alkhamisi, “Cache coherence issues and solution: A review,” *International Journal of Information Systems and Computer Technologies*, vol. 1, no. 2, 2022.

[13] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A primer on memory consistency and cache coherence*. Springer Nature, 2020.

[14] S. J. Eggers and R. H. Katz, “Evaluating the performance of four snooping cache coherency protocols,” in *Proceedings of the 16th annual international symposium on Computer architecture*, 1989, pp. 2–15.

[15] D. Borodin and B. H. Juurlink, “A low-cost cache coherence verification method for snooping systems,” in *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*. IEEE, 2008, pp. 219–227.

[16] D. Tsaliagos, “Design and implementation of a directory based cache coherence protocol,” 2011.

[17] A. Saraswat, K. Abhishek, H. K. Azad, and S. Shitharth, “Msi-a: an energy efficient approximated cache coherence protocol,” *IEEE Access*, vol. 11, pp. 48 123–48 135, 2023.

[18] R. K. Ibrahim, L. F. Jumma, I. A. Amory, and A. Al-Hilali, “Design of moesi protocol for multicore processors based on fpga,” *International Journal of Nonlinear Analysis and Applications*, vol. 12, no. Special Issue, pp. 1229–1242, 2021.

[19] G. Razavipour, A. Afzali-Kusha, and M. Pedram, “Design and analysis of two low-power sram cell structures,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 17, no. 10, pp. 1551–1555, 2009.

[20] C. Wann, R. Wong, D. J. Frank, R. Mann, S.-B. Ko, P. Croce, D. Lea, D. Hoyniak, Y.-M. Lee, J. Toomey *et al.*, “Sram cell design for stability methodology,” in *IEEE VLSI-TSA International Symposium on VLSI Technology, 2005.(VLSI-TSA-Tech).* IEEE, 2005, pp. 21–22.

[21] M. Meterelliyoz, J. P. Kulkarni, and K. Roy, “Analysis of sram and edram cache memories under spatial temperature variations,” *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 29, no. 1, pp. 2–13, 2009.

[22] J. Lira, C. Molina, D. Brooks, and A. Gonzalez, “Implementing a hybrid sram/edram nuca architecture,” in *2011 18th International Conference on High Performance Computing*. IEEE, 2011, pp. 1–10.

[23] P.-Y. Péneau, D. Novo, F. Bruguier, L. Torres, G. Sassatelli, and A. Gamatié, “Improving the performance of stt-mram llc through enhanced cache replacement policy,” in *Architecture of Computing Systems–ARCS 2018: 31st International Conference, Braunschweig, Germany, April 9–12, 2018, Proceedings 31*. Springer, 2018, pp. 168–180.

[24] S. Mittal, “A survey of recent prefetching techniques for processor caches,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–35, 2016.

[25] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo spatial data prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[26] P. Michaud, “Best-offset hardware prefetching,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

[27] J. Lee, H. Kim, and R. Vuduc, “When prefetching works, when it doesn’t, and why,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, pp. 1–29, 2012.

[28] M. Bakhshalipour, M. Shakerinava, F. Golshan, A. Ansari, P. Lotfi-Karman, and H. Sarbazi-Azad, “A survey on recent hardware data prefetching approaches with an emphasis on servers,” *arXiv preprint arXiv:2009.00715*, 2020.

[29] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

[30] T. M. Chilimbi and M. Hirzel, “Dynamic hot data stream prefetching for general-purpose programs,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 199–209.

[31] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial memory streaming,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 252–263, 2006.

[32] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, “Pythia: A customizable hardware prefetching framework using online reinforcement learning,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1121–1137.

[33] S. Jiang, Q. Yang, and Y. Ci, “Merging similar patterns for hardware prefetching,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1012–1026.

[34] N. S. Kalani and B. Panda, “Instruction criticality based energy-efficient hardware data prefetching,” *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 146–149, 2021.

[35] F. Hameed, L. Bauer, and J. Henkel, “Dynamic cache management in multi-core architectures through run-time adaptation,” in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 485–490.

[36] K. Kuan and T. Adegbija, “Energy-efficient runtime adaptable l1 stt-ram cache design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1328–1339, 2019.

[37] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, and C. Prete, “Way adaptable d-nuca caches,” *International Journal of High Performance Systems Architecture*, vol. 2, no. 3-4, pp. 215–228, 2010.

[38] C. Zhang, F. Vahid, and R. Lysecky, “A self-tuning cache architecture for embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 407–425, 2004.

[39] L. Chen, X. Zou, J. Lei, and Z. Liu, “Dynamically reconfigurable cache for low-power embedded system,” in *Third International Conference on Natural Computation (ICNC)*, vol. 5. Ieee, 2007, pp. 180–184.

[40] D. H. Albonesi, “Selective cache ways: On-demand cache resource allocation,” in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 248–259.

[41] A. Gordon-Ross, F. Vahid, and N. Dutt, “Fast configurable-cache tuning with a unified second-level cache,” in *Proceedings of the 2005 international symposium on Low power electronics and design*, 2005, pp. 323–326.

[42] W. Wang, P. Mishra, and A. Gordon-Ross, “Dynamic cache reconfiguration for soft real-time systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 2, pp. 1–31, 2012.

[43] J. Abella and A. González, “Heterogeneous way-size cache,” in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 239–248.

[44] A. Gordon-Ross and F. Vahid, “A self-tuning configurable cache,” in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 234–237.

[45] W. Wang and P. Mishra, “Dynamic reconfiguration of two-level caches in soft real-time embedded systems,” in *2009 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2009, pp. 145–150.

[46] J. D. Collins and D. M. Tullsen, “Hardware identification of cache conflict misses,” in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 126–135.

[47] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy, “Cache size selection for performance, energy and reliability of time-constrained systems,” in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006, pp. 923–928.

[48] M. D. Hill and A. J. Smith, “Evaluating associativity in cpu caches,” *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[49] E. G. Hallnor and S. K. Reinhardt, “A fully associative software-managed cache design,” *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 107–116, 2000.

[50] H. Brais, R. Kalayappan, and P. R. Panda, “A survey of cache simulators,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–32, 2020.

[51] T. E. Carlson, W. Heirman, and L. Eeckhout, “sniper,” 2013.

[52] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[53] T. Yuki, “Understanding polybench/c 3.2 kernels,” in *International workshop on polyhedral compilation techniques (IMPACT)*, 2014, pp. 1–5.

[54] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, “Path confidence based lookahead prefetching,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[55] A. J. Kim, P. V. Gratz, and A. N. Reddy, “Lookahead prefetching with signature path,” *The 2nd Data Prefetching Championship (DPC2)*, 2015.

[56] J. L. Henning, “Spec cpu2000: Measuring cpu performance in the new millennium,” *Computer*, vol. 33, no. 7, pp. 28–35, 2000.