POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

A Coherence-Capable Write-Back L1 Data Cache for Ariane

Supervisors

Prof. Maurizio MARTINA

Prof. Luca BENINI

Candidate

Michelangelo MICELI

April 2023

Abstract

Multicore processors have introduced new challenges to computer architecture, particularly in ensuring that all cores have access to up-to-date values of shared memory. This is achieved through coherence algorithms that contribute to ensuring consistency across all cores.

The most common coherence protocols implement a *writer-initiated invalidation* approach, in which all distributed copies of data are invalidated when one core requires writing permission. This approach virtualizes memory as a unique shared resource, making coherence invisible even to the strongest consistency model. However, this technique can be complex and requires an inclusive directory or costly communication between cores.

An alternative approach that is gaining popularity is *self-invalidating* data from the private cache at synchronization points, which reduces significantly the protocol complexity at the cost of performance.

In this work, we present a consistency-directed algorithm for CVA6, a 6-stage, RISC-V based CPU developed by ETH Zurich and the University of Bologna. The protocol is aware of the RVWMO consistency model and performs self-invalidation and write-back operations at synchronization points, making the processor compliant with the target memory model. The main goal is to ensure coherence with a completely distributed protocol that neither requires an inclusive directory, nor costly communication between cores. Furthermore, we study the complexity and performance of the design.

Our verification process confirms compliance with the memory model, ensuring the correctness of the coherence protocol. The design is synthesized in GlobalFoundries 22FDX technology, revealing a negligible increase in complexity and no reduction in the maximum frequency. Our performance study illustrates that the performance cost is acceptable when the communication between cores is not excessive.

Acknowledgements

Firstly I would like to express my gratitude to my supervisor Prof. Maurizio Martina, for his numerous advice on universities where develop my thesis based on my interests, and to put me in touch with ETH Zurich.

I am deeply grateful to my esteemed advisors, Nils Wistoff, Samuel Riedel, Robert Balas, and Luca Valente, for their unwavering support, invaluable guidance, and insightful advice throughout my thesis project. Working in such a stimulating academic environment has been a privilege.

I would also like to express my gratitude to Prof. Luca Benini for the remarkable opportunity to pursue my studies at ETH and for his weekly comments and feedback that have helped me to refine my work.

Finally, I would like to extend my appreciation to the IIS and its staff for their assistance and support. Their resources and infrastructure have been vital in the successful completion of my thesis project.

Thank you all for your contributions to my work.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

RVWMO
RISC-V Weak Memory Ordering

ISA
Instruction Set Architecture

SWMR
Single-Writer-Multiple-Reader

PTW
Page Table Walker

DRF
Data Race Free

TSO
Total Store Ordering

EDA
Electronic Design Automation

HDL
Hardware Description Language

LLC
Last Level Cache

FSM

Finite State Machine

Dbs

Dirty Bits

AXI

Advanced eXtensible Interface

SRAM

Static random-access memory

DUT

Device Under Test

TLB

Translation Lookaside Buffer

PLIC

Platform-Level Interrupt Controller

CLINT

Core-Local Interrupt Controller

UART

Universal Asynchronous Receiver-Transmitter

CSR

Control Status Register

SMP

Symmetrical Multiprocessing

SPI

Serial Peripheral Interface

ZSBL

Zero-Stage Bootloader

CPU

Central Processing Unit

FPGA

Field Programmable Gate Array

OpenOCD

Open On-Chip Debugger

JTAG

Joint Test Action Group

GDB

Gnu DeBugger

PVT

Process Voltage Corner

SSG

Global Slow

SS

Slow-Slow

FPU

Floating Point Unit

IIS

Integrated Systems Laboratory

MSHR

Miss Handler Status Register

RCU

Read-Copy Update

LUT

Lookup table

Chapter 1

Introduction

Multi-core processors have become popular for improving performance without excessively affecting power consumption. Since each core is equipped with a private cache containing private copies of shared values, there is a need for cache-coherence mechanisms to provide the abstraction of a unique shared memory.

Most of these mechanisms use MOESI-style protocols to ensure the SWMR invariant and the data-value invariant [1]. Each cache line has a state between Modified, Owned, Exclusive, Shared, and Invalid (whence M-O-E-S-I), and the transition between them is arbitrated through a centralized directory (directory protocols) or broadcast messages between cores (snooping protocols). To ensure that each variable is owned by a single core per epoch, they use writer-initiated invalidations: when a store is executed by a core, all shared copies of the variable are invalidated. Directory-based protocols typically register the owner and readers of each cache line in a shared directory.

This methodology is efficient since it only requires a synchronization mechanism when a variable is accessed between different cores. However, it also brings several issues. To improve concurrency, the control (and verification process) becomes complex with many transient states. It suffers from false sharing, where a cache line is erroneously shared when two cores use different data but within the same line (state is saved at cache line granularity). Additionally, it requires a shared inclusive directory, which adds to the area and power consumption. Most of the complexity arises from the aim of virtualizing the memory as a unique shared resource at every moment, which simplifies compliance even with the stronger consistency model.

Several implementations that implement these protocols can be found in the literature, such as Spandex [2], OpenPiton [3], and BlackParrot [4].

Recently, a wide research area has focused on simpler, directory-less coherence algorithms [5, 6, 7, 8]. A common characteristic of these protocols is that they rely on the DRF assumption of the language-level memory consistency model to perform *self-invalidation* at synchronization points. The main idea is that in a DRF

program, a synchronization operation is executed before accessing a shared variable, and when this happens, shared variables are invalidated to load the updated value directly from the main memory. Then, again relying on the DRF assumption, it is possible to defer flushing dirty data until a synchronization release operation.

The *potential* advantages of these protocols include a much simpler algorithm, which means fewer stable/transition states and ease of verification. Furthermore, there is less latency during store/load operations due to the absence of coherence mechanisms. Additionally, there is no need for a shared inclusive directory because it is not required to track the ownership, which means less area and power consumption.

The main disadvantage of *self-invalidation* is that, since no shared information is available, conservatively, it is necessary to invalidate even the eventually up-to-date lines. This brings two main consequences: executing an acquire or release operation becomes expensive in terms of time and power consumption, and it significantly increases the miss ratio.

Several strategies can be used to mitigate these drawbacks. One strategy is to use private/shared data classification, which can reduce the number of invalidations proportionally to the number of shared variables. This can be easily implemented in software [6] or more accurately in hardware [5]. Some strategies retain some of the classic directory and protocol complexity [9], or use costly write-through buffers [6].

The *self-invalidation* protocols listed usually measure the performance impact by implementing the coherence on top of a simulator (like RADISH [10] and GEMS [11]) that emulates a real processor, with trade-offs between performance and accuracy.

The thesis makes a significant contribution by implementing a low-complexity coherence protocol based on self-invalidation, which is consistency-directed. The implementation is done on CVA6, a 6-stage, single-issue, in-order CPU that implements the 64-bit RISC-V ISA, designed by ETH Zurich and the University of Bologna. The primary objective is to investigate the benefits and drawbacks of the self-invalidation protocol based on the DRF assumption, and its complexity, area, and performance impact on a real processor. The thesis proposes several optimizations inspired by the literature to enhance the baseline implementation. Similar to other related protocols, our implementation's self-invalidation occurs at synchronization acquire, and the write-back of dirty lines is deferred until release operations. The RISC-V ISA provides these types of synchronizations through acquire/release fences and atomics' annotations [12].

The coherence implementation is closely linked to the consistency model, and the verification process aimed to ensure compliance with the RVWMO [12] by stressing the memory model. Custom benchmarks were used to measure the performance cost of the design, which showed a significant dependency on the frequency of

synchronization operations. To enable the use of established benchmarks suitable for multicore systems that require an underlying kernel, we began an initial Linux port onto the modified platform synthesized for FPGA. After a few patches to the Linux kernel, we were able to run the operating system. We compared our implementation with a standard directory protocol running Splash-3 benchmarks, showing a tolerable penalty for most of the programs analyzed. The design was synthesized using GlobalFoundries 22FDX technology, and as expected, the area cost was negligible, and the critical path was not affected.

Structure In chapter 2, we introduce the relevant background knowledge regarding memory consistency, coherence, and the CVA6 processor used in our work. In chapter 3, we analyze and compare our approach with related works. The details of our design are discussed in chapter 4, and its evaluation is presented in chapter 5. Finally, we propose further optimizations in chapter 6.

Chapter 2

Background

In this chapter, we will cover the fundamental concepts related to memory consistency and coherence. We will start by discussing the general concept of memory consistency models and focus on the specific example of the RVWMO in section 2.1 and section 2.2. Next, we will introduce the concept of *consistency-directed coherence* used in this work in section 2.3. Finally, we will provide a brief overview of the platform modified to implement the cache-coherence algorithm in section 2.4.

2.1 Memory consistency

With the advent of multithread and multicore processors combined with out-of-order execution, it became necessary to expose to programmers what behaviour of shared memory systems to expect and instruct implementers on what to provide.

As stated in [1], a *memory consistency model* is "a specification of the allowed behaviour of multithreaded programs executing with shared memory." It defines what values dynamic loads may return. The memory model of a processor answers the following question: *Which are all the allowed values returned by each load?* Or, in other words: *In which order can memory operations from each core be made visible to other cores?*

Let us consider the example in table 2.1: the consistency model defines which are the possible values that can be read by loads L1 and L2.

Core C1	**Core C2**	**Initialization**
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	x = 0 & y = 0

Table 2.1: Simple multithreaded code. Which are the possible outcomes?

To understand a memory model, it is important to clarify some concepts. The program order of each core is the natural order in which instructions are scheduled in memory. On the other hand, the memory order represents the order in which loads and stores are performed (or are visible to the other cores). A load performs when its return value is determined, while a store performs when its value has been propagated to globally visible memory.

The most simple and strong consistency model is Sequential Consistency, which states that the memory order respects each core's program order. If we go back to table 2.1, it means that core C2 will always see S1 before L1 and core C1 will always see S2 before L2. All the allowed executions of this program for a sequentially consistent processor are summarized in fig. 2.1, along with a non-sequential behaviour.

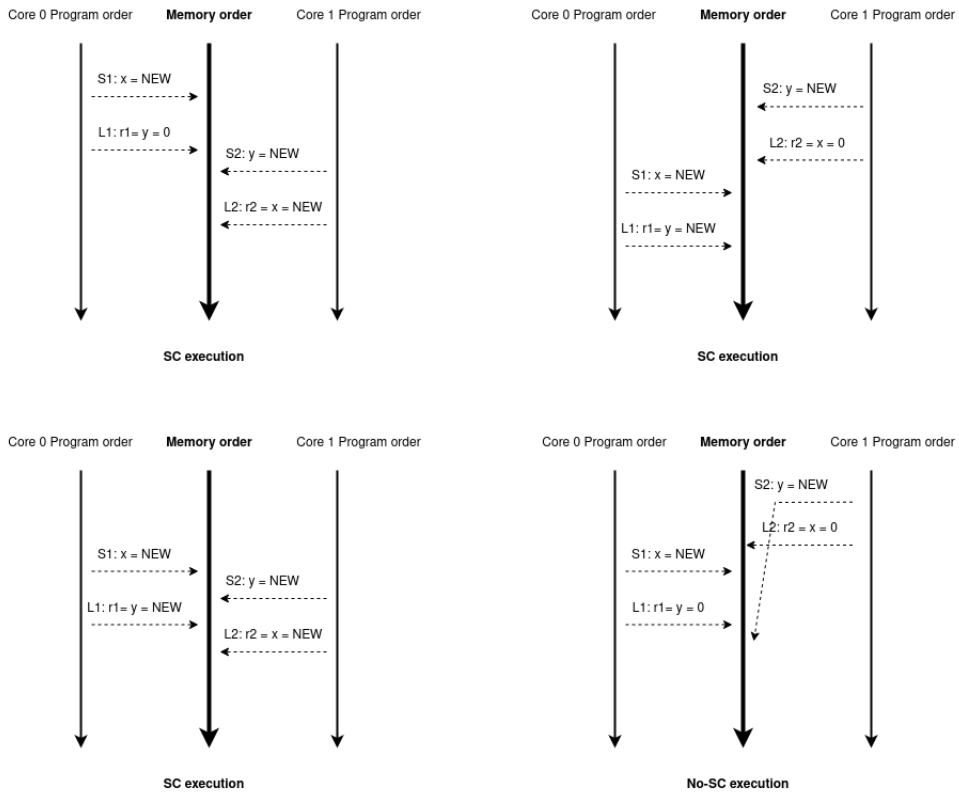

Figure 2.1: Four possible executions of table 2.1

The advantage of a strong memory model like *Sequential Consistency* is that it facilitates the programmability of the processor due to an intuitive expected behaviour. The main disadvantage is the lack of performance. A good model should ease high-performance implementations by giving freedom to the implementors, which is hard to achieve with such a strong model. For instance, processors

commonly use write buffers to hide the cache latency of servicing a store miss. If properly implemented, it is architecturally invisible for a single-core processor. However, when applied to a multi-core one, it can result in reordering between stores and following loads, which is forbidden by *Sequential Consistency*.

Furthermore, it has been recognized that there are many situations in which proper operations do not depend on ordering between stores and loads from different cores. It is clear that when the order is not normally ensured, the instruction set architecture must provide a way for imposing the order, such as with *fence* instructions. An example is shown in fig. 2.2: we don't care about the order in which stores and loads are executed in a critical section as long as they are executed in mutual exclusion.

