UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Performance Improvements of an Open-source L2 Cache

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Roche Peris, David

Tutor: Hernández Luz, Carles

Cotutor: López Rodríguez, Pedro Juan

ACADEMIC YEAR: 2023/2024

Resum

L'alt interés industrial en els processadors RISC-V de codi obert han generat la necessitat de nous components més complexos que puguen optimitzar al màxim les prestacions dels processadors. Les caches compartides, generalment l'L2 en processadors embeguts d'altes prestacions, són un dels components més crítics en els Sistemes en un Chip (SoCs). No obstant això, encara que existeixen molts processadors RISC-V multinucli, les implementacions de caches L2 actuals són generalment molt més simples que les presents en processadors comercials.

Aquest projecte presenta les millores d'una cache L2 de codi obert, implementant optimitzacions ben estudiades i presents en productes comercials per millorar les prestacions de la cache. Concretament, es posa enfasi en la implementació de dues millores: particionat basat en vies, i transferències no bloquejants. La cache ha sigut descrita en RTL i prototipada i testejada en una FPGA. Els resultats mostren que aquesta nova cache millora significativament les prestacions i la qualitat de servei (QoS) dels sistemes multinucli.

Paraules clau: Cache compartida, Particionat, Codi obert, Cache no bloquejant

Resumen

El alto interés industrial en los procesadores RISC-V de código abierto han generado la necesidad de nuevos componentes más complejos que puedan optimizar al máximo las prestaciones de los procesadores. Las caches compartidas, generalmente la L2 en procesadores embebidos de altas prestaciones, son uno de los componentes más críticos en los Sistemas en un Chip (SoCs). Sin embargo, aunque existen muchos procesadores RISC-V multinúcleo, las implementaciones de caches L2 actuales son generalmente mucho más simples que las que se encuentran en procesadores comerciales (COTS).

Este proyecto presenta la mejora de una cache L2 de código abierto, implementando optimizaciones bien estudiadas y presentes en productos comerciales para mejorar las prestaciones de la cache. En particular, se pone énfasis en la implementacion de dos mejoras: particionado basado en vias y transferencias no bloqueantes. La cache ha sido descrita en RTL y prototipada y testeada en una FPGA. Los resultados muestran que esta nueva cache mejora significativamente las prestaciones pico y la calidad de servicio (QoS) en sistemas multinúcleo.

Palabras clave: Cache compartida, Particionado, Codigo abierto, Cache no bloqueante

Abstract

The high industrial interest in RISC-V open-source processor implementations has exposed the need for new, more complex open-source supporting components to fully optimize the processors' performance. Shared caches, normally the L2 in high-performance embedded processors, are one of the most critical components in Systems-on-Chip (SoCs). However, despite the existence of many open-source multicore RISC-V processors, current open-source L2 implementations are generally much simpler than those found in Common Off-The-Shelf (COTS) processors.

In this project, we improve an existing open-source L2 cache implementation by applying well-known optimizations widely used in proprietary products to improve the performance of the cache. In particular, we tackle the implementation of two features: way-partitioning and non-blocking transfers. The cache implementation is described in

RTL and prototyped and tested on an FPGA. The results show that our cache modifications significantly improve both peak performance and Quality of Service (QoS) properties of multicore SoCs.

Key words: Shared-cache, Partitioning, Open-source, Non-blocking cache

Contents

List of Figures

List of Tables

List of Acronyms

AXI	Advanced eXtensible Interface
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
COTS	Common Off-The-Shelf (processors)
CSR	Configuration & Status Register
CPU	Central Processing Unit
EDA	Electronic Design Automation
FSM	Finite State Machine
HDL	Hardware Description Language
HPC	High Performance Computing
ISA	Instruction Set Architecture
IP	Intellectual Property
LLC	Last Level of Cache
LRU	Least Recently Used
MSHR	Miss Status Handling Register
RAM	Random Access Memory
RTL	Register-Transfer Level
SoC	System-on-Chip
SRAM	Static RAM
OPT	OPTimal page replacement
QoS	Quality of Service
VHDL	Very high speed integrated circuit Hardware Description Language

CHAPTER 1
Introduction

Traditionally, processor design has been developed in a closed-source environment dominated by a few big companies such as ARM, Intel, AMD, IBM, or Qualcomm. Although some efforts were made to define open-source processor Instruction Set Architectures (ISAs) such as SPARC [2], openRISC [3], and Berkeley's RISC I and RISC II [4], they were not really practically used, mainly due to their reduced support from the industry and academia.

However, in recent years, the open-source software philosophy has reached the hardware industry. The irruption of the RISC-V ISA has promoted the development of a vast amount of new open-source processor implementations and increasingly complex Systems-on-Chip (SoCs) designs. To some extent, RISC-V has fueled the development of not only open-source Central Processing Unit (CPU) implementations but also new supporting open-source Intellectual Property (IP) cores such as memory controllers, co-processors, cache memories, and peripherals; providing essential and extended support for these SoCs.

There is a high availability of open-source RISC-V cores [5–8]. Initial core developments have mostly focused on low-complexity CPU architectures for microcontroller-based applications. However, the fast evolution of the open-source and RISC-V ecosystem is currently aiming at increasingly ambitious use cases such as autonomous driving [9], machine learning [10], and genomics sequence alignment [11]. As more computation power is needed in SoCs, the number of cores is increased, putting a strain on the shared resources of the system, mainly on the shared caches and the interconnect. This is when the limitations of the existing supporting open-source IP cores come to light.

1.1 Motivation

Caches are one of the key elements in complex SoC designs since they facilitate minimizing data movements and relieve the programmer from the burden of explicitly handling data allocation. Additionally, the design of shared caches is also crucial for safety-related applications (e.g., autonomous driving) requiring strict performance guarantees. On the one hand, shared caches are invaluable to ensure multicore performance scalability, but on the other hand, they are also one of the major sources of contention in most multicore SoC designs.

Although much effort has been made to improve cache performance since the irruption of multicore processors in the industry, resulting in many interesting design proposals, the available open-source caches do not offer much more than a basic functionality. Therefore, there is a necessity for an open-source high-performance cache fit for state-of-the-art SoCs that aim to solve complex problems. To do so, the cache must provide good multicore performance and be scalable. Ideally, the cache must also have some sort of QoS mechanism that also makes it suitable for real-time systems.

Additionally, reusability is one of the key advantages of open-source projects over proprietary ones. Extending an existing project instead of creating a module from scratch saves developing time and resources, and leads to better, more complete results. For this reason, most open-source projects, including this one, surge as an extension to previous open designs.

1.2 Objectives

The scope of this project is the creation of an open-source shared-cache implementation fit for high-performance embedded systems, and equipped with fundamental features not available in other open-source implementations. To reach this goal, and based on the necessities outlined in the previous section, the following objectives have been set out:

- The component must be open-source
- The cache must provide performance guarantees, providing some sort of QoS mechanism
- The component must exhibit good performance in multicore systems
- The cache must be scalable

We will extend and modify an open-source cache to achieve these objectives. First, we will survey the current open-source cache designs in the market and choose the most appropriate baseline cache design, Gaisler's L2C-Lite. In this work, this baseline design will be analyzed in-depth and extended with the following main features:

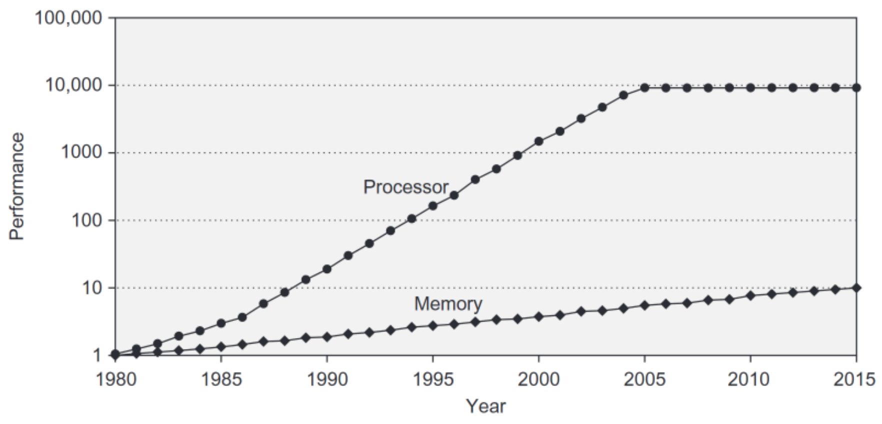
- Modularize the design, adding a standalone frontend component
- Optimize the design proposed by Gaisler, simplifying states and saving clock cycles where possible
- Change the cache implementation to a non-blocking design, more fit for multicore systems
- Implement partitioning to reduce inter-core evictions and provide more determinism when needed

The project's open-source nature must be considered during the design phase, acknowledging that adaptability and ease of use are vital for its success. Thus, our main design considerations were:

- **Modularity:** We aimed to make the design as modular as possible, providing flexibility to maximize its usability in the future.
- **Configurability:** We want to maximize configurability to make it suitable for low and high-end systems.
- **Simplicity:** To maximize the system's possible extendability, each of the modules must be simple and easily understandable.

1.3 Why are caches needed?

Component miniaturization, based on Moore's law, led to huge advancements in both processors and memories [12]. However, while the CPU design focus was to optimize for throughput, the memory design focused on capacity. Thus, the performance gap between memory and processor widened (see Figure 1.1). Even if the end of Moore's law and Dennard's scaling law has caused the stagnation of processors' performance since the mid-2000s [13], the performance gap still exists; thus, solutions to lower the penalty of memory accesses are still needed.



Source: Computer Architecture: A Quantitative Approach, 6th edition [14]

Figure 1.1: Processor and main memory (RAM) performance gap evolution since 1980

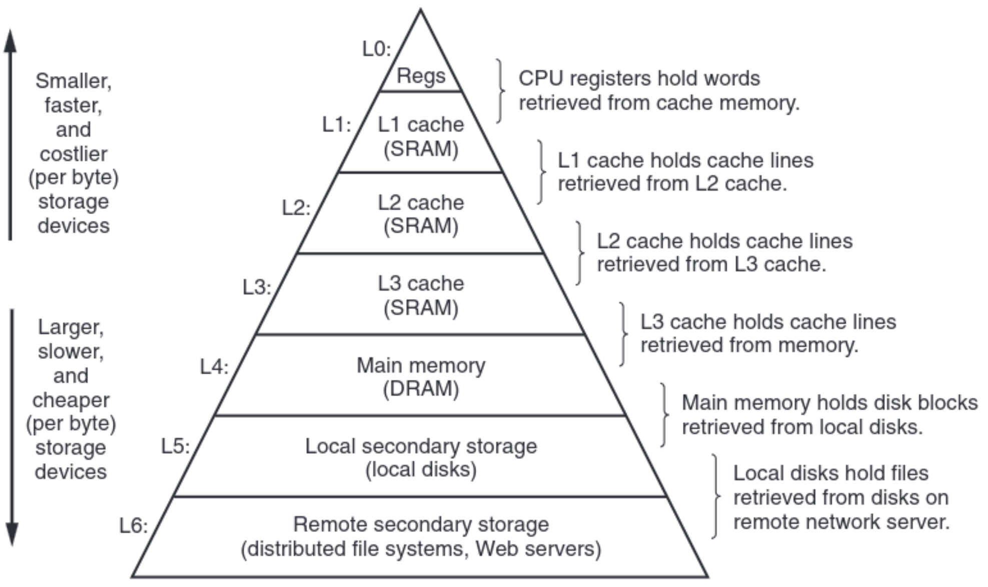
The main property used to bridge the processor-memory gap is the locality principle [15]. This principle states that a computer program tends to reuse data and instructions recently used. This is strongly related to the 90/10 law, which states that, generally, *"90% of the execution time of a computer program is spent executing 10% of the code"* [14] [16]. Two main types of locality are especially relevant:

- **Temporal locality:** A memory location recently accessed is likely to be accessed again in the near future
- **Spatial locality:** Nearby memory locations tend to be referenced closely in time

Taking advantage of the locality principle, the cost-effective alternative to reduce memory access times is implementing a memory hierarchy by combining different memory technologies, trading off speed for capacity. In a usual memory hierarchy (see figure 1.2), the higher levels comprise high-speed, small, volatile, and high-cost (per bit) memories. On the other hand, the lower levels are composed of big, non-volatile, and cheap memories but offer lower speeds. Due to the locality principle, the higher levels are much more likely to be accessed than the lower ones, significantly improving the average memory access times. In most computer architectures, the memory hierarchy is transparent to the programmer, giving the false illusion of a unified big and fast memory.

Caches are the small on-chip SRAM memories that lie (logically) between the processor and main memory. Caches are automatically managed by hardware and are essential to boost the system's performance by providing fast memory access.

In current systems, multiple levels of cache memories, generally two levels in embedded systems and three in COTS processors, are used to optimize performance.


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: Regs
L1: L1 cache (SRAM)
L2: L2 cache (SRAM)
L3: L3 cache (SRAM)
L4: Main memory (DRAM)
L5: Local secondary storage (local disks)
L6: Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network server.

Source: Computer Systems: A Programmer's Perspective [17]

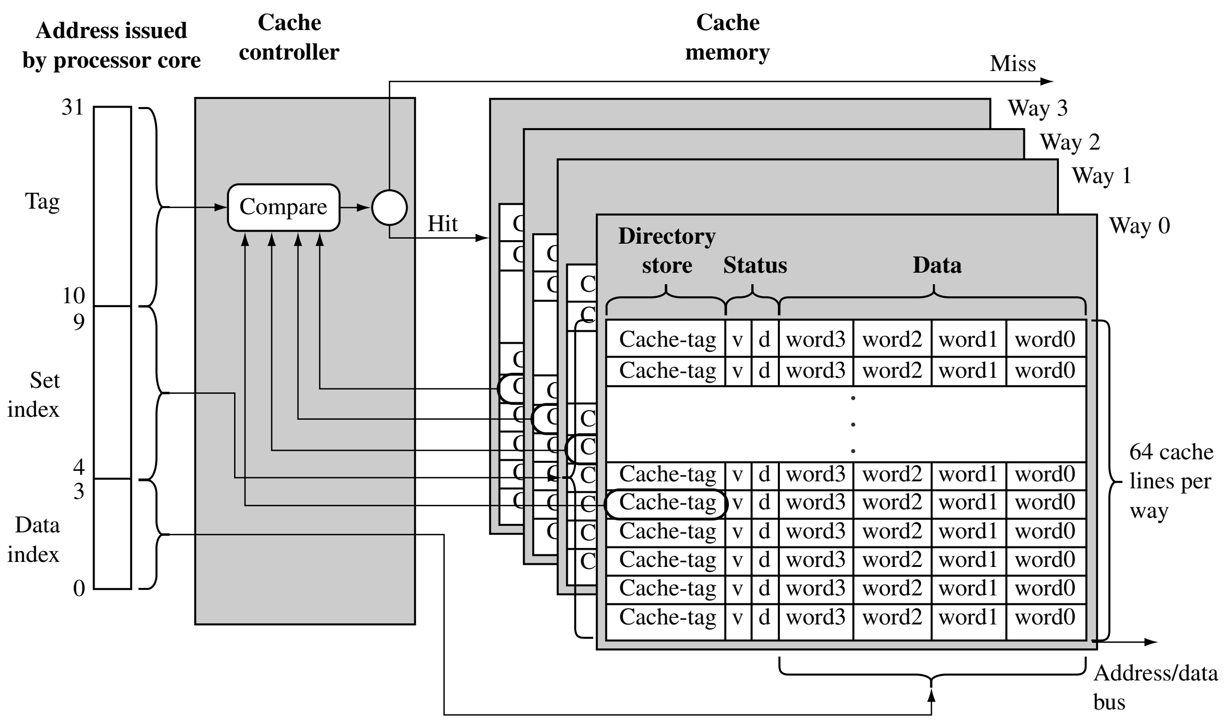
Figure 1.2: An example of a memory hierarchy

1.4 Cache memories

Figure 1.3 shows the structure of a typical cache. In a cache, words are stored in blocks, called *cachelines*, four or eight words long, to fully take advantage of spatial locality. The *data index*, also known as offset, of the address, indicates the word to be accessed inside of a block.

In a memory address, the *set index*, or just *index*, determines the location of a block in the cache. The location of the cache line depends on the mapping scheme: if the set index indicates only one cache line, the mapping is direct; if the block can be found in any cache line, the mapping is fully associative; and finally, if the set index selects a subset of all cache lines, the mapping is set-associative, with this approach the cache is divided into equal sets, where each set is called *way*. A cache hit is determined by comparing the address' *tag* with the tag of all the possible block locations. If there is a match, there is a hit; otherwise, there is a miss, so a cache line must be evicted, and the block where the data is located must be brought from the next memory level. If the cache mapping is not direct, a replacement policy must choose a line that needs to be evicted.