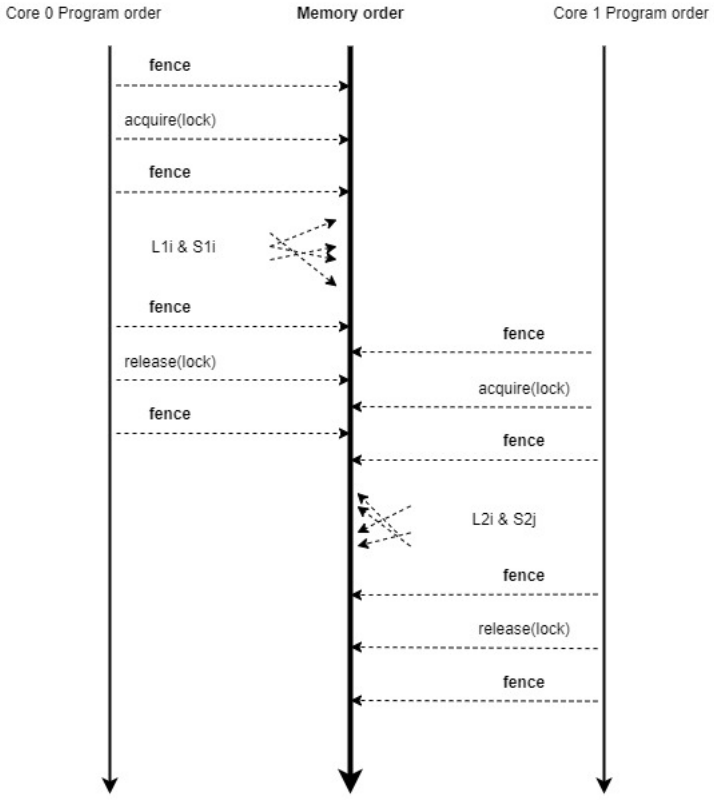

Figure 2.2: No-SC execution of lock-based program allowed by a weak memory model.

Allowing these non-strict behaviours provides opportunities for optimizing the design by implementing non-FIFO coalescing buffers, simpler support for core speculation, and coupling consistency with coherence, as done in this thesis.

2.2 RISC-V Weak Memory Ordering (RVWMO)

An example of a relaxed consistency model is the RVWMO adopted by RISC-V compliant processors, as explained in detail in [12]. It was designed to enable architects to build simple designs or aggressive implementations while being strong enough to support programming language models at high performance. The model is fully described by three axioms and the preserved program order.

Axiom 1 (*Load Value Axiom*)**.** Each byte of each load i returns the value written to that byte by the store that is the latest in global memory order among the following stores:

- Stores that write that byte and that precede i in the global memory order
- Stores that write that byte and that precede i in program order

Axiom 2 (*Atomicity Axiom*)**.** If r and w are paired load and store operations generated by aligned LR and SC instructions in a core h, s is a store to byte x, and r returns a value written by s, then s must precede w in the global memory order, and there can be no store from a core other than h to byte x following s and preceding w in the global memory order.

Axiom 3 (*Progress Axiom*)**.** No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

These axioms, combined with the subset of program order that must be respected within the global memory order, fully define the memory model. It is worth mentioning the explicit ordering mechanisms provided by the RISC-V ISA.

The first possibility is the *fence* instruction; it ensures that all memory accesses from instructions preceding the fence in program order (predecessor set) appear in the global memory order before all memory accesses after it (successor set). It is possible to restrict the predecessor and successor set to only load/store memory accesses.

The second possible mechanism is the use of acquire and/or release annotations for atomic instructions. If an atomic instruction is tagged as an acquire, it imposes that all subsequent memory accesses will be performed after the atomic operation. If it is tagged as a release, all earlier memory accesses must be performed before the atomic operation. These semantics are frequently used for locking and unlocking critical sections. An interesting example is shown in listing 2.1 and in listing 2.2 taken from [12]. It can be seen how *acquire* and *release* operations can be implemented with aq/rl annotations or with fences.

Both codes ensure a correct locking mechanism, but with slightly different ordering requirements: in listing 2.1, the stores and loads inside the critical sections

Listing 2.1: Spinlock asm code with atomics

```
    sd x1, (a1) # Arbitrary unrelated store
    ld x2, (a2) # Arbitrary unrelated load
    li t0, 1 # Initialize swap value.
    again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez t0, again # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
    sd x3, (a3) # Arbitrary unrelated store
    ld x4, (a4) # Arbitrary unrelated load

```

Listing 2.2: Spinlock asm code with fences

```
    x1, (a1) # Arbitrary unrelated store
    ld x2, (a2) # Arbitrary unrelated load
    li t0, 1 # Initialize swap value.
    again:
    amoswap.w t0, t0, (a0) # Attempt to acquire lock.
    fence r, rw # Enforce "acquire" memory ordering
    bnez t0, again # Retry if held.
    # ...
    # Critical section.
    # ...
    fence rw, w # Enforce "release" memory ordering
    amoswap.w x0, x0, (a0) # Release lock by storing 0.
    sd x3, (a3) # Arbitrary unrelated store
    ld x4, (a4) # Arbitrary unrelated load

```

must appear in the global memory order after and before the acquire and release atomics respectively, and no other ordering must be provided. In listing 2.2, it is additionally required that all loads before the "acquire" *fence r,rw* are performed before the critical section, and similarly all stores after the "release" *fence rw,w* appear in memory order after the critical section.

2.3 Consistency-directed coherence

Coherence problems arise whenever there are multiple cores, DMA engines, or external devices that can access a private cache in a shared memory system. If there is at least one write process, this can lead to incoherence when the updated value is not propagated to readers appropriately. Let's suppose that core C1 updates the value of variable x inside its private cache without propagating it to the shared memory. If core C2 is waiting for the new value, without a coherence mechanism, it will spin forever on the stale value inside its private cache without being able to see the new value. A coherence algorithm tries to avoid this kind of incoherent behaviour.

It's important to understand the relation between consistency and coherence shown in fig. 2.3. The memory model defines all the allowed executions of programs in terms of global memory order. This is determined by both the pipeline reordering of instructions and by the coherence algorithm that controls how and when the shared memory communicates with private caches. It follows that coherence can be seen as part of the effort done by implementors to design a processor in accordance with a consistency model.

[1] proposes a useful categorization of coherence protocols related to the memory model.

Consistency-agnostic coherence: Each write operation is made immediately visible to all other cores before returning. This is obtained by imposing that each variable can have either a single writer core or multiple readers (SWMR invariant) and by correctly propagating the updated values between cores (Data-Value invariant). From a consistency point of view, this makes the private cache invisible and virtualizes the memory as a unique shared resource.

Consistency-directed coherence: Writes are propagated asynchronously, allowing for stale data to be observed. However, the coherence protocol, together with the pipeline (fig. 2.3), must adhere to the ordering mandated by the specific consistency model. This more recent category gained prominence during the last decade.

If the former category simplifies the implementation of the memory model by splitting the complexity between the pipeline and the black-box coherence algorithm, the latter allows for pushing performance (or reduces complexity) by mixing together coherence and consistency. This work is focused on the implementation of a consistency-directed coherence protocol.

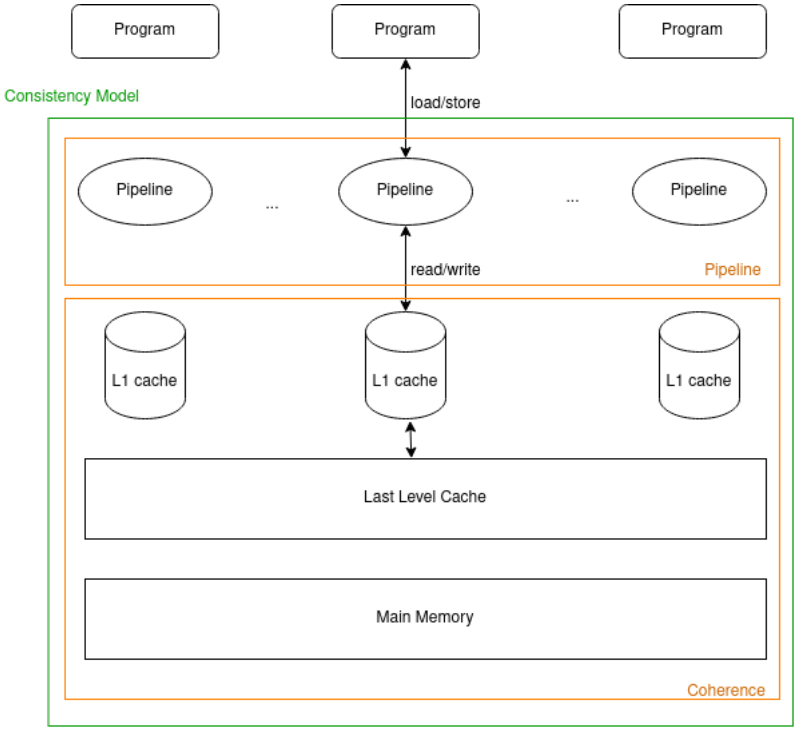

Figure 2.3: How the consistency model interacts with coherence

2.4 CVA6

CVA6 (formerly known as Ariane) is a 6-stage, single-issue, in-order CPU that implements the 64-bit RISC-V instruction set (see fig. 2.4). It fully implements the I, M, A, and C extensions, as well as three privilege levels (M, S, and U) to support a Unix-like operating system. CVA6 has configurable size separate TLBs, a hardware PTW, and a branch prediction unit (consisting of a branch target buffer and branch history table). A more complete description of the microarchitecture can be found in [13].

It's important to note that CVA6 is not fully in order. Issue and commit are performed in order, but to increase IPC, it features a scoreboard that allows the out-of-order execution of data-independent instructions which write back the result in the scoreboard during the execution stage. The access to the data cache is managed by three partially independent entities (the load unit, the store unit, and the PTW) right in the execution stage (see fig. 2.5).

CVA6 contains a tightly integrated data cache. The coherence protocol implemented in this work is developed within the data cache subsystem. The memory is a partially set-associative cache with a pseudo-random eviction policy and write-back writing policy. The cache management is the complex subsystem shown in fig. 2.6.

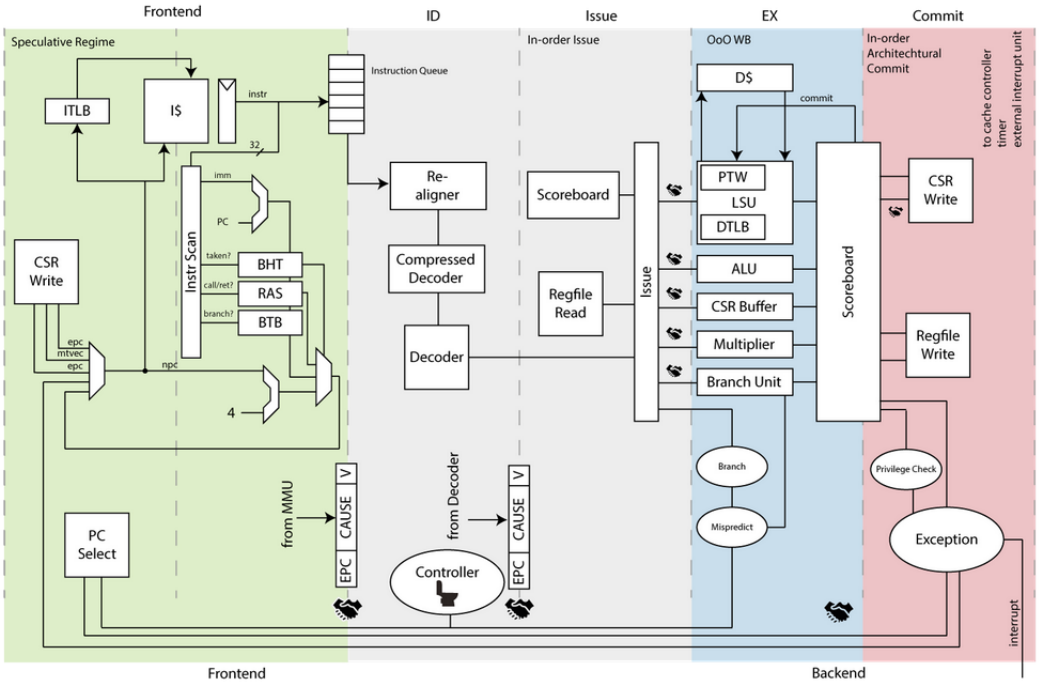

Figure 2.4: CVA6 microarchitecture from [14].

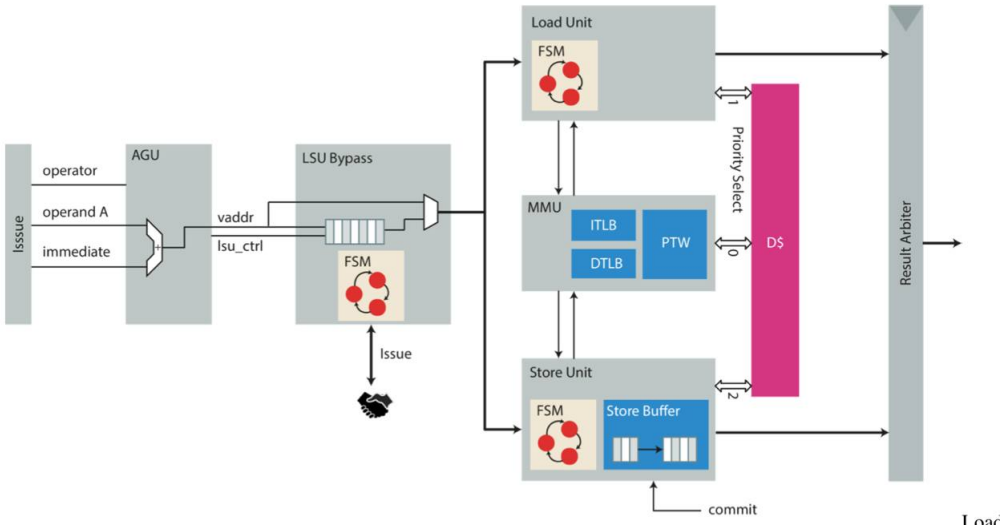

Figure 2.5: CVA6 load store unit from [14]. It interfaces with the data cache.

The three controllers handle requests from the pipeline and send them to the data cache: if it's a hit, the variable is read/written in the cache; otherwise, the request is handled by the miss handler, which is responsible for communicating with the

higher-level shared memory through an AXI bus, an on-chip communication bus protocol developed by ARM.

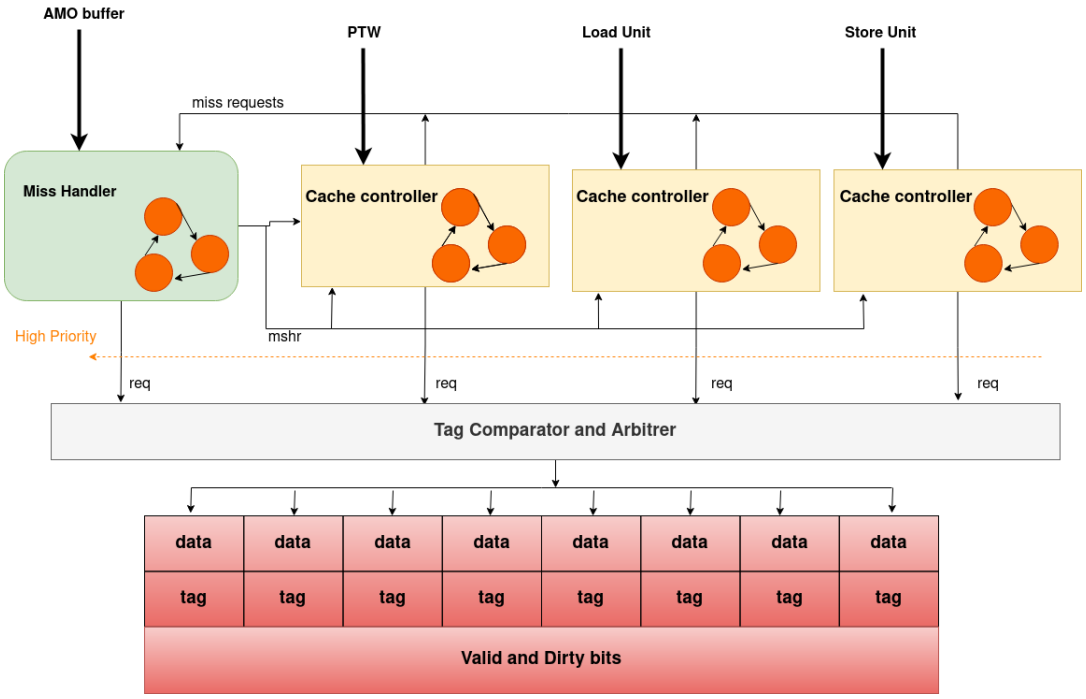

Figure 2.6: Data cache subsystem.

Chapter 3

Related Work

In this chapter, we first introduce several implementations of mostly standard coherence algorithms in section 3.1, and then we discuss alternative *self-invalidation*-based coherence protocols that inspired our work in section 3.2.

3.1 Coherence implementations

OpenPiton is an open-source, general-purpose, multithreaded manycore processor and framework described in the industry-standard Verilog [3]. It leverages the OpenSPARC T1 core with several modifications and implements a directory-based MESI coherence protocol. The communication between the private L1.5 cache and the shared distributed L2 cache is provided by three NoCs, carefully designed to ensure deadlock-free operation. The protocol belongs to the standard consistency-agnostic coherence, used to ensure a strong consistency model like TSO, with all the disadvantages discussed before. The implementation is described with HDL and simulated through an EDA tool, as we did with CVA6. OpenPiton uses Synopsys VCS, while we simulate CVA6 with QuestaSim. The work represents an important enabler for simulation, synthesis, and software exploration but does not target our goal of investigating novel low-complexity coherence protocols.

BP-BedRock is the open-source cache coherence protocol implemented within the BlackParrot 64-bit RISC-V multicore processor [4]. It implements a directory-based MOESIF cache coherence protocol with two different engines, one is FSM-based, and the other is microcode programmable. The work is focused on the advantages of a programmable coherence implementation that allows flexibility in the protocol choice, while we propose a completely different low-complexity algorithm that tries to take advantage of the weak memory model proposed by the RISC-V ISA. BP-BedRock exploits Splash-3 benchmarks to measure performance;

this choice would fit well with our design since Splash-3 improves the well-established Splash-2 benchmarks by removing data races.

Spandex is a flexible coherence interface that can efficiently integrate different coherence protocols without requiring intrusive changes to their memory structure. It can link coherence protocols that differ in invalidation strategies (writer-invalidation vs self-invalidation), write propagation (ownership vs write-through), and granularity (word vs line). It is evaluated on Simics [15] and GEMS [11] simulators. Since the work aims at interfacing coherence protocols and is compatible with self-invalidation algorithms, Spandex research is orthogonal to ours and could allow integrating our coherence with other stronger protocols in bigger heterogeneous systems.

3.2 Coherence self-invalidation protocols

A lot of effort has been devoted to simplifying standard MOESI coherence protocols in recent years. Lebeck and Wood proposed the first self-invalidation-based coherence protocol [16]. The main idea behind this protocol is that it is possible to avoid registering sharers in the directory if each core can recognize when invalidating a cache line at synchronization points. However, as the authors noted, this protocol is compatible only with a weak memory model. The proposed protocol dynamically restricts the self-invalidation through a complex classification performed at the directory. This algorithm is presented as an optimization to reduce traffic on top of a standard directory protocol.

In the direction of scaling the directory, Kaxiras and Keramidas proposed SARC [9], in which readers are definitely removed from the directory by imposing self-invalidation and writer-prediction to reduce indirection. Nevertheless, the protocol still retains the directory and the complexity of standard coherence.

More recently, directory-less protocols were proposed, with minimal complexity and based on self-invalidation [8, 6, 5]. These protocols rely on the DRF assumption and maintain coherence only in the absence of data races. Data races are the culprits of many problems in parallel programs, even with strong coherence implementations and consistency models. Despite this assumption, our proposal satisfies the RVWMO consistency model.

A further common aspect of previous works regards the implementation and evaluation with system-level simulators like GEMS [11], SIMICS [15], and RADISH [10]. These simulators allow modelling multiprocessor systems with a high level of abstraction and performing performance analysis of various system components, such as memory hierarchy, interconnects, and other parameters.

However, our work takes a different direction. We aim to analyze more in detail

the impact of this novel category of protocols, describe the algorithm with an HDL, and integrate it into a real processor (CVA6) based on the RISC-V ISA. We use an EDA (QuestaSim) for digital circuit accurate simulation and verification. The main algorithms that inspired this work are described below.

VIPS-M [6] proposes a directory-less protocol based on private/shared data classification that is performed by the operating system and registered in the TLB. No coherence mechanisms are used for private data, while shared data follow a delayed write-through policy (up to a synchronization point or an MSHR replacement) removing the necessity of registering owners in the directory. At synchronization points, shared data is self-invalidated to remove sharers. The algorithm uses page granularity for private/shared classification, making it sensitive to data access patterns, and it has a limited impact on programs that mainly access shared pages. It also requires underlying software (an operating system) to perform page classification. We use a different writing policy by deferring flush until a release operation, and we propose different lightweight strategies for reducing the cost of self-invalidation.

DIR1-SISD [5] proposes the same idea of private/shared classification, but it is carried out by hardware. The LLC performs the classification dynamically and sends it to private caches, which ensures consistency for a weak memory model by self-invalidation and self-downgrade (cache line flush) at synchronization points. This strategy presents significant advantages compared to VIPS-M: the classification is done at a finer granularity (i.e., a cache line) and improves adaptivity with simple recovery transitions between private and shared classification. On the other hand, it is slightly more complex and requires a shared directory with owner registration (but not inclusive LLC).

Again, we propose different strategies for mitigating self-invalidation, but the DIR1-SISD idea seems orthogonal to our implementation: the same classification mechanism could be integrated into our design as an optimization layer.

Neat [8] is a low-complexity self-invalidation protocol based on the distinction between two synchronization primitives: lock acquire executed before accessing shared variables and lock release after that. The baseline protocol states that whenever an acquire is encountered, the entire cache should be invalidated, and dirty lines need to be written back to shared memory. When a release is executed, all dirty lines in the cache should be flushed and cleaned, but not invalidated.

Due to the high conservativeness of the protocol, several lightweight mechanisms are used to reduce the number of invalidations at synchronization points. These mechanisms include adding the partial invalidate state and using write signatures at the LLC to recognize when the same line is used by more processors.

We use the Neat protocol as our baseline protocol to investigate the impact of self-invalidation for a real processor like CVA6 through digital circuit simulation and verification (QuestaSim), and ASIC/FPGA synthesis (Synopsys/Vivado).

Chapter 4

Implementation

As anticipated in chapter 3, the implemented protocol is inspired by [8]. It provides coherence to the data cache of a multicore processor with minimal modifications, without requiring either a directory or communication between cores and is completely integrated into the cache controller subsystem of each core without the need for a complex LLC control. Like many other self-invalidation protocols, it does not ensure the SWMR invariant, allowing for reading stale copies of data and writing updated values without any notification to other cores. Therefore, we rely on the DRF assumption provided by languages such as C++. A *data race* occurs when two threads access the same memory location, at least one of them is writing, and there are no intervening synchronization operations between them. While they are usually programming errors, data races can lead to unexpected behaviour when coupled with processors that implement weak memory models like RVWMO. Since our target processor, CVA6 implements the RISC-V ISA and its weak consistency model, we chose to implement a low-complex algorithm that provides coherence only for the important category of DRF programs. The protocol assumes the processor's ability to recognize synchronization operations. According to the RISC-V specification manual [12], each fence or atomic instruction is interpreted as a synchronization point. Moreover, our protocol is implemented in SystemVerilog HDL, which is the same language used to describe CVA6.

It follows the detailed description of the protocol in section 4.1 and the required modifications to the CVA6 data cache subsystem in section 4.2.

4.1 Protocol

The protocol is designed for a two-level memory hierarchy, consisting of a private L1 cache per core and a shared higher-level cache or main memory (see fig. 2.3). The protocol is implemented entirely in the private L1 cache controllers and requires

minimal capabilities from the interconnection network or the shared cache controller (table 4.3).

The main characteristic of the protocol is its simplicity. It requires only two states (valid/invalid) and a dirty bit per byte to be set whenever a byte in the L1 cache is overwritten. This functional requirement is important because the protocol maintains the DRF assumption at the same granularity at which data are accessed (e.g., a byte for RISC-V ISA), which helps to avoid overwriting variables with possibly stale bytes.

The protocol is fully described by tables 4.1 to 4.3. When a cache line is invalid, a load or store instruction from the core triggers a miss request from the cache controller, which is sent to the LLC. When the requested data is received, the state of the cache line switches to valid. If the core request was a load, the data is sent to the core. If it was a store, the data overwrites the received cache line, and the dirty bits are updated.

If the cache set is full and a new cache line is needed, an existing line must be evicted. If the line is clean, it can be evicted silently. Otherwise, the data and dirty bits must be sent to the LLC, the state must switch to invalid, and the dirty bits must be cleaned. To ensure that the data-value invariant is maintained, it is important that the LLC controller is able to overwrite only the bytes signaled by the dirty bits.

Finally, self-invalidation is performed at synchronization points, which are identified by the execution of fence or atomic instructions [12]. During self-invalidation, each cache line is invalidated, and any dirty bytes are sent to the LLC and cleared (table 4.1). Every memory request from the core should be stalled until the end of the self-invalidation process.

Normal Execution (NE)	per line, action *if synchronization*: -/SI;
Self-invalidation (SI)	per line self-invalidation and write-back

Table 4.1: Core wide cache control.

4.2 Design

The protocol is described in SystemVerilog and is mostly confined inside the already implemented CVA6 data cache subsystem shown in fig. 2.6.

The cache is accessed by four units: three identical cache controllers and the miss handler.