A direct mapping approach is the simplest but suffers the most from conflict misses: misses where the cache has available space, but the corresponding set is full. This happens because a block can only be allocated in one space of the cache; this can have a significant impact on a program's performance if multiple memory locations with conflicting set indexes are accessed repeatedly. On the other hand, a fully associative scheme gets rid of conflict misses; however, the comparison logic to determine whether the access is a hit or not, usually found in the critical path, is much more complex, which results in either a very large response time (higher latency) or in a reduced cache size, leading to more capacity misses. This is why, normally, all caches are set-associative, ranging from 4-16 ways, to reduce conflict misses without sacrificing latency as much. The associativity of the cache is one of the main design decisions a computer architect must make when designing a cache.



Source: ARM System Developer's Guide: Designing and Optimizing System Software [18]

Figure 1.3: Structure of a 4-way set associative cache with 256 cache lines, with a write-back policy

A write operation to the cache can be handled in different ways. If there is a hit, the data can either be written to both the cache and the next level, known as *write through*; or the data can be written to the cache marking that it has been modified (using a dirty bit), and wait until it is evicted to be written to the next level, known as *write-back*. In case of a miss, the block referenced by the write operation can be brought to the cache, *write-allocate*; or not, *write-not-allocate*. Normally, write-through caches also implement a write-not-allocate policy, and write-back caches implement write-allocate. The write-through policy is simpler to implement and eases the addition of coherency in the system (if needed), but it exhibits significantly worse performance.

1.4.1. Replacement policies

In set-associative caches, selecting a replacement policy is one of the most important aspects of cache design. Ideally, the evicted cache line should be the one that will be used the latest in the future; this is the theoretically optimal replacement policy, also known as Bélády's optimal replacement policy (OPT). However, the application of this policy is virtually impossible. Therefore, other replacement policies that approximate as much as possible to the optimal case must be formulated.

LRU

Least Recently Used (LRU) is one of the most popular replacement policies; it approximates OPT when the recent past is a good prediction for the future, and it is highly popular in page memory implementations [19]. LRU takes advantage of the temporal locality of data by replacing the cache line that was not accessed for the longest time.

Its implementation consists of setting up $log_2(Ways)$ bits in each way in a set, which indicates the order in which the ways were last accessed. When the line is accessed, the order of all the ways is updated. When an eviction is needed, the way with the lowest count is chosen to be replaced.

The main drawback of the LRU policy is its high hardware implementation cost, unfeasible in caches with more than eight ways.

pLRU

To adapt LRU to systems where its implementation would result in very expensive, new policies that approximate LRU but with a lower hardware cost appeared. This policies are called pseudo-LRU (pLRU).

One of the most common pLRU implementations consists of building a binary tree, where the leaves represent the ways of the cache. This policy receives the name of Binary Tree pLRU (BT-pLRU). BT-pLRU approximates an LRU policy by constructing a tree of size $Ways - 1$ bits for each cache set. Today, it is highly popular in cache implementations due to its significant area improvements with a negligible performance loss with respect to LRU.

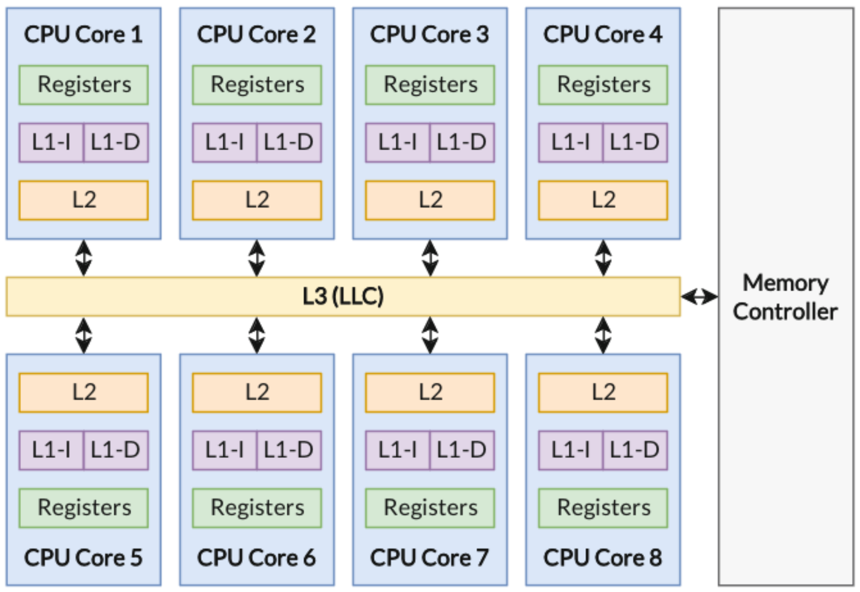
The value of each node in the tree indicates where the pLRU leaf is located: A '0' means it is down the left sub-tree and a '1' down the right sub-tree. The tree representation is stored in a vector where the root node is the Least Significant Bit (LSB), then its left child (LSB+1) and right child (LSB+2), and so on. To find the pLRU cache line, the tree must be traversed following the directions given by the nodes until reaching a leaf node. To update a node, indicating that it is the most recently used, all of the bits in the path to said node must point to the opposite sub-tree (e.g., if the MRU leaf node path is 0b001, that pLRU tree path must be updated to 0b110). [20]

1.4.2. Modern caches in multicore systems

As outlined in the previous sections, multiple levels of cache are used in usual multicore systems. In these designs, the last level of cache (LLC) is usually shared between all the cores to optimize performance while saving hardware resources. Figure 1.4 shows a cache hierarchy of a modern SoC.

Modern shared cache designs for high-performance embedded SoCS have to meet two main properties in terms of performance: peak performance and guaranteed performance (a.k.a QoS). That is, a cache design should significantly boost the performance of the SoC design and, in the context of a multicore with multiple execution threads, we additionally have to ensure that the impact of co-runners activity does not significantly degrade the performance of your task. Unfortunately, the use of shared resources by co-running tasks can significantly affect the application's performance. By using techniques such as isolation or prioritization, this interference can be reduced, ensuring QoS.

In addition, many optimizations are done to the basic design visited in figure 1.3 to improve the throughput of the cache; consequently improving the overall performance of the system. One of the most common optimizations found in modern shared caches is a non-blocking implementation, also known as a lockup-free cache. In a normal cache, when a transaction results in a miss, the component stalls and waits until the request done to the backend has finished before continuing. This is especially detrimental in shared caches because a cache miss causes interference in the rest of cores trying to access the shared resource. With a non-blocking implementation, cache hits can still be served while



Source: Trusted Execution Environments [21]

Figure 1.4: Modern eight-core x86-64 SoC cache hierarchy

a backend memory access is being handled in the background; significantly boosting performance by reducing inter-core interference.

1.5 Report Structure

This report is structured as follows: Chapter 2 will introduce the basic concepts needed to understand the extensions done to the cache, overview the state-of-the-art, and survey the available open-source caches. Chapter 3 will explore thoroughly the baseline cache design chosen, characterizing the component and explaining its internal functionality. Next, chapter 4 talks about the extensions done to the cache, emphasizing the implementation of the non-blocking cache and the partitioning scheme for both pLRU and random replacement policies. Finally, chapter 5 will talk about the testing environment, how the cache was verified, and show the performance improvements caused by the changes. Chapter 6 concludes the project and evaluates the outlined objectives.

CHAPTER 2

State of the art

This chapter will first explore the main technologies used in current cache designs, focusing on the standard interconnection buses, and how QoS is usually tackled in modern caches. After having explored the basic concepts necessary to understand the design of a cutting-edge cache, we will scout the available open-source caches in the market, analyze why a new product is needed, and choose the most adequate existing design for its posterior modification.

2.1 Cache standard interfaces

Using a standard interface is key to achieving a portable and reusable design. This is why this is one of the main features we are aiming for in our desired cache design.

2.1.1. AMBA AHB

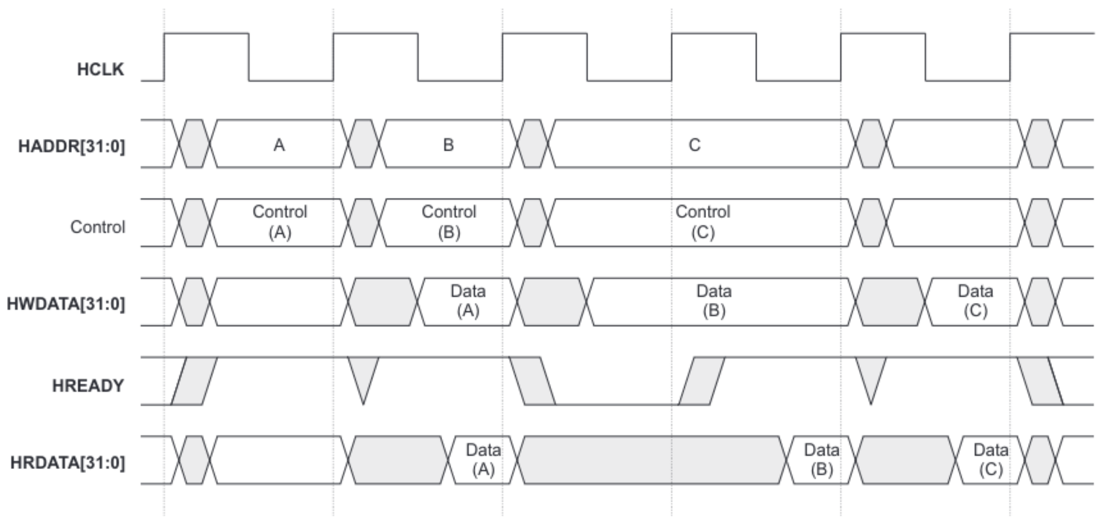
ARM's Advanced Microcontroller Bus Architecture (AMBA) is an open standard highly popular for interconnecting high-performance SoCs [22].

AHB is a multi-master, multi-slave bus with a centralized arbiter. It uses a single channel with a pipelined address and data operation, allowing only one active transaction at a time. Furthermore, AHB only supports one outstanding request per master at once, thus not allowing out-of-order transaction completions.

AHB transfers

AHB transfers are composed of two phases: an address and control phase, which lasts for one cycle; and a data phase, requiring as many cycles as needed.

Figure 2.1 shows an AHB multitransfer operation. First, a request from A starts by sending the address and control data. In the next cycle, the response to A is given by the slave, by writing the data into the `HRDATA` bus and setting `HREADY` high. In this same cycle, another request, B, is in the bus, so the slave must buffer its control data to respond in the next cycle. In cycle three, the slave needs more time to produce a response, so the `HREADY` signal is set low; this extends the data phase of B for another cycle as well as the address phase of the next incoming request, C. Then, B's response is ready so it is written into the bus in the following cycle. Finally, the response to C is ready in cycle five, in the same way as A's.



Source: AMBA specification 2.0 [22]

Figure 2.1: AHB multiple transfer example

AHB splits

There are some cases where the slave needs several cycles to produce a response, e.g. where the slave must access data from another component. As in AHB a request must remain in the bus until a response is given, this would block the entire interconnect, which severely affects performance in multicore settings.

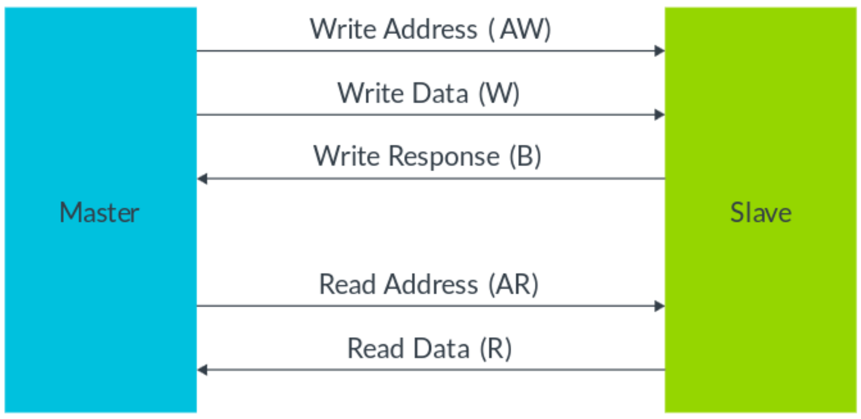
AHB 2.0 offers a special response to deal with this case: split transfers. Split responses are defined in the specification as follows:

> [The split response is used when] "The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete." [22]

By using this response, we can grant access to the bus to another master while the transfer is being processed in the background. The master who receives this response will not try to access the bus again until the slave allows it. Once the transfer is complete, the slave must assert the corresponding bit, according to the master number, in the split completion bus (HSPLIT). This will indicate to the master that it can access the bus again. It must be noted that the access to the bus is not immediately granted to this master, it will have to be arbitrated again.

2.1.2. AMBA AXI

AMBA's Advanced eXtensible Interface (AXI) is a point-to-point protocol highly popular in current interconnection implementations. It defines five main channels for communication between two nodes: the Write Address (AW), Write Data (W), and Write Response (B) for write operations; and the Read Address (AR) and Read Data (R) for read operations (see figure 2.2). The multichannel approach that AXI uses helps to improve the bandwidth of the interfaces, as read and write operations can be simultaneous.



Source: Introduction to AMBA AXI4 [23]

Figure 2.2: AMBA AXI4 channels

Furthermore, as opposed to AHB, AXI allows multiple outstanding transactions as well as out-of-order transaction completions, making easier the implementation of components such as non-blocking caches.

2.2 Cache QoS mechanisms: Partitioning

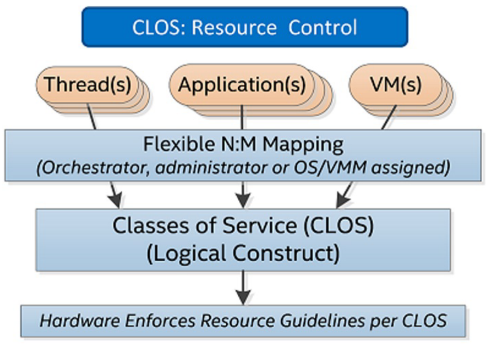
In recent years, effort has been put into the development of QoS mechanisms in order to tackle the inter-core interference found in multicore systems. One of the most common QoS schemes found in high-end commercial products is partitioning.

Partitioning has been shown to be an effective technique for reducing core interference in shared caches [24]. It removes inter-task eviction interference by restricting the eviction capacity of each thread/task. As a result, higher throughput and Quality-of-Service (QoS) can be achieved. Furthermore, partitioning is key in safety and real-time multicore systems to reduce unpredictability by assigning an exclusive cache space for each critical task, removing inter-core evictions. Initially, partitioning was exclusively implemented in high-end SoCs designed for HPC applications. Currently, it is being increasingly used in other domains, such as mobile or personal computer-oriented chips.

Intel supports partitioning through their Cache Allocation Technology (CAT) [25]. They introduce an intermediary abstraction called Classes of Service (CLOS), into which they group different threads and applications sharing the same cache space (figure 2.3). Then, the CLOS cache space can be configured to fit the applications' needs. Intel CAT is part of their Intel Resource Director Technology (RDT) feature set, present in their high-end server-oriented chips, such as the 4th and 5th generation Xeon lineup [26].

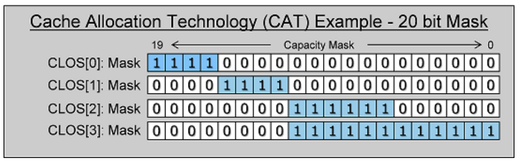
AMD also offers partitioning support through their Platform Quality of Service extensions (PQoS), available for their zen3, zen3+, and zen4 chips. They offer a very similar interface to the one offered by Intel, where the L3 cache space can be distributed among the cores using Classes of Service (COS). [29]

Finally, ARM's approach to partitioning is a bit different than Intel and ARMs. They also use an intermediary construct, which they call partition scheme IDs. The main difference is their way-based approach, where the cache space is divided by the number of ways. ARM not only offers partitioning in its high-end desktop processors, such as the Cavium ThunderX, but it is also implementing it in its new mobile state-of-the-art lineup, as the Artix-A720 chipset. [30]


CLOS: Resource Control
Thread(s)
Application(s)
VM(s)
Flexible N:M Mapping
(Orchestrator, administrator or OS/VMM assigned)
Classes of Service (CLOS)
(Logical Construct)
Hardware Enforces Resource Guidelines per CLOS

Source: Intel Corporation [27]

(a) CLOS definition


Cache Allocation Technology (CAT) Example - 20 bit Mask
19
Capacity Mask
0
CLOS[0]: Mask 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CLOS[1]: Mask 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
CLOS[2]: Mask 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0
CLOS[3]: Mask 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

Source: Intel Corporation [28]

(b) Mapping of CLOS to cache space

Figure 2.3: Intel CAT mechanism

Although partitioning is slowly becoming a fundamental feature in many-core systems, commercial companies such as ARM, Intel, or AMD hide their microarchitectural implementation; so, even if it has been proven to be an effective technique to improve both QoS and throughput, not much information is found about how to implement it in RTL.

2.3 Open-source existing caches

As of today, most modern open-source SoCs make use of cache modules. In this section, we survey the most popular, and high-performance open-source caches in the industry, outlining their most prominent features, as well as their limitations.