	Load	**Store**	**Line eviction**	**Self-invalidation**	**Data from LLC**
Invalid (I)	Miss to LLC / -	Miss to LLC / -	N/A	N/A	*If write*: set corresp. Dbs / V; *If read*: - / V
Valid (V)	- / -	Set corresp. Dbs / -	*If dirty*: write-back to LLC and clear all Dbs/I; *if clean*: -/I	*If dirty*: write-back to LLC and clear all Dbs/I; *if clean*: -/I	N/A

Table 4.2: Per line cache controller.

Miss line from core	**Write-back from core**
send data	update line data corresponding to Dbs

Table 4.3: LLC controller.

Each cache controller handles load/store requests from three partially independent units shown in fig. 2.5: the store unit, the load unit, and the PTW. Cache controllers interface the pipeline with the data cache sending a request to the miss handler when a miss occurs.

The miss handler receives requests from the cache controllers and from the core (atomic requests) and connects the data cache to the external world (i.e. the main memory) throughout an AXI adapter: when a miss request comes from one of the cache controllers it sends an AXI message, waits for the response and updates the cache memory; when it receives an atomic instruction it forwards it through the AXI bus.

The tag comparator interfaces the finite state machines (the miss handler and the three cache controllers) with the data cache providing both arbitration and tag comparison for signaling a hit.

The cache memory is n-way set-associative (with parametric n): it is made up of n SRAMs for data and tags, and another SRAM for collecting valid and dirty bits. Both data and tag SRAM word sizes are parametric, as well as the number of lines. In table 4.4 are listed the parameters that dictate the behavior of the data cache.

Our contribution consists of the design of the SRAM for retaining the valid and dirty bits and modifications to the miss handler and the cache controllers to manage coherence (highlighted with italics to distinguish them from what was already implemented).

DCACHE_LINE_WIDTH	Width of a cacheline
DCACHE_SET_ASSOC	Number of cache lines per set
DCACHE_INDEX_WIDTH	Width of the address portion used for addressing the cache, connected to the cache number of lines
DCACHE_TAG_WIDTH	Width of the address portion used for the tag comparison

Table 4.4: Data cache parameters.

4.2.1 Valid Dirty Sram

In order to keep track of the state (valid/invalid) of each cache line and to identify the modified bytes, a valid bit per cache line and a dirty bit per byte is needed. The required storage per data set can be computed using the following formula:

$$x = (DCACHE\_LINE\_WIDTH/8 + 1) * DCACHE\_SET\_ASSOC$$

For example, if the cache has 16 bytes per line, 8-way set-associativity and 256 sets the storage required is 4.352 kB for a 32kB data cache.

Given the large storage demand, a SRAM is used to store this information, as it is more cost-effective than registers. The SRAM is addressed using the same index as the data and tags SRAMs and provides all the valid and dirty bits for a set on each access. Bit enables are used to access the SRAM to ensure that only the correct dirty bits are overwritten.

A high-level diagram of the full data cache is shown in fig. 4.1.

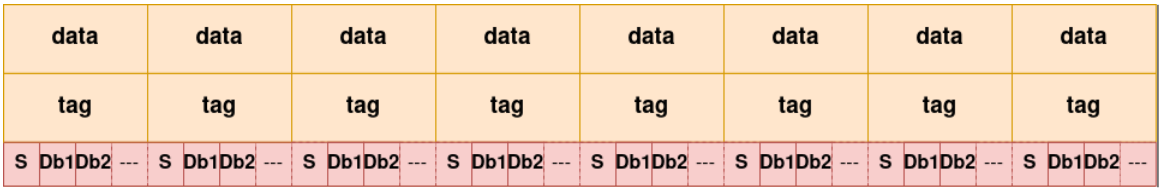

Figure 4.1: Data Cache made up of n tag and data SRAMs and a single Valid Dirty SRAM.

4.2.2 Cache Controller

The cache controller interfaces the core with the data cache. The FSM (see fig. 4.2) can handle three types of requests: load, store, and bypass.

When the pipeline sends a load request, the controller first tries to access the data cache. If it succeeds, it waits for one cycle for the tag and then compares it with the n values from the tag SRAMs. If there is a hit, the read data is sent to the core with a valid signal. Otherwise, a request is sent to the miss handler, and the controller is stalled until the miss handler responds with the data from the shared memory. This behaviour is left unchanged, but care must be taken during the self-invalidation process. *Several transitions added to save cycles between consecutive operations must be suppressed* when the miss handler is executing the self-invalidation to avoid reading stale data. As a result, at the end of each transaction, the transition to the IDLE state is imposed if the flush signal arises, where the FSM stalls until the end of flushing (or self-invalidation). In the best case, a load request has a latency of one clock cycle.

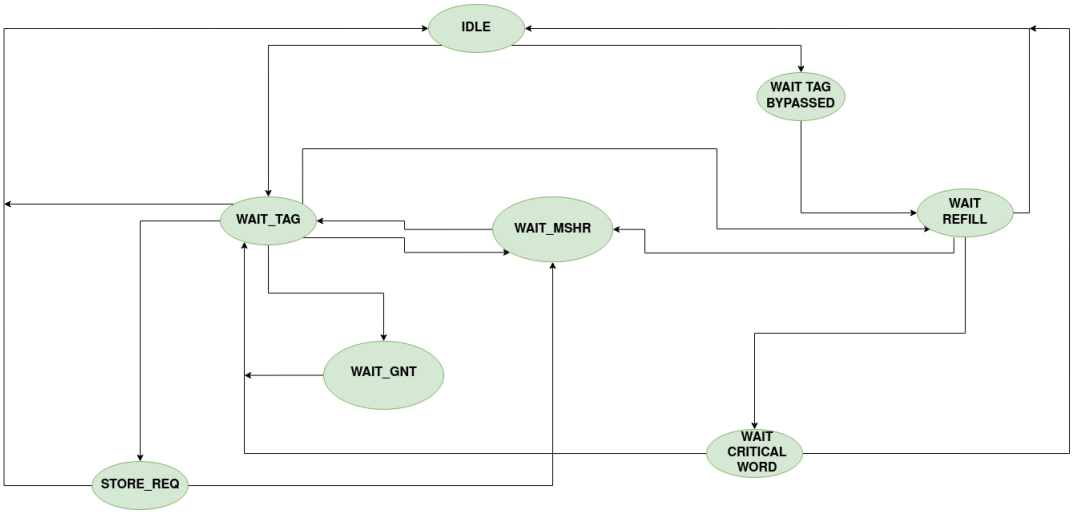

Figure 4.2: High-level cache controller FSM.

When the pipeline sends a store request, one more cycle is required for writing the cache (fig. 4.3). If there is a hit, the data SRAM is overwritten *along with the corresponding dirty bits in the Valid Dirty SRAM*. If there is a miss, the request is forwarded to the miss handler, *which will update the dirty bits*. In the best case, a store request has a latency of two clock cycles.

Finally, a bypass request is treated as a miss and directed to the miss handler. It is important to highlight that before sending a miss request, the cache controller always checks if the miss handler is actually serving a transaction to the same address checking the MSHR and eventually waits for its termination.

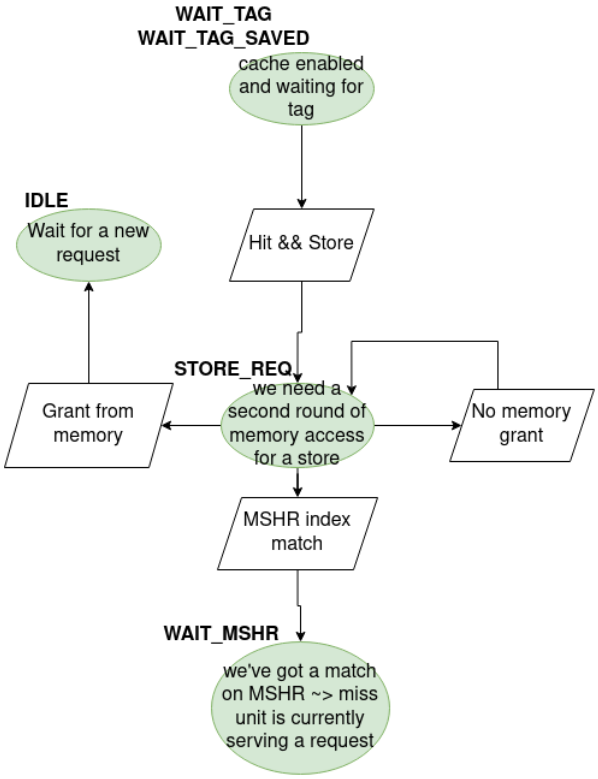

Figure 4.3: Focus on the STORE_REQ state of the cache controller.

4.2.3 Miss Handler

The miss handler interfaces the core with the rest of the system by connecting through an adapter to the AXI bus. Its FSM (fig. 4.4) serves three types of requests: misses, flushes, and atomic operations.

When a cache controller sends a miss request the FSM first reads the valid bits of the corresponding set from the data cache to check if there is space for a new line: if there is no space it picks a pseudo-random cache line from the set, sends the data *together with the dirty bits* through the AXI bus and *clears the dirty bits*. Then it requests the missed cache line (fig. 4.5), waits for the data from the shared memory, and saves the new value in the cache: if the miss was a store, the data is overwritten in the received cache line and *the dirty bits are updated*.

When a fence or an atomic instruction is executed it is interpreted by the miss handler as a synchronization operation and triggers the self-invalidation of the entire cache (see fig. 4.6). The FSM goes through each set *in order to invalidate the cache lines and clear the dirty bits*: if there is a dirty cache line it is sent to the shared memory *with its dirty bits* before being invalidated and cleared; if instead, the entire set is clean it is invalidated in block.

Finally, if the self-invalidation was triggered by an atomic operation, *it is immediately delivered to the AXI bus* and the miss handler waits until the termination of the transaction to return the result to the core.

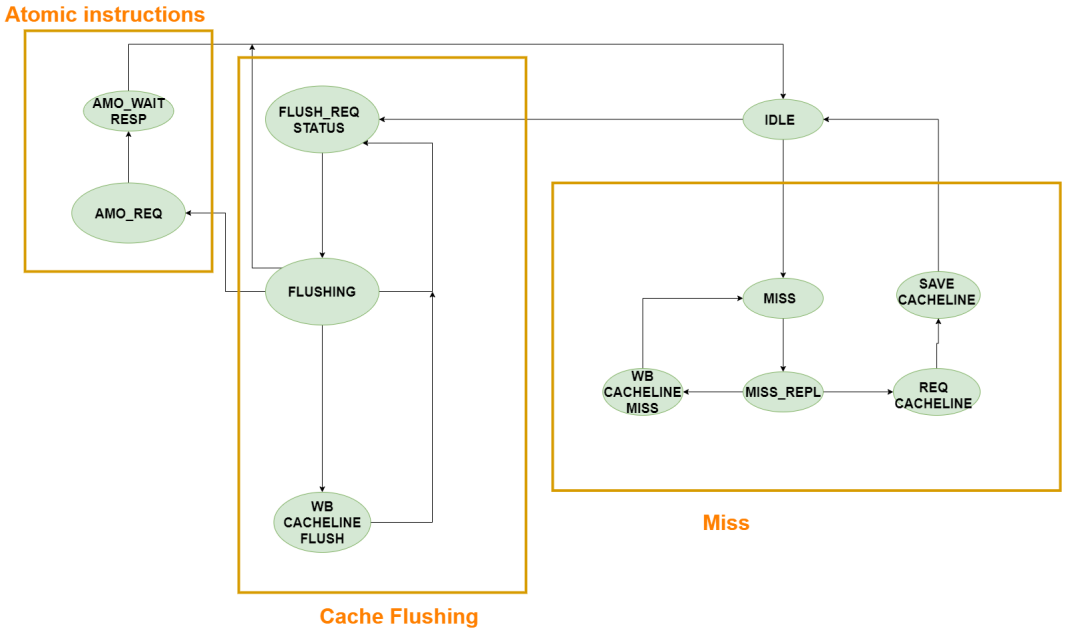

Figure 4.4: High level miss handler FSM.

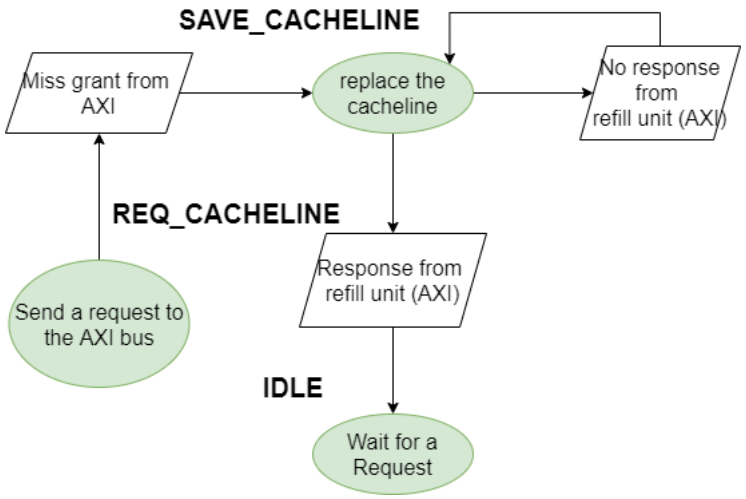

Figure 4.5: Focus on the SAVE_CACHELINE state of the miss handler.

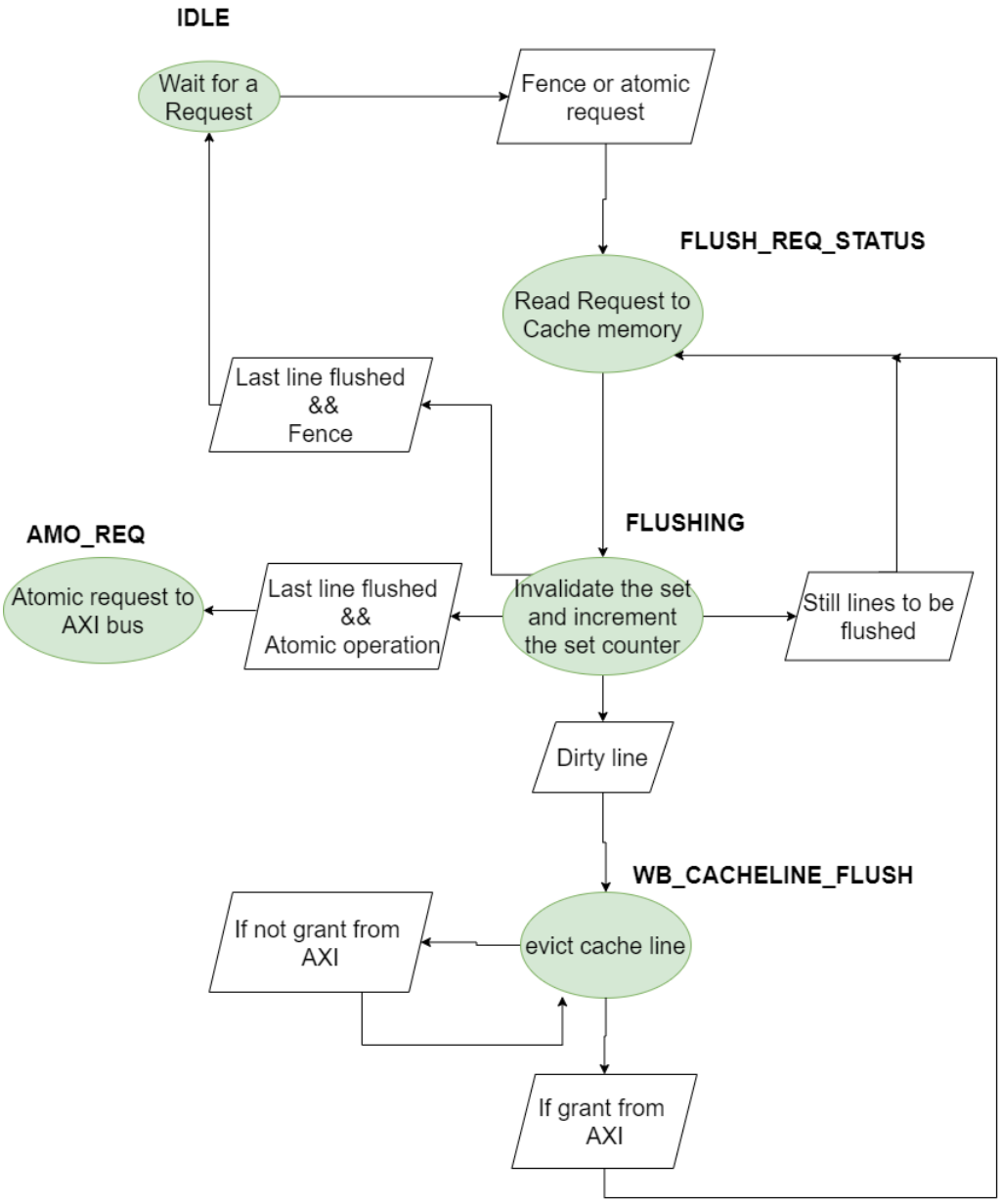

Figure 4.6: Focus on the FLUSHING set of states of the miss handler.

Chapter 5

Evaluation and Results

In this chapter, we first describe the verification procedure adopted in section 5.1. Then, we explain our attempt to boot Linux on a multicore platform that uses our coherence protocol in section 5.2. We study the performance and complexity cost of our implementation in section 5.3, and finally, analyze the results of the CVA6 core synthesis in section 5.4.

5.1 Functional Verification

First, we verify the compliance of the data cache subsystem with the protocol description detailed in section 3.2. This process is described in section 5.1.1.

Secondly, we build a multi-core platform to demonstrate the processor's ability to run parallel programs. Since the algorithm belongs to the category of consistency-directed coherence protocols, it is important to further verify that "the order in which writes are eventually made visible adheres to the ordering mandated by the consistency model" [1], or in other words, that it complies with the RVWMO. The adopted procedure is described in section 5.2.1.

5.1.1 Cache Subsystem

Experimental Setup

To set up the testing environment, we utilized the CVA6 official repository as a starting point [14]. The testbench is described in SystemVerilog and was simulated using QuestaSim (v10.7_b1). The parameters selected for the tested data cache configuration are presented in table 5.1.

DCACHE_LINE_WIDTH	128b
DCACHE_SET_ASSOC	8
DCACHE_INDEX_WIDTH	12b (256 lines)
DCACHE_TAG_WIDTH	44b

Table 5.1: Data cache configuration used for verification.

Testbench

The testbench, depicted in fig. 5.1, includes the data cache subsystem (fig. 2.6) denoted by the red block, and several testbench modules (in yellow) that emulate the requests from the core pipeline. Specifically, *AMO0* simulates the atomic buffer, *WR0* the store unit, *RD1* the load unit, and *RD0* emulates the PTW. The DUT communicates with a testbench memory via an AXI adapter that sends write and read requests. To ensure the correctness of the data cache, a *Shadow Memory* is instantiated. The verification strategy is straightforward: each write request from *AMO0* and *WR0* is forwarded to the shadow memory, bypassing the cache. This makes the values saved in the *Shadow Memory* the expected results for each load. Additionally, an external writer that directly writes to both memories emulates a second core.

To verify the correctness of the DUT, the values read by *AMO0*, *RD1*, and *RD0* during the simulation are compared to those in the *Shadow Memory*. Furthermore, to ensure the correctness of writes, the content of the *Memory* is compared to that of the *Shadow Memory* after the flushing of the entire cache.

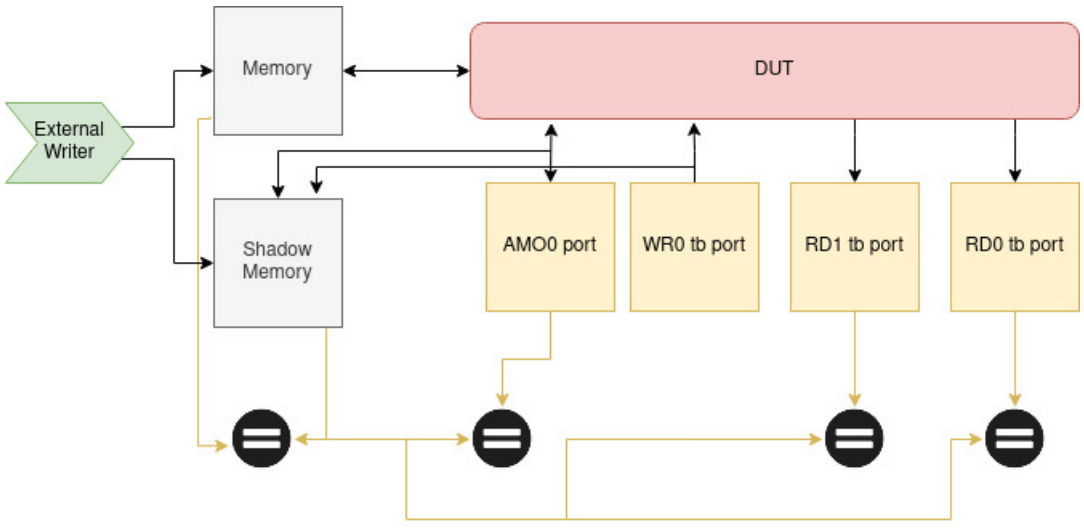

Figure 5.1: Data cache subsystem testbench.

Several tests have already been written to test the DUT, using a mix of sequence types (linear/random/burst), operations (read/write/AMOs), request rates, cachable area sizes, TLB enablement, and invalidation frequency.

In this case, the focus is on the tests designed to verify the coherence protocol, with the main added feature being the support for dirty bits. These tests aim to ensure that the dirty bits are correctly set by the cache controller and the miss handler and are appropriately sent to the shared memory.

The coherence tests are depicted in fig. 5.2. The memory is split into two halves: one is touched only by the external writer, and the other half is written by the WR0 module through the DUT. This division avoids the presence of data races. Memory fragmentation is achieved by selecting different values for one of the four LSBs of the memory address, which makes it easy to restrict address generation.

The various cases in fig. 5.2 show all tested memory divisions. The testing procedure involves the application of 20,000 write requests sent by WR0 to the DUT with random addresses limited to half memory, interspersed with random invalidations. It's essential to highlight that the size of stores is controlled to avoid crossing over the other memory half.

At the same time, the external writer overwrites its half memory, repeating the same operation on the Shadow Memory. Then, the data cache is flushed, and the memory and shadow memory are compared for correctness. If they are not equal, it means that the cache write-back overwrites unused bytes during cache line eviction or invalidation, revealing a protocol error when dirty bits are set or sent.

5.1.2 Multi-core platform

Thanks to the verification process we explained earlier, we can demonstrate that our implementation follows the protocol. However, this raises some important questions: is it sufficient to say that coherence is ensured? Can our design run parallel programs and produce correct results? And if so, what are the correct results? We aim to address these questions in this section.

Since our implementation belongs to the consistency-directed family of coherence protocols, the correctness of the algorithm is closely related to the memory model. Specifically, as long as each program outcome follows the axioms dictated by the consistency model (in our case, the RVWMO), coherence is respected, even if the SWMR invariant is not guaranteed.

This leads us to run executable C programs on a platform with multiple instances of CVA6, collect the output, and verify compliance with RVWMO. The input programs we use for this purpose are called *litmus tests*. These are simple multithread programs designed to test or expose a specific feature of a consistency model, which we explain in more detail in section 5.1.2. There are two main reasons why we chose to use litmus tests: firstly, there are several test suites in the literature

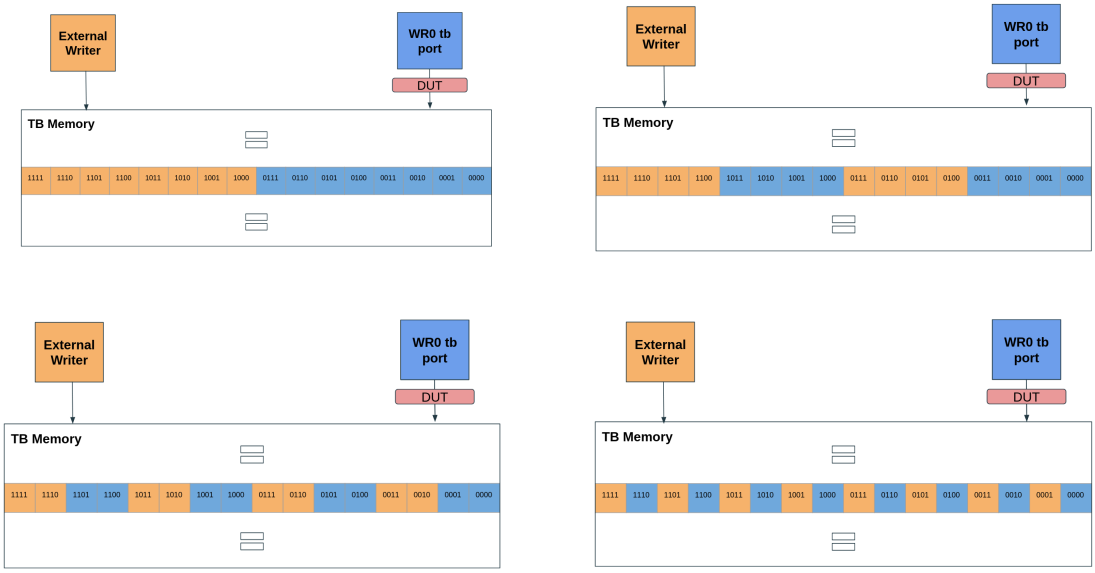

Figure 5.2: Tests added to verify the support for dirty bits. The external writer and the store unit testbench emulator overwrite a different portion of memory.

with thousands of litmus tests that stress the RVWMO more or less exhaustively (we used [17]); secondly, there is a useful tool suite called *diy7* [18] that allows us to run the same tests on top of a consistency model described in the domain-specific language *Cat* and collect all allowed results.

It is important to note that this approach not only tests the correctness of our coherence protocol, but also the CVA6 pipeline and the AXI interconnection network. Together, these determine the global order of memory accesses.

Experimental Setup

To enable the execution of executable files, we developed a multi-core platform in SystemVerilog that instantiated four instances of CVA6 with minimal peripheral support. Fortunately, most of the testing platform was already implemented, and our contribution consisted of the parametric instantiation of four CVA6 cores and their connections with the AXI bus.

It is crucial to take care of the interface between cores and three peripherals: the PLIC, which manages multiple interrupt events generated from the peripherals, the CLINT, responsible for timer interrupts, and the Debug Module. These peripherals must be configured to support a multi-core platform, particularly for FPGA synthesis, where we will need all these peripherals to run an operating system. The design is almost identical to the one used for testing Linux, which is

shown in fig. 5.6.