All of the designs we are going to study are intended for its use in real hardware systems, SoCs for the most part. In general, hardware is described using specialized languages, called Hardware Description Languages (HDL). The two most popular HDLs are Verilog (and its extension SystemVerilog) and VHDL; both are based on low-level programming languages, C and ADA, respectively. In recent years, some high-level HDLs, especially Chisel, introduced by UC Berkeley in 2012, have gained popularity. However, they have been used mostly in educational environments and have not yet reached the industry, mainly due to the difficulty of integration in non-Chisel environments. This is why Chisel designs are not included in this analysis.

2.3.1. PoC Cache

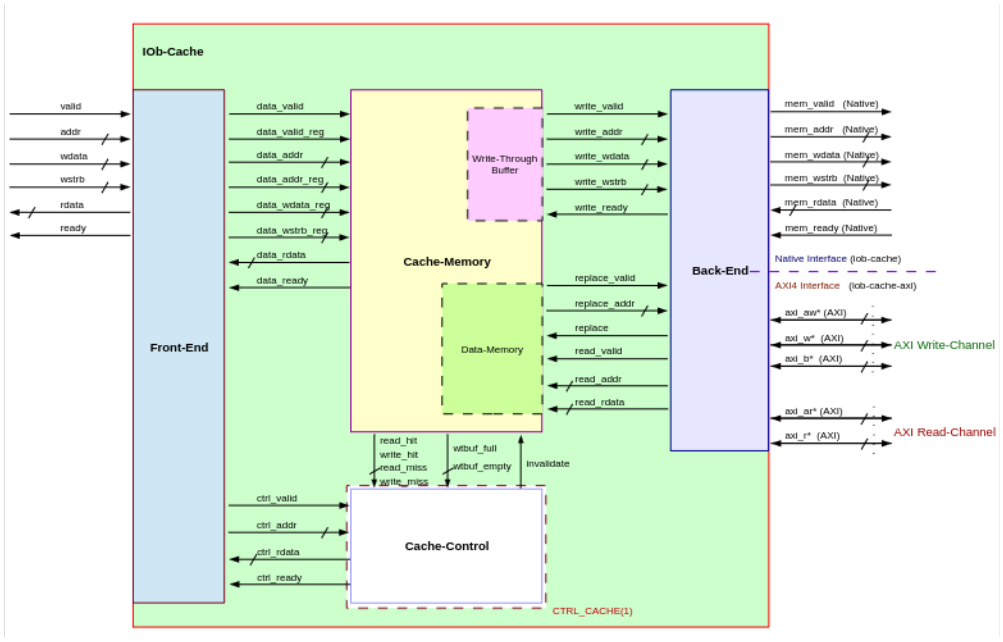
The "Piles of Cores" (PoC) library, developed by the *"Technische Universität Dresden"*, is a project that develops open-source IP cores, described in VHDL, for their reusability in any kind of project [31]. The PoC library is licensed under the Apache License Version 2.0.

The PoC library has different cache implementations, but we will only look into the most complete one: *PoC.cache.cpu*. This module offers a very lightweight implementation of a cache memory. The cache associativity can be configured to be direct-associative, set-associative, or fully associative. It offers a LRU replacement policy and configurable ways and way-size.

However, the PoC cache is very limited. It implements a write-through and no-write-allocate policy without a write-buffer, worsening multicore performance significantly.

Additionally, both frontend and backend interfaces are native, hampering its portability to other systems. Therefore, we consider that PoC is not good enough for the necessities found in current SoCs.

2.3.2. IOb-Cache



Source: IOb-Cache: A High-Performance Configurable Open-Source Cache [32]

Figure 2.4: IOb-Cache top-level module diagram

The IOb-Cache is a Verilog open-source highly configurable cache developed by IObundle for their open-source RISC-V-based SoC: IOb-SoC.

This cache can be set up as either an L1 or L2 cache, it offers an AXI backend interface and implements three different replacement policies: LRU, BT-pLRU, and MRU-pLRU.

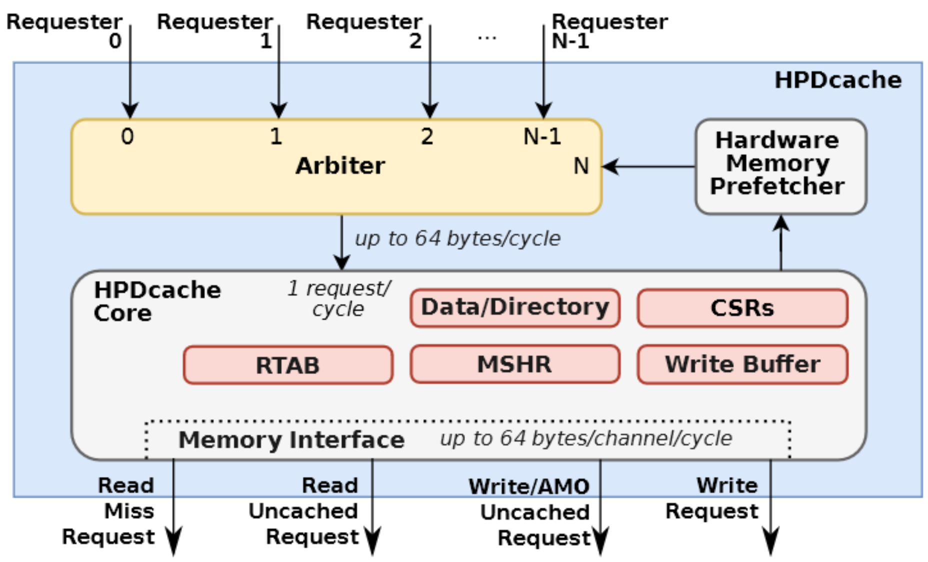
On the other hand, IOb-Cache does not support a standard frontend interface, difficulting the reusability of the module; it uses a write-through and write-not-allocate policy, that, even with a write-through buffer, puts a strain on main memory and hinders scalability; and it doesn't implement any non-blocking mechanism.

Although IOb-Cache offers a good simple cache module that is highly configurable for its use in very diverse systems, its lack of scalability results in poor performance in multicore systems. Thus, this cache does not fit our needs.

2.3.3. HPDcache

Currently, there exist more open-source L1 cache designs available than L2 ones. One alternative could be using a high-performance configurable L1 cache and adapting it to be used as an L2.

In this context, the HPDCache developed by OpenHW appears as an interesting alternative [33]. OpenHW is a non-profit organization that develops open-source IP cores and modules for their use in other projects. HPDCache is used in open SoC designs such as OpenHW's CV6 core.



Source: HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores [33]

Figure 2.5: HPDcache microarchitectural overview

HPDCache is an L1 cache that offers a non-blocking pipeline with up to 128 MSHRs, with a write-through and write-no-allocate policy and a write buffer. The cache associativity and size can be configured at compilation time. The design also includes a configurable hardware prefetcher that can improve the cache's throughput.

The main drawback of L1 designs is that they are not thought for shared environments. As such, they lack the necessary frontend features necessary for their use in multicore systems. In the case of the HPDCache, the frontend bus implements a fixed priority mechanism in which a static priority must be set between the cores connected to it at compilation time. Also, the line size is limited to 32b, which must be changed for its use as an L2.

2.3.4. OpenPiton

OpenPiton is an open-source manycore research framework created by Princeton University [34]. OpenPiton includes a high-performance L2 cache in its framework for their processor implementations. OpenPiton's cache exhibits good multicore performance thanks to its 8-entry MSHR, allowing it to have up to 8 outstanding misses. It implements a write-back and write-allocate policy and offers a pseudo-LRU replacement policy. Both cache size and associativity can be configured.

Although OpenPiton offers great performance, its reusability is quite difficult, and requires modifying significantly the original design. Firstly, their L2 cache is designed to be distributed. Then, their L2 implements a native bus in its backend and frontend. In essence, the component is highly coupled to their SoC, making portability to other projects costly.

2.3.5. L2C-Lite

The GRLIB IP library [35, 36], developed by Frontgrade Gaisler, offers multiple components for the implementation of complete SoCs. This library contains modular and reusable plug & play IP cores that make adding extensions to an existing system efficient and cost-effective. The GRLIB open-source version of this library comes with a GNU GPLv2 license for academic and prototyping purposes.

L2C-Lite is their open-source L2 lightweight cache controller. It offers a high configurability, implements standard buses (AHB and AXI), and uses a write-back, write-allocate policy. Furthermore, two different replacement policies can be configured at compile time: pRandom or BT-pLRU.

However, its multicore performance is quite limited due to the blocking nature of the cache, even with asynchronous backend writing support. This restricts the scalability of the systems implementing it.

2.3.6. Comparison

Table 2.1 resumes the cache comparison. This table shows how the available open-source caches in the market, are not fit for the proposed problem. Thus, to address our needs, we decided to extend the functionality of one of the caches.

	PoC Cache	**IOb-Cache**	**HPDcache**	**OpenPiton**	**L2C-Lite**
HDL	VHDL	Verilog	SystemVerilog	Verilog	VHDL
Configurability	Medium	High	High	Medium	High
Standard Interfaces	No	AXI4 in backend	AXI5 backend adapter available	No	AHB in frontend AHB & AXI in backend
Portability	Medium-Low	Medium	Medium	Low	Medium-High
Write policies	Write-through w/o write buffer	Write-through with write buffer	Write-through with write buffer	Write-back	Write-back
Replacement policies	LRU	pLRU	pLRU	pLRU	pRandom or pLRU
Non blocking features	None	None	up to 128 MSHR	8-entry MSHR	Asynchronous backend write
Multicore performance	Low	Low	Not supported	High	Medium-Low
QoS mechanisms	None	None	None	None	None

Table 2.1: Available open-source caches comparison

The L2C-Lite was considered the most complete alternative, due to its high configurability and portability, as well as a quite complete design feature-wise, which could be easily upgradable.

CHAPTER 3

Baseline Design Analysis

In this chapter, we will delve deeper into the previously introduced design of the L2C-lite module, outlining its limitations and understanding its structure for the upcoming modifications to the design.

3.1 Configuration

One of the most prominent features of the L2C-lite open-source cache is its high configurability. It allows designers to choose the cache associativity, way size, line size, endianness, replacement policy, and the backend bus and width. This flexibility makes it usable in both low and high-end embedded systems. Table 3.1 summarizes the component's configuration options.

Parameter	**Values**
Ways	1,2,4,8,16,32
Line size	32,64B
Way size	1,4,8... 1024KiB
Endianness	little endian, big endian
Replacement policy	pRandom, BT-pLRU
Backend interface	AHB2.0,AXI4
Backend bus width	32,64,128B

Table 3.1: L2C-Lite Parameters

3.2 L2C architecture

The architecture of the L2C-Lite module is formed by three different components, shown in figure 3.1. First, we find a control component, that interacts with the frontend interface, which is hardcoded into the module, and handles the Finite State Machine (FSM). The memory module contains the logic surrounding the SRAM, provided by Gaisler's `SYNCRAM_2P` IP core. Finally, we find a backend module that implements a detached memory interface. This module is also a Gaisler IP Core, called Generic Bus Master (GBM). L2C-Lite exclusively uses AHB 2.0 on its frontend. For the backend, the L2C-Lite cache enables the use of either AHB 2.0 or AXI.

The L2C-Lite design is quite modular; however, the frontend could also be detached from the main functionality of the component, making the design even more portable,

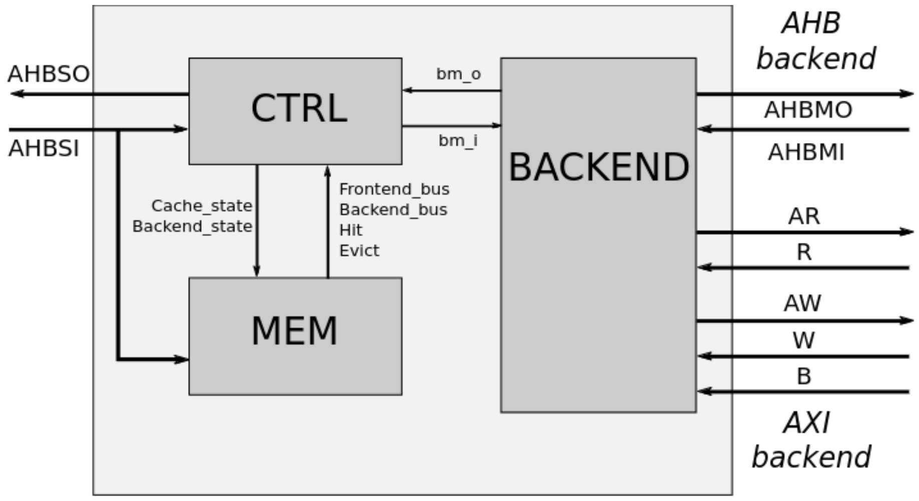


Figure 3.1: L2C-Lite top level module diagram

flexible, and extendable in the future. This is one of the design aspects that will be tackled in our extension.

Control

As aforementioned, the control module manages the state machine of the cache depending on the inputs given by the frontend bus and the current state of the cache module. The internal registers of the cache are located here: a status register, an access and hit performance counters, and a flush enable register.

The control module can be considered as the main module of Gaisler's cache. It is responsible for handling both frontend and backend interfaces, using *AHB* and *bm* buses; it communicates to the mem module with the *ctrl* bus, passing all the necessary information to ensure the correct functionality of the component; and it handles the state machines and internal registers. This results in a very high-complexity module, with many functionalities, that complicates its modification.

Figure 3.2 shows the FSM of the component. The states of the l2c-lite cache are:

- IDLE → The idle state samples the frontend bus and changes the state of the cache to serve a request if needed. If the request is valid, and it wants to access the cache memory, the state is changed to either `READ` or `WRITE`, depending on the type of operation. If the address is not cachable, a direct memory operation will be needed, then the next state will be either `DIRECT_READ` or `DIRECT_WRITE`. Finally, if the request targets the cache internal registers, the transfer is handled by the idle state. In the special case that the flush bit is set, the cache moves to the `FLUSH` state.

- READ → The read state implements a read operation to the cache. First, if the backend is busy, wait cycles are introduced. If the requested block is not in the cache, the access will be a miss and the data must be fetched from the next memory module; so the next state is set to `BACKEND_READ`. If the access is a hit, the data given by the memory module through the *ctrl* bus is copied to the frontend bus, which finishes the operation. If the next request is a read burst operation to the same memory block, the next state is set to `R_INCR`, otherwise, it is set to `IDLE`.

- WRITE → In a similar way as in the read state, after checking if the backend is free, if the access is a miss, the next state is set to `BACKEND_READ`. If it is a hit, one cycle of

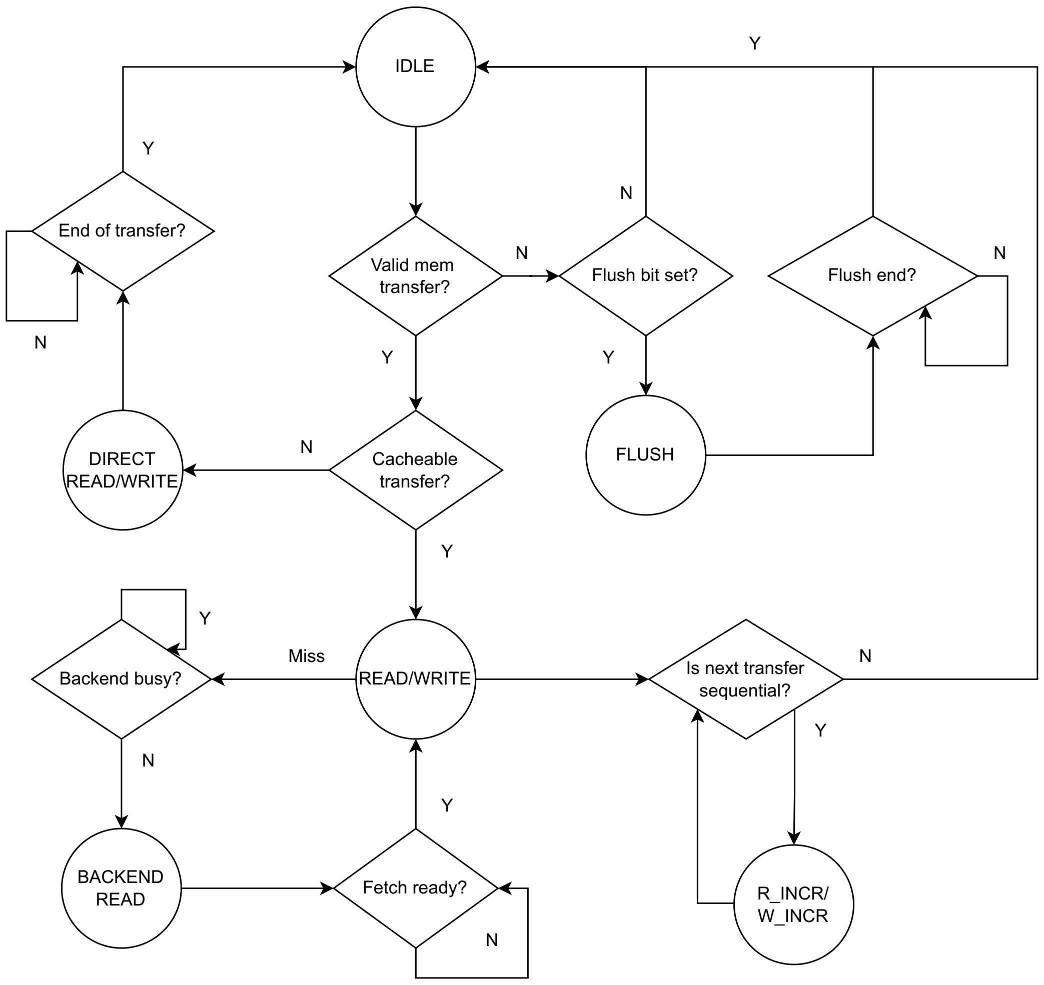


Figure 3.2: L2C-Lite FSM. Write and read states are grouped to improve readability

delay is introduced to complete the read-modify-write sequence. The next state is set to `W_INCR` if the next request is a write burst to the same memory block; else, the cache moves to the `IDLE` state.