During the verification process, it is essential to collect program outputs. We accomplish this by using the UART peripheral connected to a SystemVerilog behavioral module that logs the results in an appropriate file.

The executable tests are written in the C language and compiled using *riscv64-gcc* (v8.5.0). The SystemVerilog testbench copies the binary programs into the main memory and starts the multi-core platform. We simulate the complex system using QuestaSim (v2020.1).

Litmus tests

Litmus tests are simple multithreaded programs designed to test and highlight the characteristics of a memory consistency model. An example taken from [12] is illustrated in fig. 5.3. The litmus test code is listed on the left, and the visual representation of one particular execution is shown on the right.

Despite its simplicity, the code demonstrates a subtle feature of the memory model called the Load Value Axiom (see section 2.2), which permits the use of store buffers. Referring to fig. 5.3, the red arrows (rf) represent the dependencies from each store to the loads that return a value written by that store, the yellow arrows (fr) depict the dependencies from each load to co-successors of the store from which the load returned a value, and the green arrows (fence) show the ordering enforced by a fence instruction.

One possible global order that achieves the same results is $(b) \rightarrow (d) \rightarrow (f) \rightarrow (h) \rightarrow (a) \rightarrow (e)$. The stores are propagated to the shared memory at the end of the program, but they are executed first (and eventually stored in a buffer). This specific execution illustrates that the model allows making stores visible to loads of the same core before propagating them to the global memory, enabling the implementation of store buffers. Returning to our coherence protocol, the aforementioned axiom permits us to delay the propagation of stores from the L1 cache to the shared memory.

	Hart 0		Hart 1
	li t1, 1		li t1, 1
(a)	sw t1,0(s0)	(e)	sw t1,0(s1)
(b)	lw a0,0(s0)	(f)	lw a2,0(s1)
(c)	fence r,r	(g)	fence r,r
(d)	lw a1,0(s1)	(h)	lw a3,0(s0)

Outcome: a0=1, a1=0, a2=1, a3=0

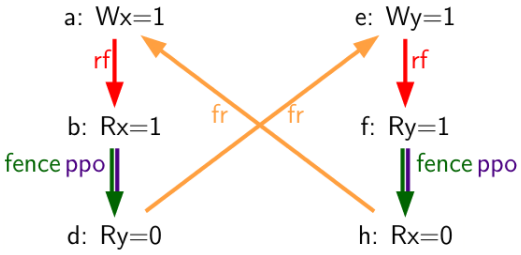

Figure 5.3: A sample litmus test and one permitted execution from [12].

Testbench

In fig. 5.4 is illustrated the testing procedure. A set of almost three thousand litmus tests with four threads at most is taken from [17]. These tests are generated partially using the *diy7* tool suite [18] and partially created manually.

To convert the assembly tests into bare-metal executable files, enhance coverage, and display results, a straightforward verification environment is developed in the C language. The CHERI-Litmus repository [19] is used as a starting point. The main flow of each test program is displayed in figure fig. 5.5. Firstly, the core ID is read from the corresponding CSR to differentiate the execution of different cores. Next, the UART is initialized with frequency and baud rate by Core 0. After this, the specific litmus test is executed 100 times. On each iteration, the variables used are randomly allocated in memory and initialized. The test is executed, and each loaded value is saved. Finally, all outputs are collected together and sent to the UART, which saves them in a properly named file.

Several operations surround the execution of the litmus test to improve coverage: firstly, both cores wait until they are both executing the test function to avoid a constant shift between cores; a random small delay is applied to interleave executions in different ways, and the real litmus test is executed. The loaded values are saved in a proper global structure, and another barrier is applied to let both cores finish at the same time. This ensures that during the last execution, results are displayed only after both cores have terminated. Each executable file is simulated on CVA6 using QuestaSim, and all results are merged together.

Litmus tests are also simulated on top of an axiomatic description of the RVWMO, producing all allowed results saved in a log file. Finally, the hardware results are compared against the model output using *diy7* [18]. The comparison demonstrates compliance with the RVWMO. It is noteworthy to highlight that the coherence protocol respects the memory model despite litmus tests not being DRF programs. This means that the RVWMO is weak enough to allow a self-invalidation coherence algorithm like ours. However, our testing approach based on sample test programs is unable to exhaustively stress the memory model and obtain 100% coverage. For instance, the Progress Axiom states informally that each memory access should be eventually visible to other cores in finite time (see section 2.2). This axiom can't be verified by litmus tests, and in the absence of synchronization operations, our coherence protocol never propagates or fetches updated variables to the shared memory, violating the axiom.

5.2 Booting Linux

Our research on benchmarks shows that an underlying operating system is a typical requirement among multithreaded benchmarks since synchronization is complex to

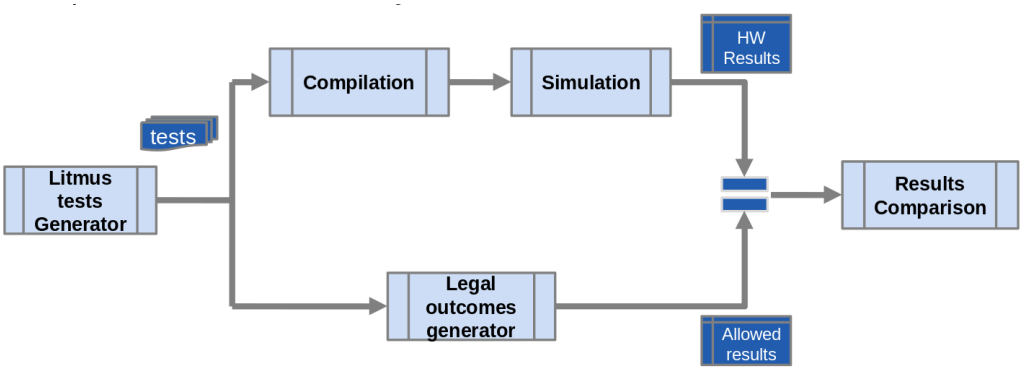

Figure 5.4: Testing procedure to run and verify litmus tests on CVA6.

be implemented and is typically provided by system calls.

As CVA6 was designed to be capable of booting a Unix-like operating system, we attempted to port a SMP version of Linux to our platform, as the ability to run an operating system would be a significant enabler. Our attempt is described in detail in this section.

5.2.1 Experimental setup

In order to be able to run the operating system with reasonable performance we synthesized a dual-core platform with our coherent CVA6 for FPGA. The infrastructure and procedure to produce the bitstream are provided by [14] and our contribution is limited to the instantiation of multiple CVA6 instances, their connections with the AXI bus, and the configuration of peripherals in accordance to the presence multiple cores. A high-level representation of the platform is illustrated in fig. 5.6.

The platform comes with a preloaded Bootrom that includes the ZSBL. This program initializes the UART and SPI, copies the content of the SD card to the main memory, and jumps to the first memory address. To support two cores, we partially modified the code, removing an infinite loop in the main function of the ZSBL that prevented the boot core (which loads memory) from waking up the second core through an interrupt. This allows both cores to jump to the operating system.

The Bootrom also includes the platform device tree, a data structure used to describe the hardware components of a computer system. It informs the operating system about the physical structure of the system, including the number of CPUs, memory and peripheral addresses, and other hardware components. It's crucial to modify the old device tree to indicate the presence of one more CPU and its interrupts. Additionally, the device tree enables us to register the UART as

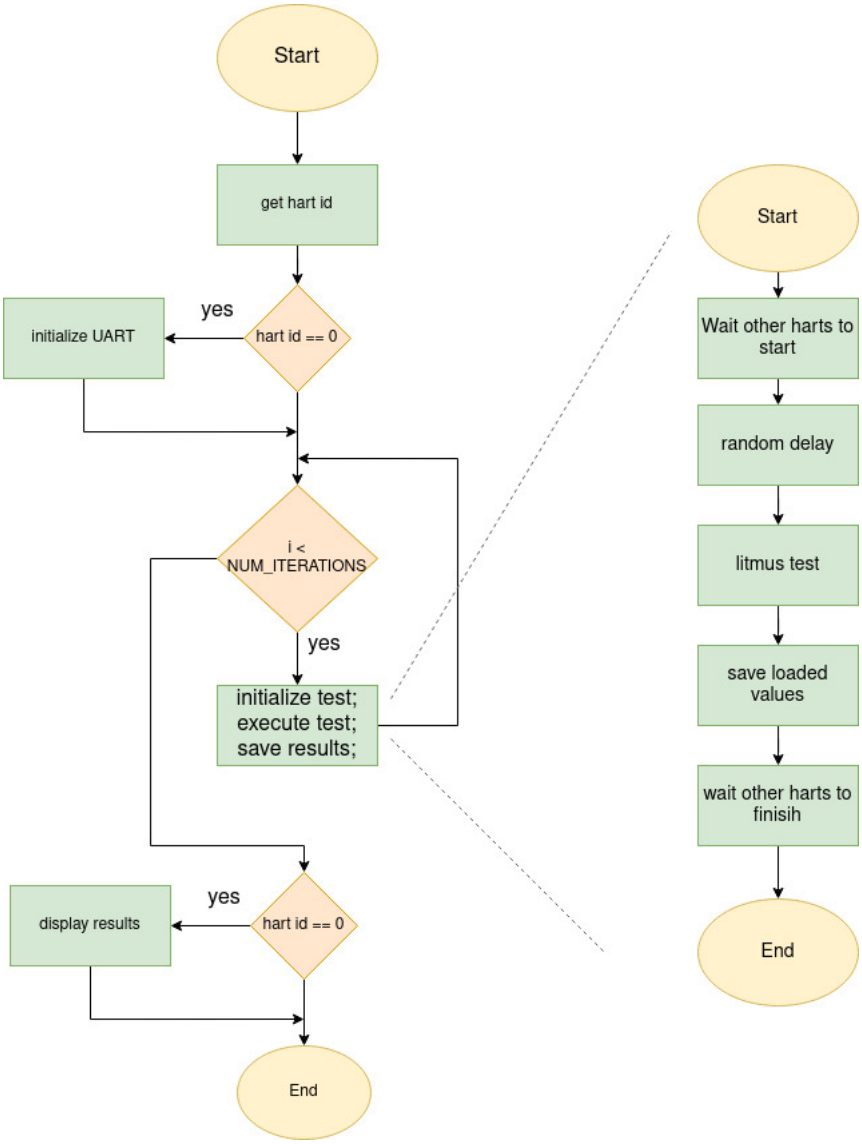

Figure 5.5: Flowchart of the C testing environment to run litmus tests.

standard input and output to communicate with the operating system.

We designed the platform in SystemVerilog and synthesized it for FPGA using Vivado (v2018.2). The resulting bitstream is loaded onto the Digilent Genesys 2 board, which is based on the latest Kintex-7 FPGA from Xilinx.

We used a pre-built Linux image from [20]. Although this distribution is quite old and uses the Berkeley Boot Loader to initialize the system's hardware and set up the environment to execute the operating system, we chose to use it because it is the same distribution used by OpenPiton [3], a multicore platform that can

be configured to instantiate CVA6 cores with a write-through L1 cache. We were motivated to use the same image because it allowed us to reuse some of their work and compare our results. The image is saved on an SD card in a properly formatted manner.

The booting process is analyzed using OpenOCD , which is an open-source software package for debugging and programming embedded target devices in-system. It enables us to connect through JTAG to the hardware debug module of our platform and attach GDB to debug the operating system effectively.

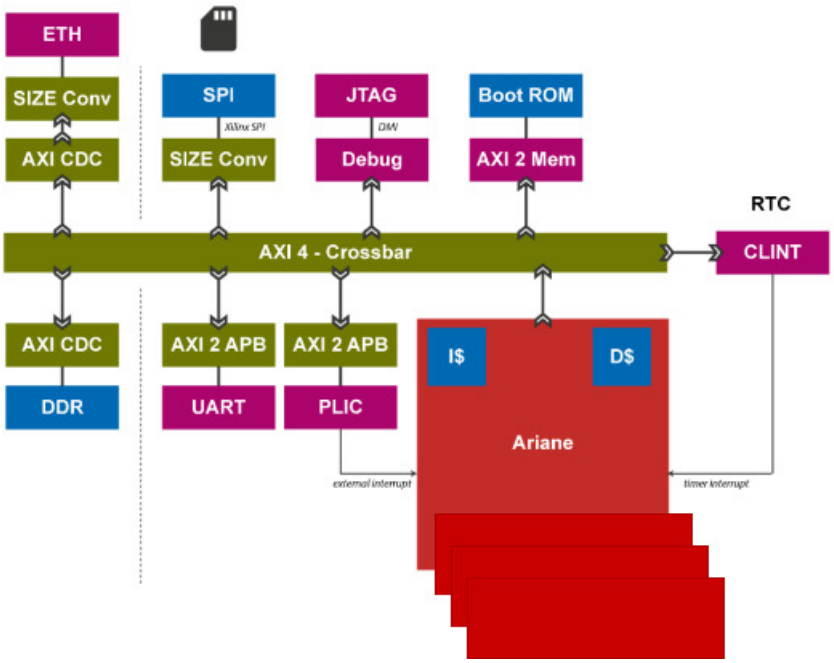

Figure 5.6: Multi-core platform used for running Linux (taken from [14]).

5.2.2 Synchronization issue

After making sure the ZSBL was working properly, we started the execution of the operating system image.

The first obstacle encountered was the inability of waking up the second core demonstrated by the output error "CPU1: failed to come online". Using GDB we found out that the culprit was the presence of a data race (as defined in [1]) in the wake-up synchronization mechanism used by Linux. In listing 5.1 and listing 5.2 it can be seen the approach used for waking non-boot cores: the *_start* module chooses randomly the first hart to boot the kernel and causes the others to spin in a loop waiting for their stack pointer to be written by that main hart. If the other

harts don't come online, the boot one prints the error message we encountered. The data race is due to the absence of a synchronization operation executed by the sleeping hart, as can be seen in listing 5.2. The stack and task pointer are read without the execution of any barriers. The RISC-V memory model doesn't give any assurance on when each store will become visible to other cores (neither if the time is 1000 milliseconds), making the implementation faulty. On the other end, without an operation interpreted as a synchronization our coherence protocol never invalidates variables from its private cache, reading always the stale stack pointer value. We propose a small patch introducing a fence instruction before reading the variables pointed by the stack and the task pointer, between lines two and three of listing 5.2 that solved the bug.

Despite the modification, the operating system remains deadlocked. From the output of GDB, it can be seen that both harts are stalled in the arch_spin_lock() function shown in listing 5.3, but with different values of the *lock* variable. It appears that the unlock operation is not being propagated correctly to the main memory, causing each hart to wait indefinitely for the lock to be released by the other, which believes it has already released it.

This issue could be related to the implementation of the *release* and *acquire* operations in Linux, as shown in listing 5.4. The release of a lock followed by the acquire of the same lock performed by another hart constitutes a data race, as the two operations access the same variables without any intervening synchronization. While this behaviour is compatible with RVWMO, which requires that each store should be visible to other cores in finite time, our implementation does not satisfy the axiom in the presence of data races, and therefore, the operating system cannot be executed.

In listing 5.4 is shown the usage of special memory access macros, namely READ_ONCE and WRITE_ONCE. These macros are used whenever the operating system needs to order memory accesses. To solve the deadlock, we conservatively introduced a full fence at the end of the WRITE_ONCE macro and one at the beginning of the READ_ONCE macro. Additionally, to avoid the same issue, we inserted a full fence before executing the *wfi* instruction inside the *wait_for_interrupt* function. This allows one core to write back all modified variables before entering a wait condition, enabling the other processor to proceed with its computation without waiting for any lock.

The proposed patches allowed us to successfully boot Linux on our platform. However, it occasionally happens that during execution, one core detects a stall and the system gets stuck. In such cases, connecting with GDB can resolve the stall, as the interaction between the debug module and the cores causes the cache flushing, behaving like the execution of a fence instruction. We discovered that these stalls are generated by the RCU synchronization mechanism used by the Linux kernel. The advantage of RCU's approach is that readers do not need to

acquire any locks, perform any atomic instructions, write to shared memory, or execute any memory barriers. However, this optimized synchronization is not compatible with our coherence implementation and should be avoided.

Overall, our platform is capable of running an operating system, even though occasionally the RCU module detects a stall and the system needs to be resumed by connecting with GDB.

Listing 5.1: ___cpu_up() C function from the linux kernel. It wakes up non-boot cores.

```
1     int ___cpu_up(unsigned int cpu, struct task_struct *tidle)
2     {
3     int ret = 0;
4     int hartid = cpuid_to_hartid_map(cpu);
5     tidle->thread_info.cpu = cpu;
6
7     /*
8      * On RISC-V systems, all harts boot on their own accord.  Our
     _start
9      * selects the first hart to boot the kernel and causes the
     remainder
10     * of the harts to spin in a loop waiting for their stack pointer
      to be
11     * setup by that main hart.  Writing ___cpu_up_stack_pointer
     signals to
12     * the spinning harts that they can continue the boot process.
13     */
14    smp_mb();
15    WRITE_ONCE(___cpu_up_stack_pointer[hartid],
16           task_stack_page(tidle) + THREAD_SIZE);
17    WRITE_ONCE(___cpu_up_task_pointer[hartid], tidle);
18
19    lockdep_assert_held(&cpu_running);
20    wait_for_completion_timeout(&cpu_running,
21                        msecs_to_jiffies(1000));
22
23    if (!cpu_online(cpu)) {
24        pr_crit("CPU%u: failed to come online\n", cpu);
25        ret = -EIO;
26    }
27
28    return ret;
29    }
30
```

Listing 5.2: Wait loop for non-boot harts.

```
1     .Lwait_for_cpu_up:
2     /* FIXME: We should WFI to save some energy here. */
3     REG_L sp, (a1)
4     REG_L tp, (a2)
5     beqz sp, .Lwait_for_cpu_up
6     beqz tp, .Lwait_for_cpu_up
7     fence
8
9     /* Enable virtual memory and relocate to virtual address */
10    call relocate
11
12    tail smp_callin
13
```

Listing 5.3: arch_spin_lock() C function from the linux kernel. It waits until the lock is released.

```
1     Wait until the lock is released.}, label={lst:arch_spin_lock}]
2     static inline void arch_spin_lock(arch_spinlock_t *lock)
3     {
4         while (1) {
5             if (arch_spin_is_locked(lock))
6                 continue;
7
8             if (arch_spin_trylock(lock))
9                 break;
10        }
11    }
12
```

Listing 5.4: Release and acquire Linux implementation.

```
1   #define __smp_store_release(p, v)
2
3       compiletime_assert_atomic_type(*p);
4       RISCV_FENCE(rw,w);
5       WRITE_ONCE(*p, v);
6   } while (0)
7
8   ...
9
10  #define __smp_load_acquire(p)
11  ({
12      typeof(*p) ____p1 = READ_ONCE(*p);
13      compiletime_assert_atomic_type(*p);
14      RISCV_FENCE(r,rw);
15      ____p1;
16  })
17
18
```

5.3 Benchmarks and FPGA utilization

In this section, we analyze the performance and area impact of our implementation. For the sake of comparison, we decided to compare our platform with OpenPiton [3], a many-core research framework developed by Princeton.

We made this choice because it is possible to configure OpenPiton to instantiate the same CVA6 core that we used for our work but with a write-through data cache and without any coherence mechanism. Since the two platforms use the same core, it is simpler to compare the overhead required for providing coherence. OpenPiton implements a directory-based MESI coherence protocol through an L1.5 data cache that communicates with the L2 distributed data cache through three NoCs, which are carefully designed to ensure deadlock-free operations. The L1.5 data cache serves as both the glue logic between the AXI protocol used by CVA6 and the three NoCs, and as a write-back layer.

Both platforms were synthesized for the Genesys 2 Kintex-7 FPGA board. Our system is the same one used for booting Linux, with a 4-way 8kB L1 data cache. OpenPiton is configured to use the same L1 cache size, an L1.5 data cache with equal dimensions, and a 64kB L2 cache. Both our platform and OpenPiton use similar peripherals and have a core clock frequency of 50MHz.

Next, we present the performance results in section 5.3.1, followed by the comparison in terms of FPGA utilization in section 5.3.2.

5.3.1 Splash-3 Benchmarks

Our research led us to choose the Splash-3 benchmark suite to analyze the performance of our implementation. This suite requires a Linux-based operating system environment, which we are able to provide, and is well-suited for the study of centralized and distributed shared-address-space multiprocessors. Splash-3 is based on the popular Splash-2 suite of parallel applications but without data races. We need to use Splash-3 since data races, along with the RISC-V weak memory model, can lead to unexpected results and errors.

The results of running these programs on OpenPiton and our coherent platform for two cores are shown in fig. 5.7. It can be observed that our platform has a penalty between 40% and 66% concerning OpenPiton for the barnes, fmm, and radiosity programs. This is due to the presence of one or two orders of magnitude more synchronization inserted to remove data races, which also means more barriers. The execution of memory barriers is costly for our coherence protocol (as explained in appendix A), and this results in a significant penalty. On the other hand, most of the analyzed programs have a small penalty in the range of 0% to 18%. It is also important to note that OpenPiton uses a 64kB L2 cache that certainly improves its performance, even though the DDR3 is fast compared to the core frequency of

50MHz.

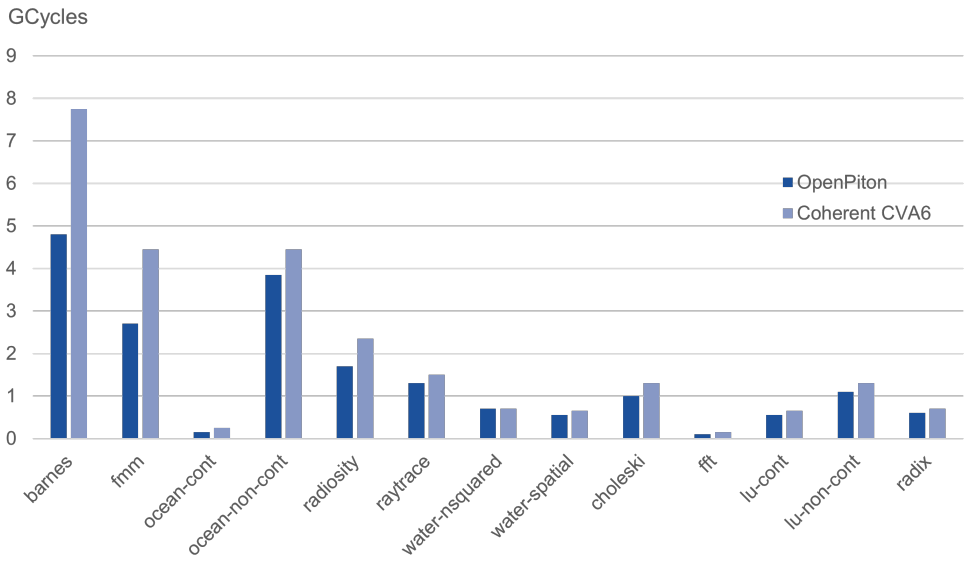

Figure 5.7: Number of cycles to execute each Splash-3 program.

5.3.2 FPGA utilization

We compared the FPGA utilization for three platforms: OpenPiton, the dual-core system that uses CVA6 with our coherence protocol, and the same platform with the baseline CVA6 core without our implementation. We aim to compute the cost in terms of utilization of our protocol compared to the core from which we started. Moreover, we want to compare it with the standard coherence implementation used in OpenPiton. To have a fair comparison, the utilization due to the OpenPiton L2 cache is removed from reports, and only its directory required for coherence is included.

Figure 5.8 shows the results: the darker colours represent the utilization of the two cores, and the lighter ones represent the rest of the system, such as peripherals and interconnections. By comparing the difference concerning the baseline CVA6, it can be seen that OpenPiton uses 53% more LUTs and 5 times more flip-flops than our implementation. Despite the L1.5 cache, OpenPiton has a smaller block RAM usage than our platform due to a more efficient mapping between this specific cache and the board RAM.

Figure 5.9 highlights the block RAM usage of the cores alone. It shows that the significant cost of our implementation is the Valid&Dirty SRAM, which requires

4% (2% per core) of RAM usage, increasing it by 20% compared to the baseline CVA6. It is important to notice that since the *xpm_memory_base* Vivado module used for mapping the Valid Dirty&SRAM doesn't allow to use of bit enables, the memory was oversized by a factor of eight to access each line at bit granularity, as required by our protocol. A more accurate analysis of the area overhead compared to the baseline CVA6 core is done in the following section.

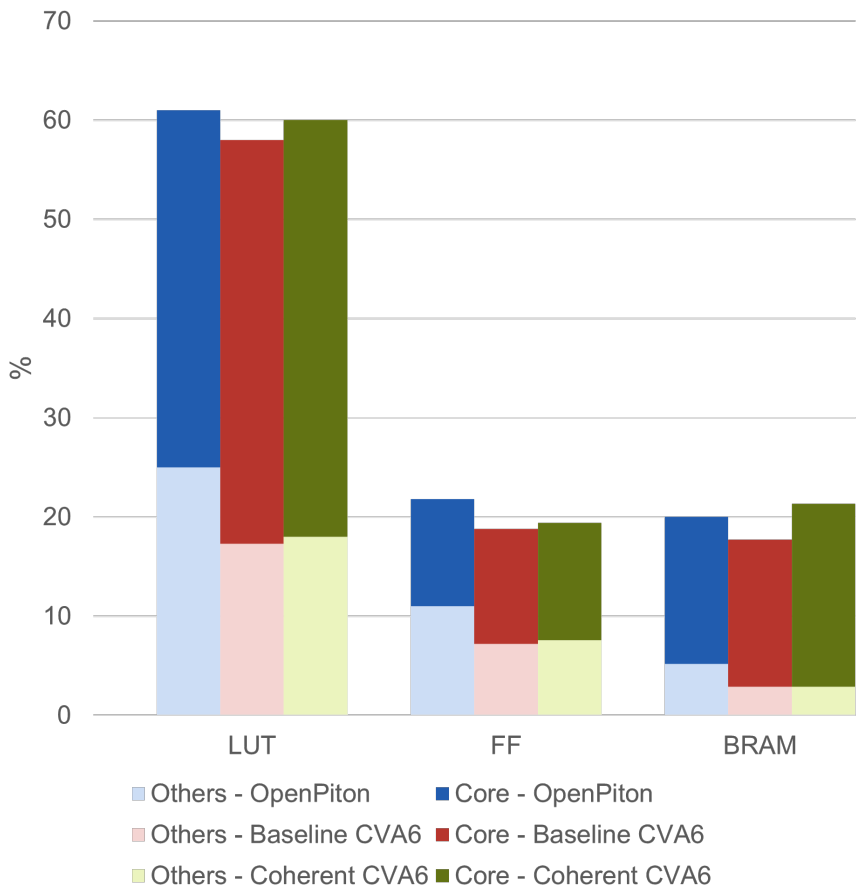

Figure 5.8: FPGA utilization report for three platforms: OpenPiton, the dual-core platform with coherent CVA6, and the same platform without our coherence implementation.