- BACKEND_READ → This state does a read operation to the next memory level to obtain the data needed to complete either a read or write operation. The *bm* bus is used to do so. The cache waits until the data is ready. Once ready, the fetched data is written into the cache and the state moves to either `READ` or `WRITE` depending on the transfer type.

- DIRECT_READ → A read operation to the next memory level is done in the same way as in the backend read state, but instead of sending the data to memory module to write it into the cache memory, the data is written to the frontend bus. Once finished, the cache moves to the `IDLE` state.

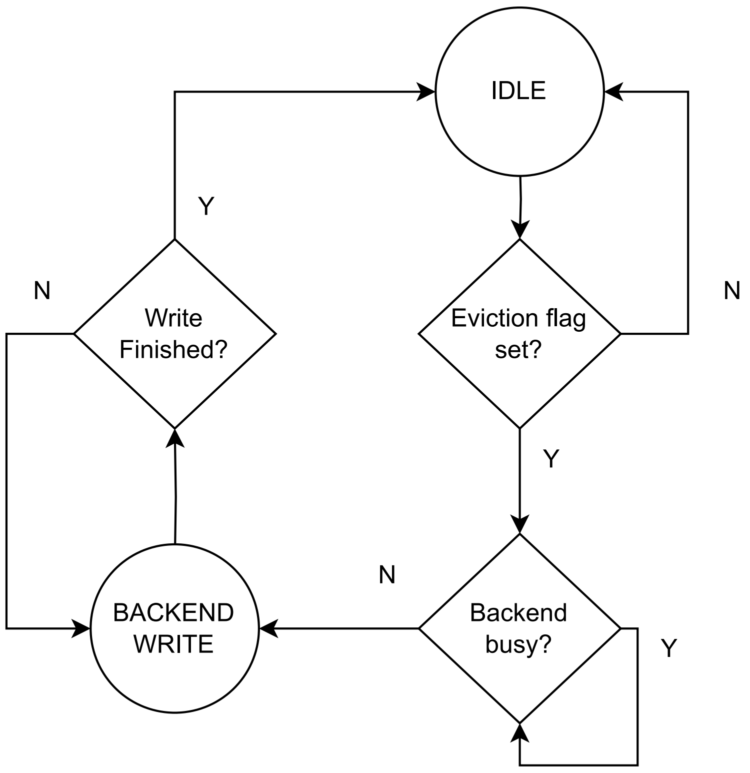
- DIRECT_WRITE → Issues a write request to the next memory level and waits until the transaction is confirmed. Then, the next state is set to `IDLE`.

- FLUSH → The cache invalidates all of the availble lines. The cache waits until the memory module indicates that the flush operation has finished, then goes to `IDLE`.

- R_INCR → This state writes into the frontend bus the requested data available in the *ctrl* bus because of the previous read operation. If the next operation is a read burst accessing the same block, the state of the cache will not change. Otherwise, it is set to `IDLE`.

- W_INCR → The cache either keeps its state if the next operation is also a write burst to the same block, or moves to `IDLE`. In this case, a delay cycle is not needed because the previously read data can be reused.

The control module contains another state machine containing the necessary logic to implement the asynchronous writes to the next memory level (figure 3.3). This FSM has two states: `IDLE` and `BACKEND_WRITE`. The backend moves to the backend write state when, after a miss to the cache, a dirty line is evicted and the new line is being written into the data memory. The logic needed to do this write operation to memory is the same as the one used in the direct write transfers, writing the request into the *bm* bus, and waiting until the transfer is confirmed.



Based on diagram in [37]

Figure 3.3: L2C-Lite backend finite state machine

Memory

L2C-Lite implements write-back and write-allocate policies. It comprises two types of memories: One data memory, which holds the cache lines; and one tag memory, in which the tag, the line owner, and the valid and dirty bits are stored. Each way has its own separate tag and data memories. These memories are dual-port, allowing one read and one write operation per cycle. However, L2C-Lite only allows one operation (either write or read) at a time. This will be optimized in the new version of the cache.

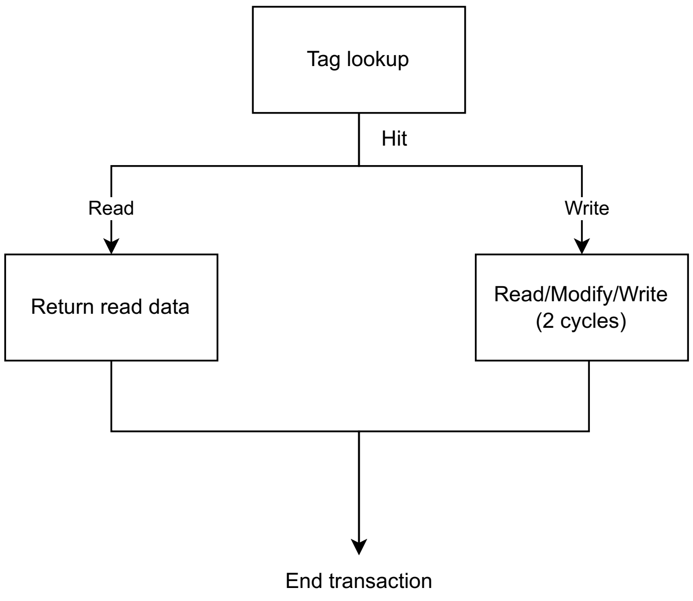


Figure 3.4: L2C-Lite hit flow

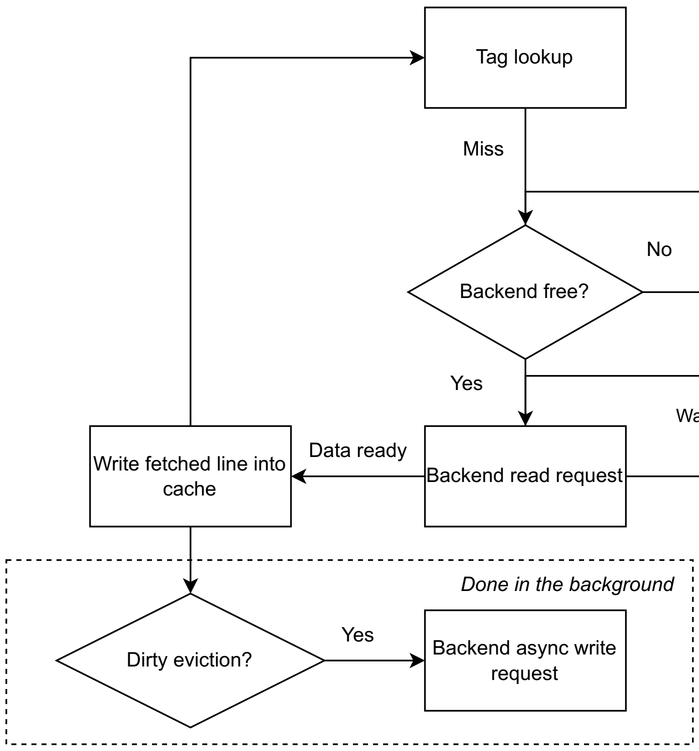
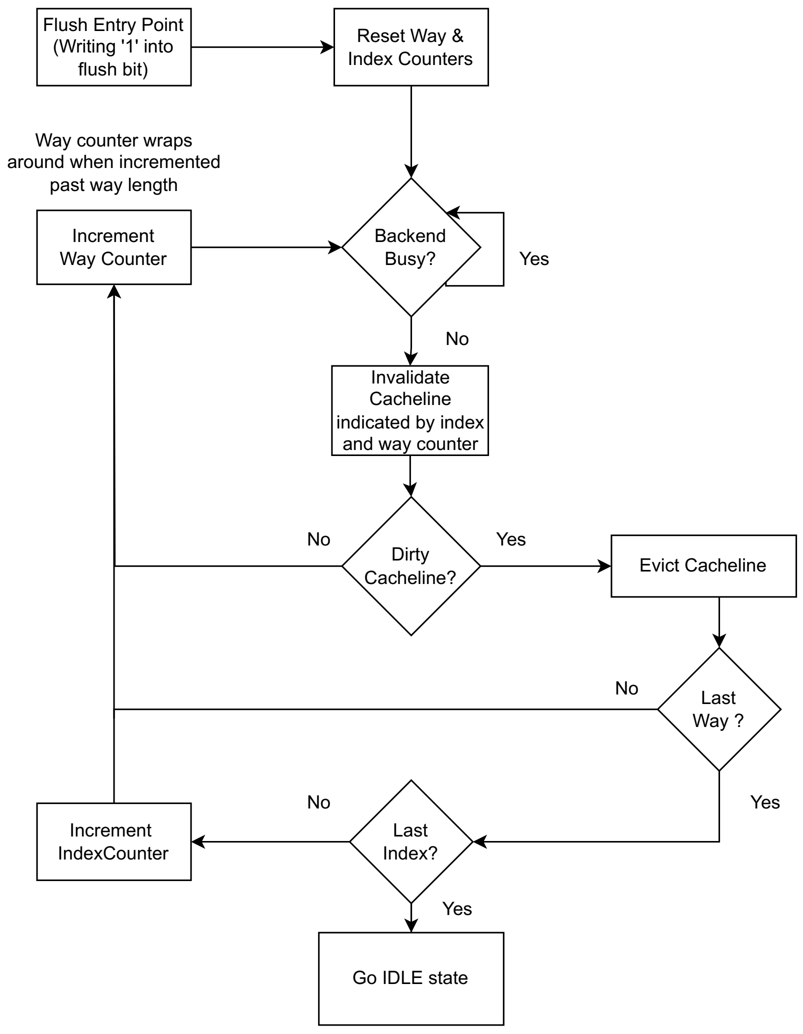


Figure 3.5: L2C-Lite miss flow

L2C-Lite is a blocking cache, only supporting processing one request at a time. A request to the cache starts with a parallel tag lookup, checking whether the access is a hit or a miss. A hit read takes one cycle, and a write operation takes two cycles due to the read-modify-write sequence (figure 3.4). When a miss occurs, the cache will wait until the backend is free (if needed) and until the backend read operation ends, not processing any new possible upcoming requests. However, the cache offers asynchronous backend writes, allowing cache hits to be served while an evicted line gets written to the next memory level. Figure 3.5 summarizes the miss flow of the component.

The replacement policy implementation is found in the memory module. In the current design, changing the replacement policy can only be done at runtime. The random policy consists of a simple wrapping counter that represents the way to be replaced. On the other hand, the pLRU replacement policy is implemented in two functions, one chooses the line to evict in case of a miss, and the other one updates the pLRU tree in case of a hit.


Flush Entry Point (Writing '1' into flush bit)
Reset Way & Index Counters
Way counter wraps around when incremented past way length
Increment Way Counter
Backend Busy?
Yes
No
Invalidate Cacheline indicated by index and way counter
No
Dirty Cacheline?
Yes
Evict Cacheline
No
Last Way ?
Yes
No
Increment IndexCounter
Last Index?
Yes
Go IDLE state

Based on diagram in [37]

Figure 3.6: L2C-Lite flush mechanism flow diagram

The flushing mechanism in the cache invalidates all of the cache lines. As write-back is the write policy being used, the dirty lines must be written back to the next memory level. Figure 3.6 shows the flow of the flush state of the cache. Two counters, one that keeps track of the cache way, and another one that keeps track of the index, are being used. All of the cache lines are visited, invalidating them, and writing them to the next memory level if needed.

3.3 Methodology

All of Gaisler's synchronous, single-clock modules are based on the two-process hardware description methodology. This methodology helps to reduce the number of errors when describing a component and, thus, helps to reduce the development time.

This methodology defines that each synchronous entity (VHDL file) must comprise only two processes. The first one must contain all of the combinational logic, which does not depend on the clock. The second process must contain the sequential logic, that is, the description of the registers of the component. Furthermore, the system's registers are defined in records, which makes adding or modifying signals much easier and prevents errors such as not setting default values, causing undesired flip-flops; or missing signals in the sensitivity list, which can cause the synthesis tools to incorrectly reduce necessary logic. [38]

This methodology is crucial to understand the VHDL code in the L2C-Lite module and to extend the module effectively. In addition, this approach will be followed for the rest of the new modules created to provide future functionality in the module.

CHAPTER 4

Architectural design & implementation

This chapter explains in detail the changes done to the original design to achieve a more modular, scalable, and fit design for multicore systems; emphasizing the implementation of the non-blocking and partitioning mechanisms, central to this work. We call our new extended design L2C-Lite+, and will refer to it as such.

This new design has been publicly published under the same license as L2C-Lite, the GNU GPLv2 license. It can be found in: https://github.com/Dar0k/L2C-Lite-plus.

4.1 Revisited cache architecture

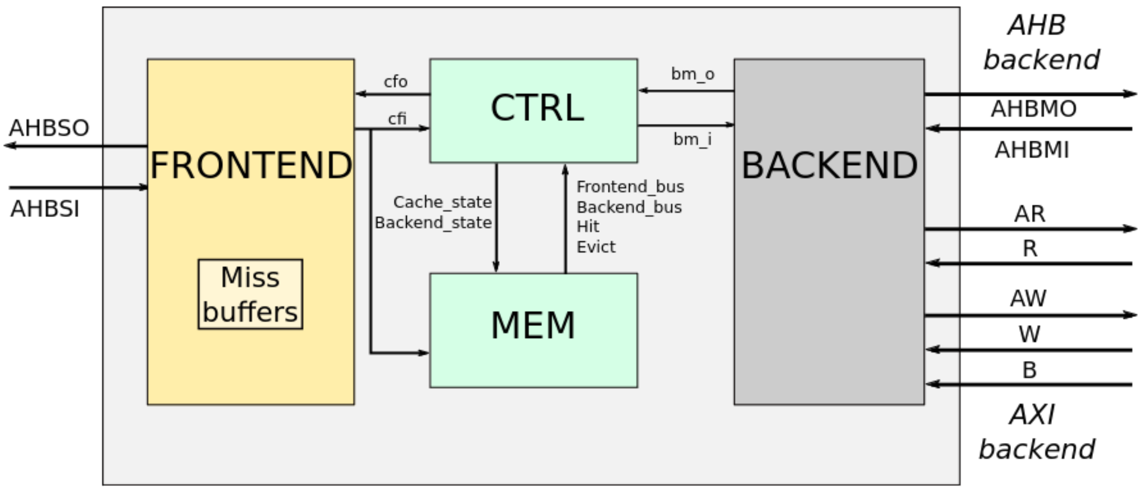


Figure 4.1: L2C-Lite+ architecture top view.
In yellow: new modules; in green: modified modules

To achieve a more modular and flexible design, we revisited the existing cache architecture, shown in figure 4.1. The main change made to the architecture was the addition of a frontend module. This new addition isolates the bus implementation, previously hardcoded into the internal cache operation, which makes swapping the frontend bus to a different technology much more effortless. Not only that; it also facilitates the modification of the request and response mechanism for further extensions. Furthermore, this significantly reduced the complexity of the previous control and memory modules, making the new version more simple and comprehensible.

Our new frontend implementation keeps supporting only AHB 2.0, whose requests are translated into a less complex native bus that fits the needs of the cache. However, the addition of the frontend module enables the seamless integration of other more powerful interfaces such as AXI.

4.2 Frontend

The frontend module was designed to be as simple as possible, allowing future extensions and optimizations if needed. This module handles both requests and responses of the AHB slave interface. The requests are translated from AHB into a simpler bus, the cache-frontend bus: `cf`. This bus implementation contains only the necessary information for the correct operation of the cache, simplifying the control and memory modules. Tables 4.1 and 4.2 show the cf bus signals' description.

Signal name	**Size in bits**	**Description**
Valid	1	Indicates whether the request must be served ('1') or ignored ('0')
Write	1	Indicates whether the operation is a read ('0') or a write ('1')
Transfer_size	3	Indicates the size of the transfer. AHB's transfer sizes, defined in the HSIZE vector are being used [22]
Addr	32	Indicates the operation's target address
IO_request	1	Indicates whether the access is directed to memory ('0') or to internal registers ('1')
Master	4	Indicates the cpu or peripheral that made the request
Write_data	AHBDW	Bus where the data for write operations is located

Table 4.1: Frontend to cache bus (cfi) signals' description

Signal name	**Size in bits**	**Description**
Resp	2	Cache response signal. Four different signals are available: Wait ("00") when more time is needed to obtain a response, Ready ("01") when the transfer ends, and then miss_accept ("10") and miss_deny ("11") to support non-blocking backend transfers
Read_data	Linesize	Returns the requested read data from memory
Fetch ready	1	Indicates when the previous backend transfer has ended ('1'), and the backend gets freed

Table 4.2: Cache to frontend bus (cfo) signals' description

The frontend unrolls AHB's pipelined address/data phases (section 2.1.1), maintaining in the cf bus the current request at all times. A MUX is used to drive the requests into the cache. When the cache indicates that the previous request has ended (through the response signal), the next request is selected in the MUX, saving one cycle. This is necessary in order to pipeline the request/response mechanism that we implemented in the cache, which will be explained in detail in the following sections. Figure 4.2 shows the physical implementation of the frontend mechanism explained above.

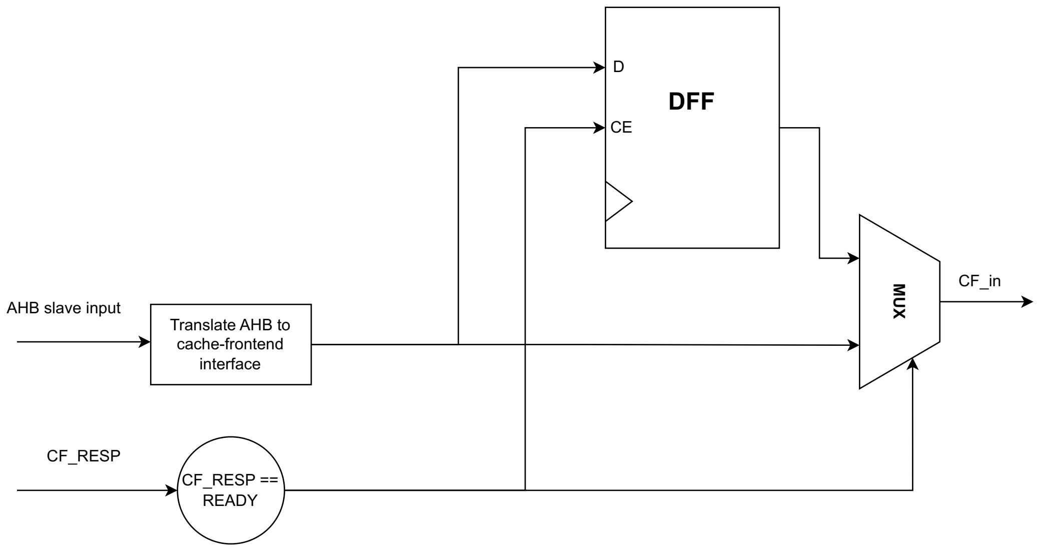


Figure 4.2: Frontend pipeline unrolling physical implementation

4.3 Control module

In this section, we introduce the contributions done to the control module of the cache.

4.3.1. Non-blocking implementation

In our cache design, we changed the blocking implementation of the original L2C-lite into a hit-under-miss approach, in which we can keep serving hits as they come, but only have one outstanding request into the backend. This implementation required changing significantly the cache's access flow. To support this, the cache response has been extended to include two new cases: `MISS_ACCEPT` and `MISS_DENY`. As in the original version, the first action that is performed when a memory request enters the cache is looking up the tag, which will decide whether the transaction results in a hit or a miss. The cache behavior for serving hits has not been modified significantly. However, the miss flow has been notably altered. In our implementation, a miss transfer is only processed if the backend is free. Otherwise, a `MISS_DENY` response is given, indicating to the frontend that the transfer must be retried later.

Once the backend is free, if the incoming request is a read miss, the cache gives a `MISS_ACCEPT` response. This indicates that the request has been made to the next memory level, and a signal will be made active once the data is received. Once the data is received, the transfer is completed by writing the response into the frontend bus. At the same time, this upcoming data is written into the cache's memory.

On the other hand, when a write miss request comes (and the backend is free), it gets accepted, giving an `OKAY` response, indicating that the transfer has been completed. The cache saves the write data, and launches a request to the backend. Once the data is back, it is modified according to the write operation and written into the cache. As the data has

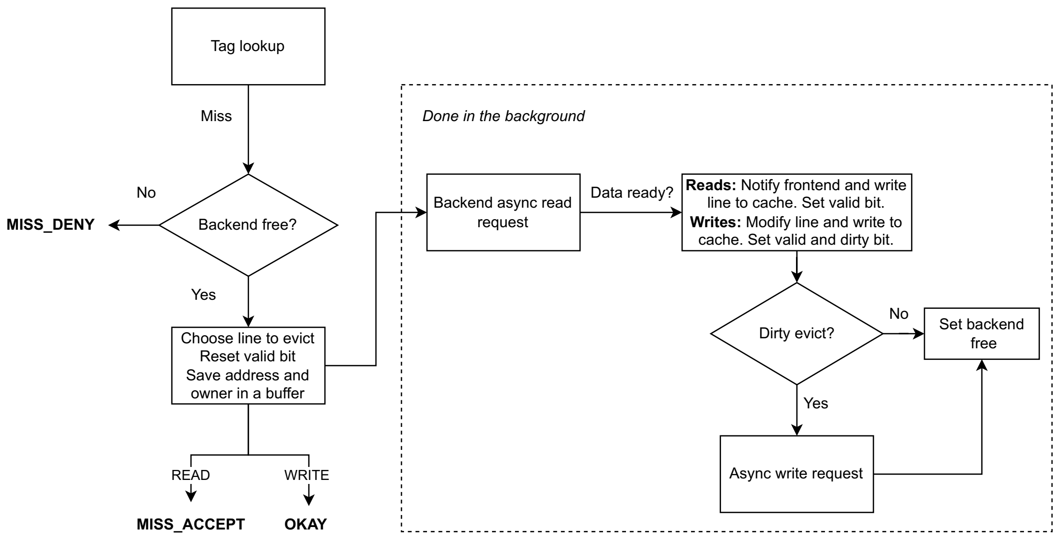

Tag lookup
Miss
Backend free?
No
MISS_DENY
Yes
Choose line to evict
Reset valid bit
Save address and owner in a buffer
READ
WRITE
MISS_ACCEPT
OKAY
Done in the background
Backend async read request
Data ready?
Reads: Notify frontend and write line to cache. Set valid bit.
Writes: Modify line and write to cache. Set valid and dirty bit.
Dirty evict?
No
Set backend free
Yes
Async write request

Figure 4.3: L2C-Lite+ miss flow

been modified, the dirty bit is set. Further accesses to this line while it is being fetched from memory won't cause a problem as they will result in a miss and will be denied until the backend is free, when the operation has ended.

In both reads and writes there are some common extra actions being made to ensure that the mechanism works as intended. Once the evicted line has been chosen, its valid bit is set to '0'. If the evicted line is dirty, the write request to memory will be made immediately after the miss request data gets back, having priority over other misses. Finally, in order to write the data back into the cache after it gets back from the backend, some information, such as the address and the owner, must be saved in buffers to update the tag accordingly. Figure 4.3 summarizes the non-blocking mechanism changes.

If a miss is coming back from the backend while a write hit operation is taking place we need to set some type of priority, as both operations request to write to the cache simultaneously. In this case, writing the miss to the cache will have priority over serving the hit, resulting in a wait cycle being inserted into the write hit transaction.

Frontend adaptation

The design of the non-blocking mechanism has been made keeping in mind the different possible frontend implementations, and it has been tried to make it as portable as possible. For example, proper MSHR registers could be set up in the frontend, which can send the next miss transfer as soon as the backend gets freed.

Our frontend implementation uses AHB, a blocking bus, in which the next transfer cannot be served until a response for the previous one has been given. Thus, adding MSHRs to the frontend would not be as beneficial. Our solution makes use of AHB's split responses. This special response unlocks the bus and sleeps the master until it is explicitly re-activated by making use of the `HSPLIT` vector. When a `MISS_ACCEPT` or `MISS_DENY` response comes from the cache, the frontend layer sends a split response to the master. Then, the bus gets freed to be able to serve new requests. Once the requested (read) data returns from the backend, the frontend awakens all of the slept masters.

One thing to consider is that, in AHB, once a master gets awakened it must compete again for access to the bus. Then, it is possible that in read miss transactions the fetched data from the backend gets modified or even evicted from the cache before the requesting master can read it. This can lead to errors or cyclic dependencies that prevent the core from advancing. To solve this, we added a miss buffer for each master to the frontend. This buffer, of size $linesize + 1$, holds the last fetched data from the backend and a valid bit. When the fetched data returns from the backend, it is written into the cache and forwarded to the frontend, where it gets saved in the buffer corresponding to the requesting master, and the valid bit is set. When a master accesses the cache, if its miss buffer's valid bit is set to 1, the frontend returns the corresponding data from the miss buffer and resets the valid bit.

It must be kept in mind that with this implementation, a read miss transfer is more expensive than before, as now, on top of the miss base latency, the master must compete to access the bus to read the data. However, this could be avoided by using a non-blocking bus. This would improve the cache throughput and not affect the transfers' latency. However, we decided to make this trade-off to improve multicore performance.

4.3.2. Optimizing for AHB: Pipelining requests and responses

To fully optimize the throughput using AHB, we pipelined the request and response mechanism, taking advantage of AHB's implementation, where the response to the current request is made while the next request is available in the bus.

Then, now, instead of always passing through the `IDLE` state, if another valid request is ready in the bus, the next state is calculated just at the end of the previous transaction, saving one cycle in the case of back-to-back transfers. Otherwise, the cache will still move to the `IDLE` state.

Even if this optimization is not as useful in single-core environments, as idle cycles are usually found between requests; in multicore systems, where the interconnect pressure is much higher, most requests are back-to-back, making the saved cycle noticeable. This functionality has been implemented in a separate function that is called at the end of each of the states, simplifying the code significantly. The implementation of this function can be seen in listing 4.1.

```vhdl
function stateAfterResponse(signal frontend : in cf_out_type)
    return state_type is
    begin
        if frontend.valid = '1' then
            if frontend.IO_request = '0' then -- Request to cache data
                if frontend.write = '0' then  -- Read
                    if is_cachable(frontend.addr(31 downto 28), cached) then
                        return READ_S;
                    else
                        return DIRECT_READ_S;
                    end if;
                else    -- Write
                    if is_cachable(frontend.addr(31 downto 28), cached) then
                        return WRITE_S;
                    else
                        return DIRECT_WRITE_S;
                    end if;
                end if;
            else    -- IO request

                if frontend.write = '0' then
                    return IO_READ_S;
                else
```

```
                    return IO_WRITE_S;
                end if;
            end if;
        else    -- Not valid
            return IDLE_S;
        end if;
end function stateAfterResponse;
```

Listing 4.1: Function to calculate next state after operation has ended

4.3.3. Simplifying the state machine

Some changes were made to the state machine to implement the non-blocking cache and the pipelining mechanism explained above. Two new states were added, and three were removed, simplifying the state machine overall. Figure 4.4 shows the new main cache FSM after the changes.

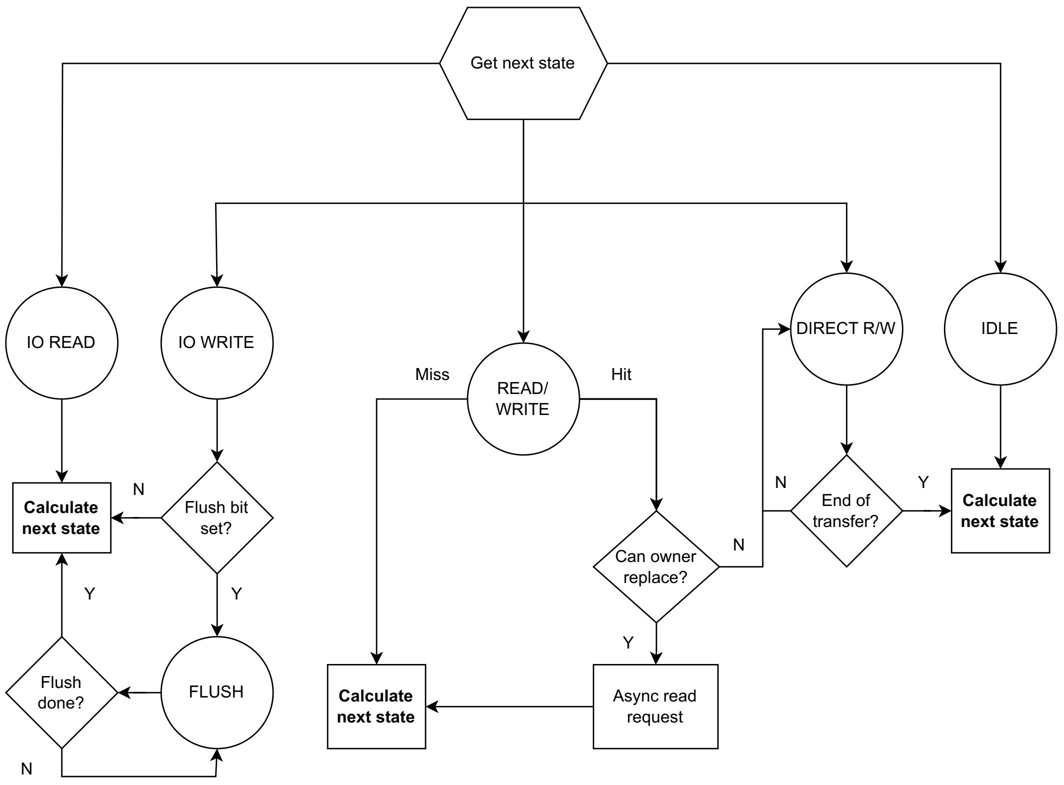


Figure 4.4: L2C-Lite+ main FSM

The two newly added states were `IO_READ` and `IO_WRITE`. These states enclose the functionality to access and modify the cache internal registers, previously held in the `IDLE` state. This makes the `IDLE` state smaller and easier to understand as now only one main function is done in each state.

On the other hand, some other states were removed. Two of them were the `R_INCR` and `W_INCR` states, which gave support to burst operations. As the cache now does not add an idle cycle between requests, an N read burst (hit) operation can be served in N cycles. Thus, the additional logic is not necessary now, and the states can be removed. In the next section, we will explain the changes made to provide one-cycle write operations;

this means that the write burst state will also no longer save any cycles, and thus, it can be removed.

Finally, the `BACKEND_READ` state was moved to the backend state machine to support non-blocking backend read operations. This extends the backend FSM as it can be seen in figure 4.5.

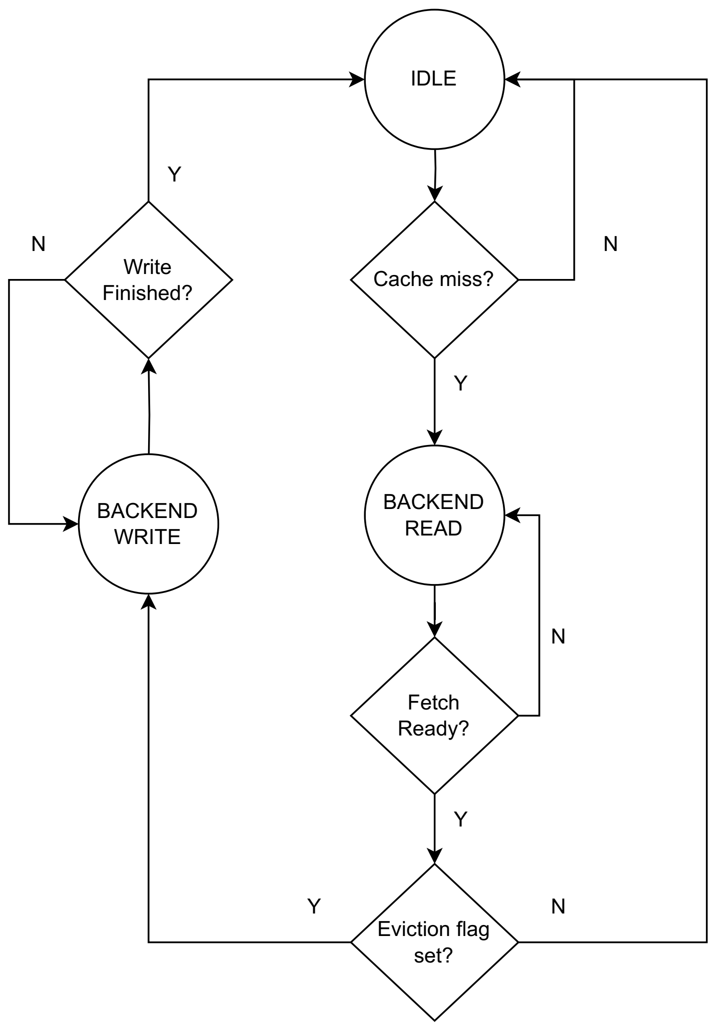


Figure 4.5: L2C-Lite+ backend finite state machine

4.4 Memory module

This section talks about the contributions made to the memory module of the cache.

4.4.1. Simultaneous read and writes

The previous cache design did not take full profit from the underlying two-port memory because, even if simultaneous read and write operations to the memory were possible, only one of them was done at the same time.

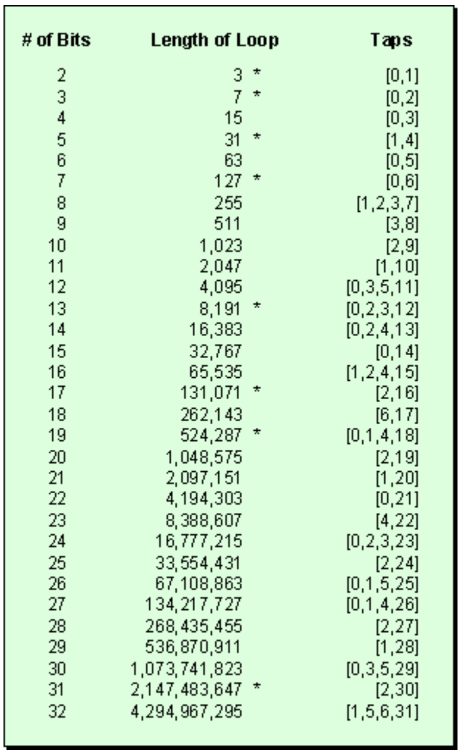
This optimization makes write-hit operations last one cycle instead of two. When a write request starts, in cycle zero, a read operation to the cache is initiated. In the next cycle, cycle one, the memory returns the data, the tag is checked, and if it matches, the data is modified accordingly. Finally, the write operation is prepared to be effectuated at the end of the cycle. If the next request requires reading from the cache, a reading request to the memory is prepared to be done simultaneously with the write.

One of the cases that must be considered is when writing and reading operations are being made simultaneously to the same memory address. The *SYNCRAM* module being used did not guarantee that the result would be correct. Thus, extra logic is being used to bypass the write data into the reading port when the address matches.

4.4.2. Random replacement policy update

As was discussed in the baseline design, the random replacement policy could be improved, as the replacement could be undesired when access to the cache follows a certain pattern. In addition, the policy had to be revisited in order to implement partitioning over it.

The new implementation is enclosed in a new module, called *randomWayReplacement*, containing the logic that selects the next way to evict. This implementation is based on a Linear Feedback Shift Register (LFSR), a shift register whose output only depends on its previous state. It provides a pseudo-random output, which eventually repeats; the larger the number of bits, the longer the cycle length. Figure 4.6 shows the maximum cycle length depending on the number of bits being used.



# of Bits	Length of Loop	Taps
2	3 *	[0,1]
3	7 *	[0,2]
4	15	[0,3]
5	31 *	[1,4]
6	63	[0,5]
7	127 *	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1,023	[2,9]
11	2,047	[1,10]
12	4,095	[0,3,5,11]
13	8,191 *	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]
17	131,071 *	[2,16]
18	262,143	[6,17]
19	524,287 *	[0,1,4,18]
20	1,048,575	[2,19]
21	2,097,151	[1,20]
22	4,194,303	[0,21]
23	8,388,607	[4,22]
24	16,777,215	[0,2,3,23]
25	33,554,431	[2,24]
26	67,108,863	[0,1,5,25]
27	134,217,727	[0,1,4,26]
28	268,435,455	[2,27]
29	536,870,911	[1,28]
30	1,073,741,823	[0,3,5,29]
31	2,147,483,647 *	[2,30]
32	4,294,967,295	[1,5,6,31]

Source: EEtimes: Linear Feedback Shift Registers (LFSRs) [39]

Figure 4.6: LFSR maximum length of loop and needed XOR taps depending on the number of bits

In our case, an 8-bit LFSR is being used, where the output of bits 1,2,3 and 7 is being "XORed" and being used as the input to the shift register. With respect to the previous mechanism, the perceived "randomness" of the component is much higher, preventing excessive replacements in a subset of ways due to certain access patterns much more unlikely.

```vhdl
-- GENERATES RANDOM NUMBER WITH 8-BIT LFSR
randomNumberGen: process(clk, lfsr)
begin

    if(rising_edge(clk)) then

      for i in 1 to 7 loop
          lfsr(i) <= lfsr(i-1);
      end loop;

    end if;

    lfsr(0)      <= lfsr(1) xor (lfsr(2) xor (lfsr(3) xor lfsr(7)));
    randomNumber <= to_integer(unsigned(lfsr(0 to 7)));

end process;
```

Listing 4.2: LFSR random number generation

4.4.3. Partitioning

To partition L2C-lite+ we have followed a way-based partitioning approach. This technique segments the cache into way-size sections and assigns them to one or more masters (usually CPU cores), who will be the only ones allowed to allocate lines in that section. Our design is influenced by Intel's Cache Allocation Technology (CAT) [25], introduced in section 2.2. In a similar way, we define four different groups into which the bus masters (either cores or peripherals) can be mapped. Each group has associated bitmasks representing the cache space into which its corresponding masters can allocate data. Both the mapping of masters to groups and the modification of bitmasks can be done at runtime by any master by writing into the cache's Configuration and Status Registers (CSRs), providing a flexible design that can fit the user's needs and allowing software-defined dynamic partitioning policies to be implemented. Our partitioning mechanism implementation allows to define either overlapped bitmasks, better for performance, or isolated bitmasks, better for safety. A group can also be restricted of allocating lines into the L2 by assigning them 0 ways. This will cause them to do direct-memory operations instead.

Allowing the partitioning mechanism to be easily changed by software opens the door to other dynamic partitioning approaches that can build on top of the hardware implementations and adapt to the state of the cache to fully optimize the performance of the available hardware.

pRandom

No information was found about the implementation details of partitioning over a random replacement policy, so a novel approach was taken. The previous modularization of the random way selection, explained in section 4.4.2, significantly easied the design and implementation of this mechanism.

The basic idea behind our design is using the random number generation provided by the LFSR to shift a pointer each cycle that indicates the way, W, to be replaced. The remainder of the division of LFSR with the number of ways provides the new position of the pointer; this is a cheap operation as the number of ways must be a multiple of two.

Then, partitioning bitmasks defined by the user are used to determine which ways each owner can evict. If the partitioning bitmask in position *W* is '1', that is, the master can evict in that line, a signal that indicates the last chosen way is updated. When a line has to be replaced using the random replacement policy, the line in the last chosen way is the one evicted.

Although this design has not been proven theoretically, its practical implementation works well as the way selection has been observed to be (relatively) uniform.

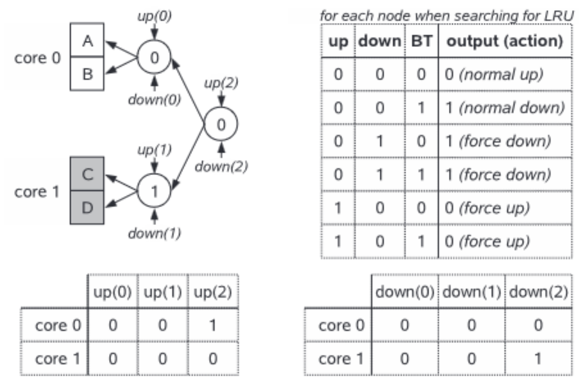
```
waySelection: process(clk, rstn, enable)
variable wayPointer : integer := 0;
begin

    if rstn = '0' then
        bitmask_array <= (others => (others => '1'));
        lastWay       <= (others => ways);

    elsif rising_edge(clk) and enable = '1' then

        if write = '1' then

            -- WRITES BITMASK AND SETS LAST WAY TO UNDEFINED
            bitmask_array(bitmask_select) <= bitmask_write;
            lastWay(bitmask_select)       <= ways;

        else
            -- UPDATES POINTER POSITION
            wayPointer := wayPointer + randomNumber;
            wayPointer := wayPointer rem ways;

            -- UPDATES LAST WAY IF POINTER IS SELECTING VALID WAY
            for i in 0 to numBitmasks - 1 loop

                if bitmask_array(i)(wayPointer) = '1' then
                    lastWay(i) <= wayPointer;
                end if;
            end loop;
        end if;
    end if;
end process;
```

Listing 4.3: Random partitioning mechanism

pLRU

Our implementation of the partitioning mechanism is based on the design proposal by Kedzierski, Kamil, et al. [1]. They propose extending the replacement algorithm with two vectors of bits per way, `pLRU_left` and `pLRU_right` of size $Ways - 1$ (see Figure 4.7). Each bit in these vectors is associated with a parent node of the binary eviction tree and can force the direction of the search down the left or right subtree. If the bit in the left vector is set to '1', that would mean that no ways are assigned to that master in the right sub-tree, so the search must continue down the left sub-tree. Likewise, if the bit in the right vector is set to '1', the search must continue down the right sub-tree. If the bit is set to '0' in both the right and left vectors, we must check the node's value indicating the sub-tree in which LRU way is located (behaving as a normal BT-pLRU scheme). We added one special case to this design: setting both left and right bits in the root node to '1' represents that the master has no ways assigned, leading to a direct-memory operation.

To implement this policy, the update and eviction pLRU functions had to be modified (see listing 4.4). In the update function, the path to the visited node is only updated in



up	down	BT	output (action)
0	0	0	0 *(normal up)*
0	0	1	1 *(normal down)*
0	1	0	1 *(force down)*
0	1	1	1 *(force down)*
1	0	0	0 *(force up)*
1	0	1	0 *(force up)*

	up(0)	up(1)	up(2)
core 0	0	0	1
core 1	0	0	0

	down(0)	down(1)	down(2)
core 0	0	0	0
core 1	0	0	1

Source: Adapting Cache Partitioning Algorithms to
Pseudo-LRU Replacement Policies [1]

Figure 4.7: pLRU partitioning implementation proposed in [1], we call up and down vectors, left and right, respectively, following more standard tree representations

the parts of the tree in which the owner can access ways. In this way, we ensure that the node accessing cannot interfere in the eviction order in ways they do not own. On the other hand, in the eviction function, the search is restricted to the ways that the owner has the power to evict, as explained before.

```
---- PROCEDURE DESCRIPTION:
--          Updates the pLRU bits to indicate which was the most recent access.
procedure pLRU_update (variable pLRU_data_out : out std_logic_vector(0 to ways
   - 2);
    variable pLRU_data                      : in  std_logic_vector(0 to ways
       - 2);
    variable hit_index          : in integer range 0 to ways - 1;
    variable pLRU_left          : in std_logic_vector(0 to ways -2);
    variable pLRU_right         : in std_logic_vector(0 to ways -2))  is
    variable index                          : integer range 0 to ways - 1 :=
        hit_index;
begin
    pLRU_data_out := pLRU_data;
    index := hit_index / 2 + (ways/2 - 1);
    if(pLRU_left(index) = '0' and pLRU_right(index) = '0') then
   if (hit_index rem 2) = 0 then
    pLRU_data_out(index) := '0';
     else
          pLRU_data_out(index) := '1';
      end if;
    end if;
    for i in 1 to log2ext(ways - 1) loop
        if (index rem 2) = 0 then
            index                  := index/2 - 1;
      if(pLRU_left(index) = '0' and pLRU_right(index) = '0') then
              pLRU_data_out(index) := '1';
      end if;
        else
            index                  := index/2;
      if(pLRU_left(index) = '0' and pLRU_right(index) = '0') then
              pLRU_data_out(index) := '0';
      end if;
        end if;
    end loop;
end pLRU_update;

---- PROCEDURE DESCRIPTION:
--          Traverses the pLRU tree to find the "Least recently used" index.
```

```
procedure pLRU_evict(variable pLRU_data : in std_logic_vector(0 to ways - 2);
    variable output_index               : out integer range 0 to ways;
    variable pLRU_left             : in std_logic_vector(0 to ways - 2);
    variable pLRU_right            : in std_logic_vector(0 to ways - 2))
    is
    variable index                      : integer range 0 to ways := 0;

begin
    if pLRU_left(0) = '1'  and pLRU_right(0) = '1' then
    output_index := ways; -- CPU HAS NO WAYS ASSIGNED
    else

      for i in 1 to log2ext(ways - 1) loop
          if (pLRU_data(index) = '0' and pLRU_left(index) = '0') or pLRU_right(
              index) = '1' then
              index := 2 * index + 1 + 1;
           else
                  index := 2 * index + 1;
          end if;
     end loop;

      if (pLRU_data(index) = '0' and pLRU_left(index) = '0') or pLRU_right(
          index) = '1'  then
    output_index := 2 * (index - (ways/2 - 1 )) + 1;
      else
          output_index := 2 * (index - (ways/2 - 1));
      end if;
    end if;
end pLRU_evict;
```

Listing 4.4: pLRU partitioning RTL implementation

4.5 Configuration and Status Registers (CSRs)

New internal registers were added to the cache to see the component's state and control it at runtime. Apart from the partitioning-related registers, already explained in the previous section, new performance counters were added to the cache to see the behavior of the cache in more detail under certain conditions. Two new performance counters have been added: a *dirty eviction* counter that keeps track of the backend writes being done to the next memory level, and a *replacement counter per way*, especially useful to monitor the effect of partitioning setups. Additionally, now the replacement policy is configurable at runtime instead of at compile time by writing into the cache configuration register.

Most of the registers are still located in the control module, but the ones related to partitioning have been implemented in the memory module. However, the existing register interface remains unchanged for the programmer: access to the CSRs can be achieved by reading and writing to the cache address + the corresponding offset. Table 4.3 summarizes the CSRs of the module. A full explanation of the registers' structure and options is available in the technical specification, in Appendix B.

AHB address offset	Register
0x00	Flush enable register
0x04	Access counter
0x08	Miss counter
0x0C	Cache configuration
0x10	Dirty eviction counter
0x14-0x20	Group 0-3 random partitioning bitmask
0x24-0x30	Group 0-3 pLRU left partitioning vector
0x34-0x40	Group 0-3 pLRU right partitioning vector
0x44	Master to group register
0x48	Way 0 replacement counter
0x4C	Way 1 replacement counter
...	...
0x48 + N*4	Way N replacement counter

Table 4.3: L2C-Lite+: Cache updated registers. N stands for the way number.

CHAPTER 5
Evaluation

5.1 Testing environment & configuration

To evaluate the L2C-lite performance we have built a 6-core SoC with an architecture similar to the one proposed in SELENE [40], shown in figure 5.1, which implements GRLIB's NOEL-V cores. NOEL-V is a high-performance embedded processor built for safety-related applications. Our configuration of the core is a pipelined, dual-issue, in-order processor with a private 4-way 16KB L1$ implementing a write-through policy. Six symmetrical cores were used.

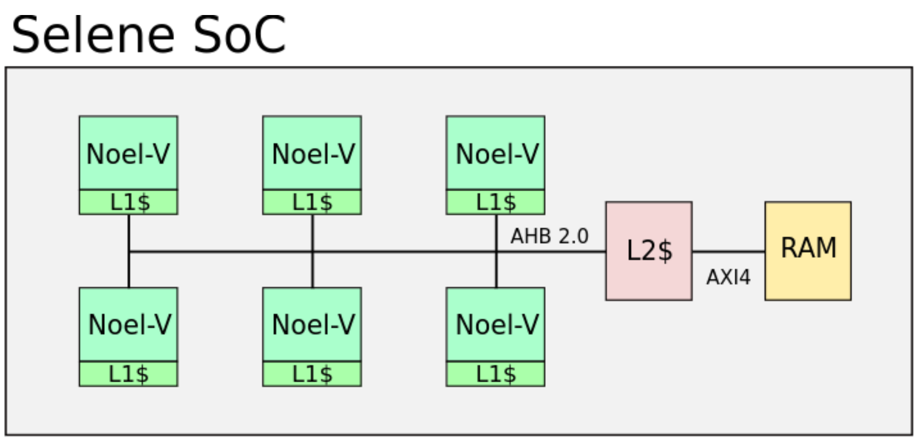


Figure 5.1: Testing environment simplified top view

Both L2C-Lite and the improved L2C-Lite+ will be (separately) plugged as shared L2 caches into the SELENE SoC in order to obtain metrics. Both cache configurations will be 4-way, 512KB with 64B line size. The cores can access the L2 using a 128b-wide shared bus implementing the AHB 2.0 bus. The L2 connects to a DDR4 SDRAM using a 128b-wide AXI4 bus.

5.2 Verification

To verify a module of this complexity, many tests are needed to ensure that the behavior exhibited by the component is correct for any access pattern. This is a really hard problem, widely studied, and any commercial company that develops hardware has a dedicated team, the verification team, that is in charge of ensuring the system's correct functionality. In our case, verification was achieved using simulation and emulating tools.

Simulation is key in the design of hardware components. It allows, in an easy and fast way, the developer to see what is happening inside the component by showing how the internal signals change over time depending on the stimuli provided by the testbench. This is a fundamental step in the development of hardware, as implementing a final testable product is expensive both financially and time-wise. However, it must always be kept in mind that the behavior observed simulation is not always the same as the one exhibited by the final physical component, which can be caused due to an incorrect description or timing problems. In VHDL development, either Siemens' ModelSim or Xilinx Vivado are usually used to simulate designs. In this project, ModelSim is being used.

For this project, the SELENE testbench allowed us to perform full system simulations in both single-core and multicore environments. This testbench allowed us to simulate real (simple) code. An example of the code being used to verify the system is shown in listing 5.1. This code verifies the correct functionality of the internal registers of the cache, executes code in multiple cores, makes use of partitioning, and launches a testbench that accesses the L2, doing both read and write hits and misses. This was the main code used for simulation to ensure everything was working correctly.

```c

int main()
{
  volatile char * memorySpace = malloc(sizeof(char)*L2_WAY_SIZE*L2_ACCESSES*6);

  for(int i = 0; i < 6; i++)
  {
    pointers[i] = memorySpace+i*L2_WAY_SIZE*3;
    #ifdef DEBUG
      printf("Pointer of core %d: %p\n", i, pointers[i]);
    #endif
  }


  #ifdef PARTITIONING
    partitioning(PLRU);
  #endif

    //testAllBitmasks();

  countersReset(); // Reset performance counters
  setupSecondaryCores(6); //Start secondary cores

  mainCoreTestbench(); // Execute testbench

    for(int i = 1; i < 22; i++) //Print all cache registers
    {
    readReg(BASE_ADDRESS + i * 4);
    }

    return 0;
}
```

Listing 5.1: Example of verification code

5.3 FPGA prototyping

After the component was verified in simulation, the system was emulated in an FPGA. The SoC was implemented in an AMD's Virtex UltraScale+ VCU118 FPGA, and results were obtained using bare-metal applications. To launch the different workloads in the SoC the GRMON debug monitor was used.

GRMON is Gaisler's own debug monitor built to verify their cores. It connects to the System Under Test (SUT) with a debug link using either UART, Ethernet, or JTAG debug interfaces. GRMON allows to easily launch code in the SoC cores, change the system configuration, and see the internal state of the components.

GRMON's ability to see the content of memory, insert breakpoints, observe the content of the registers at all times, and print the instruction trace makes testing and debugging code for the designed SoC fast and effective.

In this work, GRMON was used to test the cache design in the FPGA board, debug the component, and launch the evaluation workloads that will be discussed in the next sections.

5.4 Partitioning evaluation

We have developed a resource-stressing kernel to validate the cache partitioning implementation. This benchmark uses two cores that access the same cache set. One of them, the victim, is performing hits to the L2 memory, while the other one, the aggressor, is performing misses. Then, we measure the performance and observe the execution time inflation. However, it is important to keep in mind that since both cores are also using other shared resources, mainly the bus, the execution time inflation is not only the result of inter-task evictions but a combination of the contention suffered across the whole SoC. Thus, to validate we are able to remove inter-task interference effectively we have employed the SafeSU [41] contention monitoring unit. The SafeSU allows us to measure the contention that occurs in the interconnect. Then, if the execution time inflation we observed in the end-to-end execution measurement is greater than the contention measurements provided by the SafeSU this means there we are suffering from inter-task evictions. Note that in this controlled experiment we were able to remove other sources of contention by pre-loading data in the shared cache.

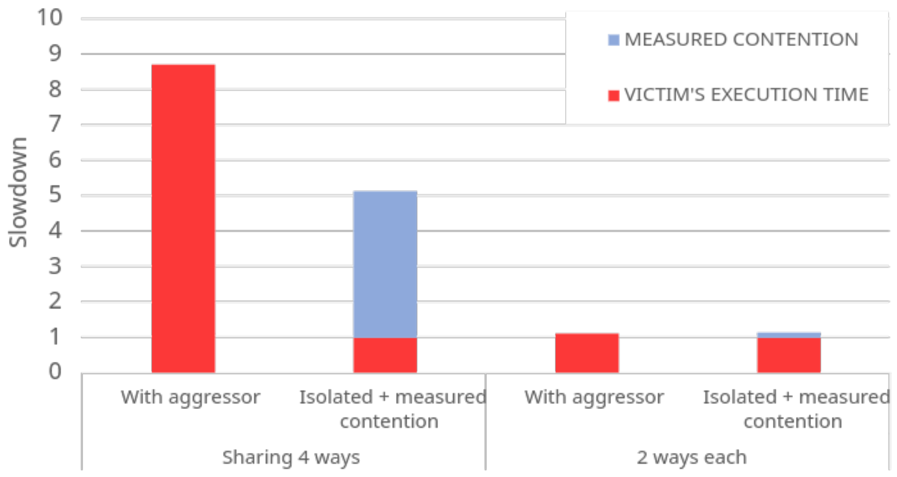


Figure 5.2: Weighed slowdown on victim's execution time of a L2 hit synthetic benchmark with an aggressor causing inter-core evictions

Figure 5.2 shows the results of this test. On the left side, we have the bars representing the shared cache results and on the right side, we have the partitioned cache results. The two bars of the shared cache show that the victim's execution time inflation cannot be explained by the amount of contention measured (in blue) with the SafeSU in the interconnect. The execution time slowdown is around 9X, whereas the slowdown that can be justified with the SafeSU is only 5X. However, when we activate the partitioning we see that the observed slowdown is almost negligible and close to 1 (no contention). It is important to mention that the contention monitored in the shared cache was observed at the interconnect, but is actually a consequence of the cache behavior. When the cache is not partitioned the high amount of misses that need to be served causes the shared resource to be blocked most of the time. As we can see in the partitioned cache results, the contention due to interconnect activity is quite low, which was expected since arbitration contention is generally lower than end-point contention.

5.5 Performance evaluation

To test the performance of the nonblocking capabilities implemented in L2C-lite+, we used EEMBC's Coremark-pro benchmark [42], which is highly popular for measuring embedded architectures' performance. We are executing Coremark-Pro in bare-metal, using one core as a victim, which will run integer benchmarks since there is no open-source floating point unit in GRLIB, and the rest of cores will act as aggressors, causing contention in the cache by executing L2 read and write miss stress workloads. One of the stress workloads being used is shown in listing 5.2.

```c
void cache_access(volatile char * puntero, int offset, int large_offset, int
    num_acceses)
{
   for(int t = 0; t < L1_ACCESSES; t++)
      {
      int k = 0;
      for(int i = 0; i < num_acceses; i++)
      {
         for(int j = 0; j < L2_ACCESSES; j++)
         {
            puntero[t*32+k+j*large_offset];
         }
         k += offset;
      }
     }
}

// mhartid represents the CPU unique id
void secondaryCoreTestbench(int mhartid)
{
  volatile char * my_pointer = pointers[mhartid];

  while(1)
   {
      cache_hit(my_pointer,L1_WAY_SIZE, L2_WAY_SIZE, L2_ACCESSES);
   }

}
```

Listing 5.2: Read miss stress workload code running in aggressors

A TCL script, which can be seen in listing 5.3, is being used to configure the cache and launch the tests into the cache through GRMON.

```tcl

set cores 6
set reps 3
set benchmarks "cjpeg-rose7-preset.riscv core.riscv parser-125k.riscv
    sha-test.riscv"
```

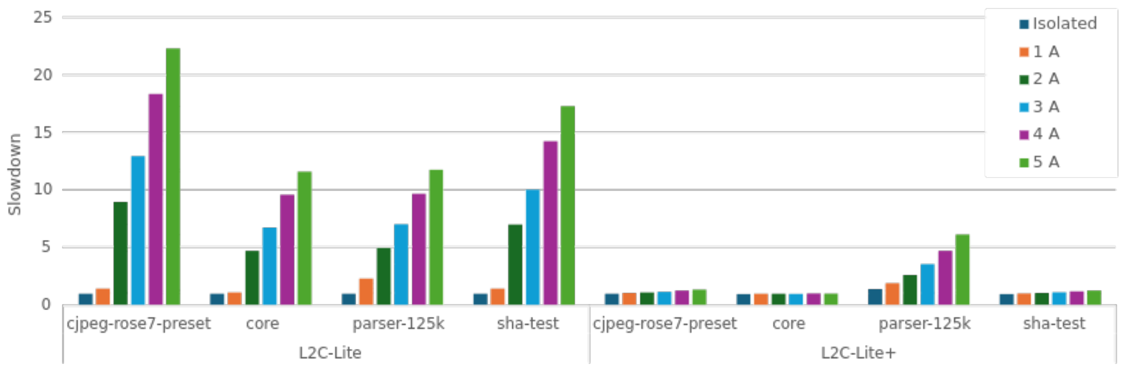
```

cpu disable 1
cpu disable 2
cpu disable 3
cpu disable 4
cpu disable 5

for {set i 0} {$i < $cores} {incr i} {

    cpu enable $i
    puts "$i cores"

    for {set t 0} {$t <  [llength benchmarks]} {incr t} }
        for {set j 0} {$j < $reps} {incr j} {

            puts [lindex $benchmarks $t]
            load [lindex $benchmarks $t]
            run
            wmem 0xffff0000 1    #Flush cache

        }
    }
exit
```

Listing 5.3: Tcl code used to obtain performance results

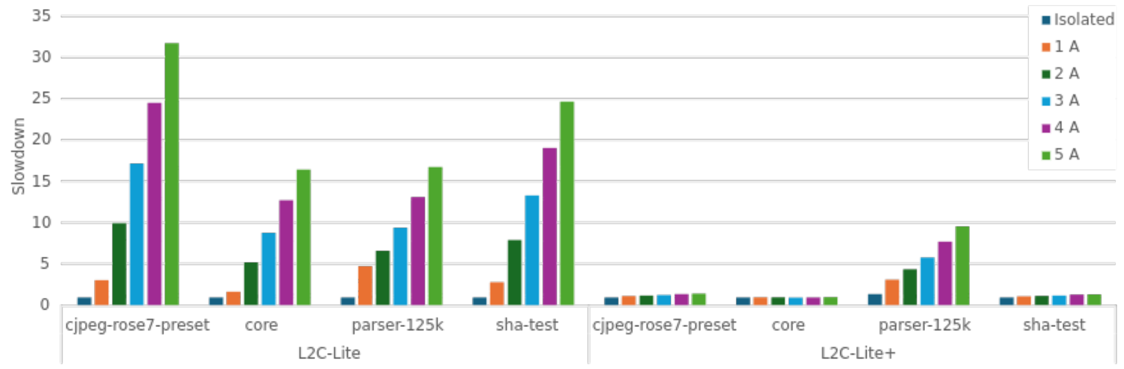
The test results can be seen in figure 5.3. This graph shows how L2C-Lite+ significantly reduces the interference between cores in all cases, even with very memory-intensive aggressors. Our improvement in the parser-125k benchmark is not as high as in the rest of the cases; this is due to the fact that this benchmark is memory-bound and causes many misses in the L2. Misses in the cache still suffer from interference, as only one request is processed in the backend at a time. One of the reasons why the changes are so significant is because of the write-through policy used in the L1 caches. This causes many write-hit accesses into the cache, making the effect of a miss very significant to the rest of the cores sharing the cache.

The slowdown in the original cache is even worse in the case of write miss aggressors because dirty evictions stress the cache's backend even further, blocking the cache for longer. The effect of write miss aggressors in the new cache is still visible in the memory-bounded benchmarks, such as parser-125k, where the slowdown is increased by around 1.6X due to the increased backend demand.

Table 5.1 quantifies the performance improvement of the new cache in multicore environments. We calculate the speedup as the quotient between the slowdowns before and after the changes. We obtain an average speedup of 13.5X for non-memory-bound benchmarks, and a 90% improvement for memory-bound benchmarks in the case of read-miss aggressors; and an average speedup of 18.5X for non-memory-bound benchmarks and an improvement of 75% for memory-bound benchmarks in the case of write-miss aggressors.



(a) Read miss aggressors



(b) Write miss aggressors

Figure 5.3: Weighed slowdown comparison of cache performance based on the number of aggressors (A) doing read and write miss transactions to the L2 before (L2C-Lite) and after the changes (L2C-Lite+). The victim core is running EEMBC coremark-pro integer benchmarks.

Aggessor Operation	**Application**	**Isolated**	**1 Aggressor**	**2 Aggressors**	**3 Aggressors**	**4 Aggressors**	**5 Aggressors**
Read miss	*Cjpeg-rose7-preset*	0,99	1,35	7,963	10,836	14,359	16,103
	Core	1,001	1,122	4,731	6,701	9,41	11,283
	Parser-125k	0,703	1,219	1,885	1,959	2,048	1,913
	Sha-test	1,006	1,41	6,496	8,826	11,668	13,346
Write miss	*Cjpeg-rose7-preset*	0,99	2,68	8,116	13,034	17,467	21,529
	Core	1,001	1,67	5,194	8,73	12,547	16,051
	Parser-125k	0,703	1,506	1,517	1,618	1,697	1,751
	Sha-test	1,006	2,552	6,686	10,619	14,271	17,823

Table 5.1: Single-thread performance speedup based on number of aggressors doing read and write miss operations in the shared cache

CHAPTER 6
Conclusion

In this project, we expose the need for new, more powerful open-source components to meet the demand requirements of state-of-the-art applications. In this context, an extension to an existing L2 cache that tackles both peak and guaranteed performance needs in large multi-core systems has been presented. The effect of the new additions has been tested with benchmarks that exhibit the cache behavior in extreme cases. We prove that in the improved L2 cache, the performance penalty of sharing the cache is almost eliminated in the case of regular workloads and greatly reduced for memory-intensive ones.

The main contributions made to the cache were:

- Transforming the design to a non-blocking cache, imrpoving multicore performance
- Include a partitioning mechanism to improve the QoS of the system for both random and pLRU policies
- Simplified the state machine and optimized the performance of both the AHB bus interface and the operations to the cache memory
- Modularized the design, including a frontend detachable interface making the design easily portable to other systems
- Improved the random replacement policy with a better randomizer preventing unnecessary evictions caused by specific memory access patterns
- Included new performance counters that provide a better view of what is happening inside of the component in real time, and providing the possibility of switching the replacement policy from random to prlu

As it stands, our cache provides an easy-to-use, high-performance solution for multicore systems with unique features in the open-source domain, such as partitioning. Finally, this cache has been used in the SELENE project and will be part of the upcoming project ISOLDE.

6.1 Objective evaluation

This section evaluates the objectives defined in the section 1.2:

- **The component must be open-source:** The L2C-Lite+ design extends Gaisler's open-source design, and as such, it is licensed under the GNU GPLv2 license.

- **The cache must provide performance guarantees, providing some sort of QoS mechanism:** Our cache offers performance guarantees through the presented way-based partitioning mechanism. This mechanism has been explained and verified; its effect on performance has been quantified and its flexible design allows for an easy usage to improve performance in any system with this cache.
- **The component must exhibit good performance in multicore systems:** The cache's multicore performance has been greatly improved thanks to the non-blocking design, which has been proven to be a fundamental feature in shared cache designs.
- **The cache must be scalable:** The cache's great configurability makes it fit for any type of system. In addition, the non-blocking cache architecture has been shown to almost eliminate the penalty of sharing the cache, making it fit for SoCs with a higher core count.

As explained, all of the set objectives defined at the start of the project have been clearly reached, and as such, if we focus on the main goal of the project: *"Creating an open-source shared-cache implementation fit for high-performance embedded systems"*, we can affirm that the scope of the project has been achieved.

Finally, other aspects such as the portability, modularity, and simplicity of the design have also been improved, and as such, they have improved the cache's ease of use and expanded the potential reusability of the project.

6.2 Relation to the degree studies

This project is strongly related to the computer architecture subjects taken during the computer architecture degree: Computer organization (ETC), computer architecture and engineering (AIC) and advanced architectures (AAV). This work talks about a microarchitectural implementation to improve a system's performance using techniques studied during the degree. Cache memories were introduced in ETC and AIC, the shared resources problem was explored in more detail in AAV, where partitioning as a QoS mechanism to hide the system's co-runners effect is introduced.

On the other hand, this project has shown a physical implementation. As such, the high-level components typically used in computer architecture courses must be described in some way, and using a correct methodology and tools. All of this was seen in the Design of Digital Systems (DDS) course. Furthermore, the low-level logic and memory components needed to implement these high-level designs must be understood to effectively verify and debug the design, mainly seen in computer fundamentals (FCO).

Finally, many other concepts and tools learned in the degree have been used during the development of this project: programming languages such as C, python, or assembly; knowledge about software licenses such as the GPLv2 license; or the knowledge of Linux and bash to launch, format and obtain final results.

6.3 Future work

Despite the great improvements in performance shown in this project, further extensions to the design could be done to improve performance:

- AXI frontend support: AHB has been shown to be a very limiting frontend interface. For example, if the cache is going to be used in systems with Out of Order

(OoO) processors, AHB does not provide out-of-order transfer support, defeating the entire purpose of an OoO architecture. Additionally, AHB's active transfer response mechanism significantly affects in some cases the performance of single-core workloads, and can affect the determinism of the component. Providing support in the frontend for AXI would solve all of these problems, and would help to potentially boost the performance of the component and easy the addition of new functionality to the cache.

- Multiple outstanding backend transfers: As it stands, the backend only supports one outstanding transfer. When using AXI in the backend, the system could be optimized to launch multiple backend transfers at a time, reducing the interference shown in the memory-intensive workloads and improving the throughput of the system.
- Prefetcher: A prefetcher would help reduce the misses done in the cache, effectively boosting the throughput of the system. A very simple prefetcher could consist of, when requesting a block from main memory, also requesting the next block, taking advantage of the spatial locality of a normal program.
- Timings: Currently, the cache module is very tight on timings for a frequency of 100Mhz, as the tag lookup and response are being done in the same cycle. This increases the critical path, and prevents the cache from being implemented in higher-frequency systems. The main solution would be to separate the tag lookup and response into two different states; however, this would increase the latency of the component.

Bibliography

[1] Kamil Kędzierski, Miquel Moreto, Francisco J. Cazorla, and Mateo Valero. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[2] SPARC International Inc and David L Weaver. *The SPARC architecture manual*. Prentice-Hall Englewood Cliffs, NJ, USA, 1994.

[3] James Tandon. The openrisc processor: open hardware and linux. *Linux Journal*, 2011(212):6, 2011.

[4] Manolis GH Katevenis, Robert W Sherburne Jr, David A Patterson, and Carlo H Séquin. The risc ii micro-architecture. *Advances in VLSI and Computer Systems*, 1(2):138–152, 1984.

[5] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[6] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fd-soi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[7] Cobham Gaisler. NOEL-V processor. [Online] Available at: https://www.gaisler.com/index.php/products/processors/noel-v.

[8] Christopher Celio, David A Patterson, and Krste Asanovic. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[9] PULP Platform (2023). "Carfield". [Source code] Available at: https://github.com/pulp-platform/carfield.

[10] Florian Zaruba, Fabian Schuiki, and Luca Benini. A 4096-core risc-v chiplet architecture for ultra-efficient floating-point computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–24, 2020.

[11] Abbas Haghi, Lluc Alvarez, Jordi Front, Juan Miguel De Haro Ruiz, Roger Figueras, Max Doblas, Santiago Marco-Sola, and Miquel Moreto. Wfasic: A high-performance asic accelerator for dna sequence alignment on a risc-v soc. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 392–401, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13. Washington, DC, 1975.

[13] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[14] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2019. 6th edition.

[15] Peter J Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[16] Western Oregon University. *Important Programming "Rules of Thumb"*. [Online] Available at: https://wou.edu/las/cs/csclasses/cs161/Lectures/rulesofthumb.html. Last accessed: June 12th 2024.

[17] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2016. 3rd edition.

[18] Andrew Sloss, Dominic Symes, and Chris Wright. *ARM system developer's guide: designing and optimizing system software*. Elsevier, 2004.

[19] Prashant Shenoy and Shashi Singh. Lecture 18: Replacement policies, November 2008. CMPSCI 377: Operating Systems. University of Michigan. [Online] Available at: https://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec18_notes.pdf. Last accessed: June 27th 2024.

[20] University of Virginia. Tree-plru. [Online] Available at: https://www.cs.virginia.edu/luther/3330/S2022/tree-plru.html. Retrieved June 6th 2024.

[21] Carlton Shepherd and Konstantinos Markantonakis. *Background Material*, pages 11–29. Springer International Publishing, Cham, 2024.

[22] AMBA specification 2.0. Technical Report ARM IHI 0011A, ARM Limited, May 1999. [Online] Available at: https://documentation-service.arm.com/static/5f916403f86e16515cdc3d71.

[23] Introduction to AMBA AXI4. Technical Report 102202, ARM Limited, August 2020. [Online] Available at: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/102202_0100_01_Introduction_to_AMBA_AXI.pdf?revision=369ad681-f926-47b0-81be-42813d39e132 Last accessed June 19th 2024.

[24] Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2), may 2017.

[25] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[26] Intel Corporation. Intel® resource director technology (intel® rdt). [Online] Available at: https://eci.intel.com/docs/3.0.1/development/intel-pqos.html. Last accessed: June 27th 2024.

[27] Khang T Nguyen. Introduction to memory bandwidth allocation. Technical Report 659747, Intel Corporation, December 2019. [Online] Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html Last accessed: June 17th 2024.

[28] Khang T Nguyen. Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. Technical Report 672631, Intel Corporation, November 2016. [Online] Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html Last accessed: June 17th 2024.

[29] Amd64 technology platform quality of service extensions. Technical Report 56375, Advanced Micro Devices, Inc., February 2022. [Online] Available at: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf Last accessed: June 27th 2024.

[30] Arm dynamiq shared unit technical reference manual r3p0: L3 cache partitioning. Technical report, ARM Limited. [Online] Available at: https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/l3-cache-partitioning. Last accessed June 27th 2024.

[31] Thomas B. Preußer, Martin Zabel, Patrick Lehmann, and Rainer G. Spallek. The portable open-source ip core and utility library poc. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2016.

[32] João V. Roque, João D. Lopes, Mário P. Véstias, and José T. de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, 14(8), 2021.

[33] César Fuguet. Hpdcache: Open-source high-performance l1 data cache for risc-v cores. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, page 377–378, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 217–232, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Jiri Gaisler. An open-source vhdl ip library with plug&play configuration. In Renè Jacquart, editor, *Building the Information Society*, pages 711–717, Boston, MA, 2004. Springer US.

[36] Frontgrade Gaisler AB. *GRLIB IP Library User's Manual*, 2024.1 edition, Apr. 2024. [Online] Available at: https://www.gaisler.com/products/grlib/grlib.pdf.

[37] Måns Arildsson. Development of a light weight l2-cache controller, 2022. [Online] Available at: https://www.diva-portal.org/smash/get/diva2:1641479/FULLTEXT01.pdf Last accessed: June 18th 2024.

[38] Jiri Gaisler. A structured vhdl design method. *Fault-tolerant microprocessors for space applications*, pages 41–50, 2011.

[39] Max Maxfield. Tutorial: Linear feedback shift registers (lfsrs) - part 1, December 2006. EE Times. [Online] Available at: https://www.eetimes.com/tutorial-linear-feedback-shift-registers-lfsrs-part-1/. Last accessed: June 21th 2024.

[40] Carles Hernàndez, Jose Flich, Roberto Paredes, Charles-Alexis Lefebvre, Imanol Allende, Jaume Abella, David Trilla, Martin Matschnig, Bernhard Fischer, Konrad Schwarz, Jan Kiszka, Martin Rönnbäck, Johan Klockars, Nicholas McGuire, Franz Rammerstorfer, Christian Schwarzl, Franck Wartet, Dierk Lüdemann, and Mikel Labayen. Selene: Self-monitored dependable platform for high-performance safety-critical systems. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 370–377, 2020.

[41] Guillem Cabo, Sergi Alcaide, Carles Hernández, Pedro Benedicte, Francisco Bas, Fabio Mazzocchetti, and Jaume Abella. Safesu-2: a safe statistics unit for space mpsocs. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1085–1086, 2022.

[42] EEMBC. Coremark™-pro. [Online] Available at : https://www.eembc.org/coremark-pro/.

APPENDIX A

Relation to Sustainable Development Goals

Sustainable Development Goals	**High**	**Medium**	**Low**	**Not applicable**
SDG 1. No poverty				✓
SDG 2. Zero hunger				✓
SDG 3. Good health and well-being				✓
SDG 4. Quality education		✓		
SDG 5. Gender equality				✓
SDG 6. Clean water and sanitation				✓
SDG 7. Affordable and clean energy			✓	
SDG 8. Decent work and economic growth	✓			
SDG 9. Industry, innovation and infrastructure	✓			
SDG 10. Reduced inequalities				✓
SDG 11. Sustainable cities and communities			✓	
SDG 12. Responsible consumption and production			✓	
SDG 13. Climate action				✓
SDG 14. Life below water				✓
SDG 15. Life on land				✓
SDG 16, Peace, justice and strong institutions				✓
SDG 17. Partnerships for the goals				✓

We consider that this project is specially related to SDG 8 and 9. This work has presented an innovative solution to improve the performance of the industries' computing devices. Furthermore, the presence of open-source components is vital to building prototypes, potentially reducing a project's economic cost.

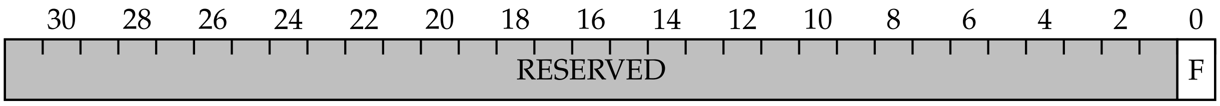
We can also relate it to SDG 4 due to the project's open-source nature, which can be used in educational environments. For example, this project could help to study how to implement a cache in RTL or how partitioning improves the performance and QoS in real systems.

Finally, we see some relation to SDGs 7, 11, and 12 due to the improvement in performance by optimizing an existing design, providing more computing power using the same resources.

APPENDIX B
Technical documentation

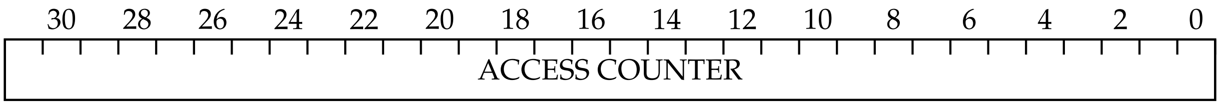
B.1 Configuration and Status Registers (CSRs)

B.1.1. 0x00: Flush Enable Register



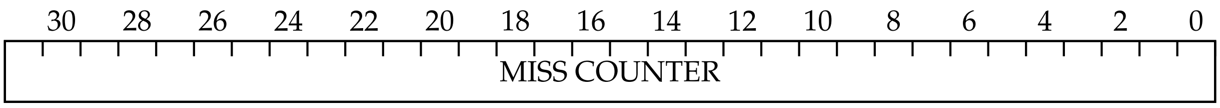
31:1 → RESERVED
0 → Flush enable (F). Write 1 to initiate cache flush[w]

B.1.2. 0x04: Access Counter Register



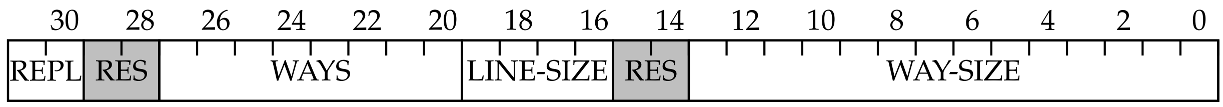
31:0 → Access counter. Write 0 to clear counter[r/w]

B.1.3. 0x08: Miss Counter Register



31:0 → Miss counter. Write 0 to clear counter [r/w]

B.1.4. 0x0C: Status Register

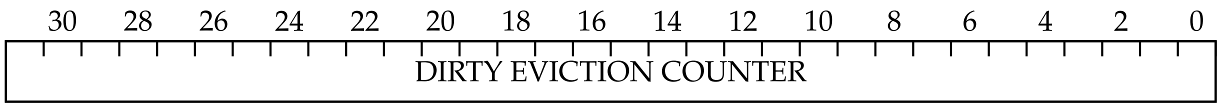


31:30 → Replacement policy(REPL), 0 = Pseudo random, 1 = Pseudo LRU [r/w]
29:28 → Reserved
27:20 → Multi-way configuration (WAYS), Associativity -1 [r]
19:16 → LINE-SIZE [r]

- 0 : 16B
- 1 : 32B
- 2 : 64B
- 3 : 128B
- 4 : 256B

15:14 → Reserved
13:0 → WAY-SIZE - Size in KiB [r]

B.1.5. 0x10: Dirty Eviction Counter Register

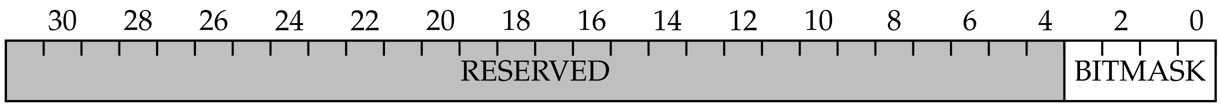


31:0 → Dirty eviction counter. Write 0 to clear counter [r/w]

B.1.6. 0x14-0x20: Group Random Partitioning Bitmask Registers

There are 4 registers of this type, **one for each partitioning group**. Writing a ’1’ into bit N, allows that group (the one whose register we are writing to) to allocate lines in way N.

For instance, **assuming the cache is 4-way set associative** the register would be as follows:

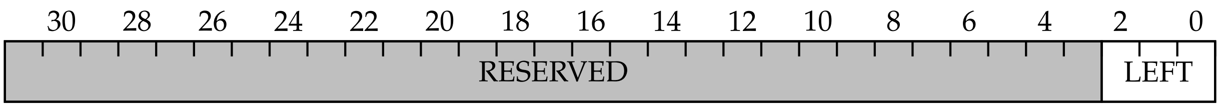


3 → Way 3 allocation. ’1’ Enables allocation of new lines. ’0’ Disables allocation. [r/w]
2 → Way 2 allocation. ’1’ Enables allocation of new lines. ’0’ Disables allocation. [r/w]
1 → Way 1 allocation. ’1’ Enables allocation of new lines. ’0’ Disables allocation. [r/w]
0 → Way 0 allocation. ’1’ Enables allocation of new lines. ’0’ Disables allocation. [r/w]

B.1.7. 0x24-0x30: Group pLRU Left Partitioning Vector Registers

There are 4 registers of this type, **one for each partitioning group**. Writing a '1' into bit N, forces the search down the **left** sub-tree *when searching on node N*. The length of these registers is $W - 1$, being W the total number of ways. The root node is located on the MSB.

Assuming a **4-way set associative** cache:

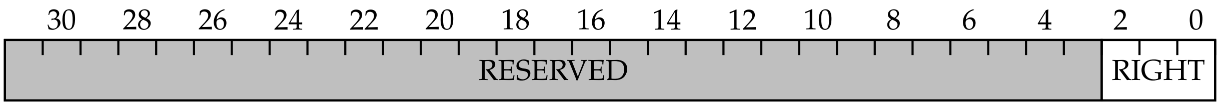


2 → Root node force left. '1' Force left. '0' Not force.[r/w]
1 → Node 1 force left. '1' Force left. '0' Not force.[r/w]
0 → Node 0 force left. '1' Force left. '0' Not force.[r/w]

B.1.8. 0x34-0x40: Group pLRU Right Partitioning Vector Registers

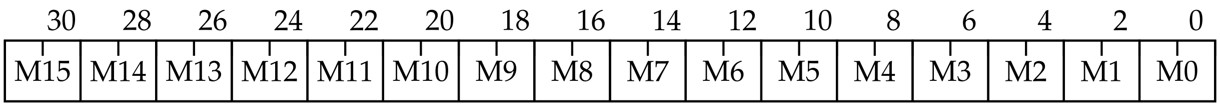
There are 4 registers of this type, **one for each partitioning group**. Writing a '1' into bit N, forces the search down the **right** sub-tree *when searching on node N*. The length of these registers is $W - 1$, being W the total number of ways. The root node is located on the MSB.

Assuming a **4-way set associative** cache:



2 → Root node force right. '1' Force right. '0' Not force.[r/w]
1 → Node 1 force right. '1' Force right. '0' Not force.[r/w]
0 → Node 0 force right. '1' Force right. '0' Not force.[r/w]

B.1.9. 0x44: Master To Group Register



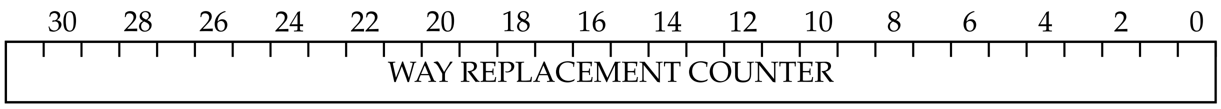
31:30 → Master 15 group.[r/w]
29:28 → Master 14 group. [r/w]
27:26 → Master 13 group. [r/w]
25:24 → Master 12 group. [r/w]
23:22 → Master 11 group. [r/w]
21:20 → Master 10 group. [r/w]
19:18 → Master 9 group. [r/w]
17:16 → Master 8 group. [r/w]
15:14 → Master 7 group. [r/w]

13:12 → Master 6 group. [r/w]
11:10 → Master 5 group. [r/w]
9:8 → Master 4 group. [r/w]
7:6 → Master 3 group. [r/w]
5:4 → Master 2 group. [r/w]
3:2 → Master 1 group. [r/w]
1:0 → Master 0 group. [r/w]

Assigning group to master: Writing the group number into the corresponding master bits assigns that group to that master. For instance, writing 0xc0000001 into the register would assign master 15 to group 3, master 0 to group 1 and the rest of masters to group 0

B.1.10. 0x48-0x48+N*4: Way Replacement Counter Register

There are as many counters as ways in the cache.



31:0 → Way replacement counter. Write 0 to clear counter [r/w]