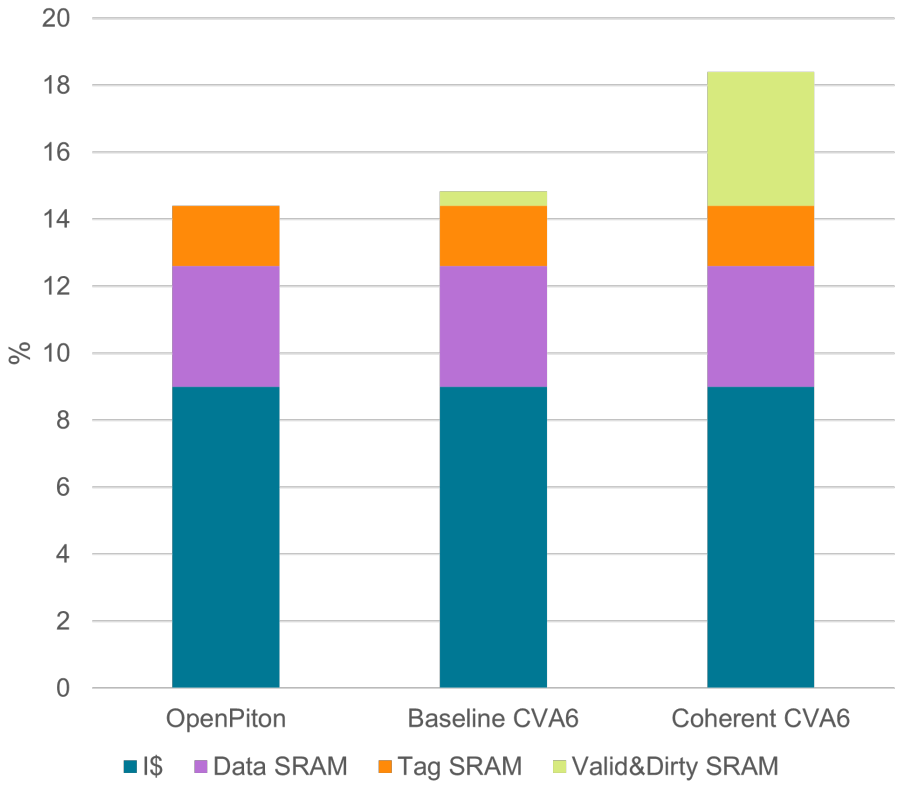

Figure 5.9: Focus of the utilization report on the block RAM usage of the two cores.

5.4 Synthesis

5.4.1 Experimental setup

We used Synopsis (v2022.03) to synthesize the CVA6 core in GlobalFoundries 22FDX technology. The data cache configuration used in this step is described in table 5.1. The macro cells used to map the SRAMs present in the data cache subsystem are listed in table 5.2. It is important to remember that eight data and tag SRAMs are instantiated, as many as the number of ways in a set.

It can be seen that the Valid Dirty SRAM requires two macros because the library doesn't have a cell with a word width of 136 bits, causing inefficiency. A simple solution to save area could be to store the valid bits inside the tag SRAMs that are bigger than required since our tag is 44 bits wide, and use only a macro cell of 128 word width for storing the dirty bits.

The PVT corner synthesized is shown in table 5.3, together with the equivalent area of a NAND gate in *gf22* technology. We decided to operate in the worst

condition; the SSG process corner provided by *gf22* is slightly less pessimistic than the classic SS.

Finally, the clock constraints imposed on the synthesis are summarized in table 5.4

SRAM	Macro Cell	Macro Word Width (bit)
Data SRAM	IN22FDX_R1PH_NFHN_W00256B128M02C256	128
Tag SRAM	IN22FDX_R1PH_NFHN_W00256B046M02C256	46
Valid Dirty SRAM	IN22FDX_R1PH_NFHN_W00256B128M02C256 IN22FDX_R1PH_NFHN_W00256B046M02C256	128 46

Table 5.2: Correspondence between the SRAMs of the data cache subsystem and the macro cells where they are mapped.

Process	Temperature	Voltage	GE
SSG (global slow)	-40C	0.72V	0.199um2

Table 5.3: Synthesis PVT and gate equivalent for the *gf22* technology.

5.4.2 Area and timing results

Area The hierarchical area report from Synopsis focused on the modified modules, is presented in table 5.5. Comparing it with the synthesis of the CVA6 core before the implementation of the coherence protocol, a negligible increment in the total area (1.6%) can be observed. The increase in size is due entirely to the Valid Dirty SRAM, which grows by 44%. This growth is almost equal to the area of the 46-word width macro cell, which could be saved by storing the valid bits inside the tag SRAMs. It may seem like the protocol virtually does not affect the area at all; however, this is not entirely true. In the old version of the core, the Valid Dirty SRAM, used to store a state bit and only a dirty bit per each cache line, is mapped with a macro cell of 128-word width, wasting a lot of space. A smarter design would map the valid bit and the single dirty bit in the tag SRAM, saving area.

By cleverly mapping SRAMs to macro cells for both versions, our design would have an area cost of a 128-word width cell, corresponding to 3.2% of the total core area, which is still negligible.

Timing We computed the maximum frequency of our design and compared it with the old version to determine whether our modifications affected the critical path, considering the process corner shown in table 5.3. We first performed synthesis of the CVA6 core by imposing a clock period of 0ps and gradually increasing it until a slack of almost zero was achieved. The maximum frequency in the range

Period	Uncertainty	Latency
1250ps	100ps	500ps

Table 5.4: Clock constraints for the synthesis.

	Total Area	Miss Handler	Cache Controller	Valid Dirty SRAM
CVA6 without coherence	1.24MGE	13.8kGE	3.16kGE	39.8kGE
CVA6 with coherence	1.26MGE(1.6%)	14.2kGE(3%)	3.16kGE	57.2kGE(44%)

Table 5.5: Area results for the CVA6 core

of (961MHz, 971MHz) was obtained for both cores, with the critical path going through the FPU inside the execution stage.

In terms of area, our modifications resulted in a negligible growth of only 3.2% at worst, with no effect on the critical path's maximum frequency.

Chapter 6

Conclusion and Future Work

We have implemented a low-complexity, consistency-directed coherence protocol for CVA6, a RISC-V core processor described in SystemVerilog. Inspired by recent efforts to simplify coherence and reduce directory cost, our algorithm performs self-invalidation at synchronization points, which can be recognized by the execution of ordering instructions such as fences or atomics with acquire/release annotations.

As a result of our approach, we provide coherence only for programs that satisfy the DRF condition. While this may appear to be a strong constraint, it is worth noting that there is an industry-wide agreement that data races must be prevented. When the processor implements a weak memory model, programmers cannot assume that modifications to shared variables will be immediately propagated to other cores; they can only be certain that they will eventually occur at some undefined point. This makes data races extremely dangerous.

We demonstrated the ability of our coherence protocol to run parallel applications on a multicore platform that was built with multiple instances of CVA6. Despite the DRF requirement, our verification process showed compliance with the target RVWMO except for the Progress Axiom in the absence of synchronization operations. We believe that if we fully comply with the consistency model, i.e. by incorporating a timer to periodically invalidate the data cache every n cycles, we can remove the DRF limitation. Eventually, any errors due to data races would be a result of a programming bug rather than a coherence error.

Our analysis has shown that the frequency of synchronization operations heavily impacts the performance cost of our coherence protocol, with degradation ranging from a few per cent to a significant degree. This effect stems from two factors: the first-order cost, which is related to the time needed to execute synchronizations, and the second-order penalty, which is caused by the conservative self-invalidation that increases the miss ratio. In Section 6.1, we propose design optimizations to mitigate performance degradation.

We compared our implementation with OpenPiton, which implements a directory-based coherence protocol. The Splash-3 benchmark suite showed that the penalty ranges from 0% to 66%, indicating a strong dependence on the frequency of synchronizations.

The ability to run a Linux-based operating system, albeit with the requirement of some small patches, represents a significant enabler for our work. It appears that further work is necessary in this direction to prevent the RCU module from detecting stalls and causing the system to become stuck.

Finally, our synthesis for *gf22* technology demonstrates that the cost in terms of area is negligible, and there is no modification in the critical path, indicating its effectiveness in terms of complexity. As a further step, it would be compelling to conduct a power analysis by annotating switching activity for a suite of benchmarks. This would allow us to understand the power impact of our implementation.

6.1 Design Optimizations

The implementation presented in this thesis can be improved with several optimizations that we have considered during our work. The primary weakness of the protocol is its conservative invalidation and write-back of each cache line at synchronization points, and this section describes ways to reduce this cost.

Distinguish between acquire and release semantics The related work on self-invalidation is based on the common intuition that the purpose of *release* synchronization is to make globally visible all modified variables since other cores may need to access them. As a result, the core that executes the *release* instruction needs to write back dirty cache lines, but since it is the last writer, there is no need to invalidate them. This approach leads to a reduction in the miss ratio due to fewer invalidations.

Fortunately, the RISC-V ISA provides a way to differentiate between acquire and release semantics during the decode stage, using acquire and release annotations, fence predecessors, and successors sets. This information can be sent to the data cache subsystem and used to determine whether to invalidate the cache line during synchronizations.

Partial Invalid state An interesting idea for reducing the number of invalidations at acquire using a lightweight mechanism is presented in [8]. The authors suggest that self-invalidation can be postponed until clean bytes are read. If the load instruction address only dirty bytes, or the operation is a store, there is no need to send a miss request to the shared memory for a DRF program.

To achieve this, the authors introduce a new state called *Partial Invalidate (PI)* which allows for partial invalidation of all clean lines at acquire. Then, if a clean byte is addressed by the core for a load operation, a miss request is sent to the shared memory. This approach could significantly reduce the number of write-backs, improving both the synchronization overhead and the hit ratio.

Implementing this feature would require adding one bit per cache line to encode three states (I/V/PI) and some modifications to the cache controller unit to send a miss request when a clean partial-invalidated byte is read.

Write Signatures Another idea proposed by [8] suggests avoiding invalidation when a cache line is not updated in the LLC by any other core since the last self-invalidation. To implement this, *write signatures* need to be stored in the LLC for each core. These signatures are sent to the core executing an acquire and so cleared, and set when another core writes back its cache line to the LLC.

Although this approach reintroduces some complexity in the form of a shared directory, it can be mitigated by using Bloom Filters. Furthermore, this optimization can significantly improve performance when acquire operations are frequent.

Private/Shared data classification A promising alternative to the lightweight optimizations discussed so far is to classify variables as either private or shared [6, 5]. This approach offers the advantage of executing coherence operations only on shared variables, thus significantly reducing the performance cost of self-invalidation.

There are several ways to perform this classification, as discussed in chapter 3. One method involves the operating system, which saves the result in the TLB with minimal complexity but collects information at the page granularity and requires an underlying kernel. Alternatively, this classification can be performed in hardware by implementing a more sophisticated LLC controller that registers the classification in a structure similar to a shared directory and sends it to cores together with data. This approach increases complexity but improves granularity and adaptivity by allowing for dynamic switching between private and shared classifications. Notably, interesting work in this direction is presented in [5].

Appendix A

Performance estimation

As previously mentioned in section 5.2, our research has shown the need for an underlying operating system or a custom synchronization library to run well-known multithreaded DRF benchmarks.

To simplify the process and obtain some performance estimations, we have designed custom C programs with simple synchronization mechanisms that use shared variables. The procedure used to run executable files on a multi-core platform is the same as the one used to run litmus tests (as seen in section 5.1). The C testing environment is very similar, and the data cache configuration has been previously verified (table 5.1).

Our SystemVerilog testbench registers each instruction retired by every core in a log file, along with other relevant information such as simulation time, instruction opcode, program counter value, and the content of used registers. This information has proven extremely useful for debugging program execution and is utilized in this phase to compute the number of cycles required to execute an instruction or function using a Python script.

It is worth noting that the data cache is connected to a behavioral main memory with a latency of one clock cycle via the AXI bus. The latency for accessing the main memory is mainly due to the complex interconnection network that follows the AXI handshake protocol and introduces a random delay toward the main memory between 0 and 15 cycles.

In this section, we will first analyze in detail the time required for synchronization operations alone in appendix A.1. We will then discuss the performance overhead for a custom benchmark in appendix A.2.

A.1 Synchronization overhead

We have designed two simple functions for multithreaded synchronization, namely *acquire* and *release*, which are shown in listing A.1. The *acquire* function attempts to acquire the lock by atomically storing one to the corresponding address. If the lock was free (i.e., points to zero), the function terminates; otherwise, it keeps trying. The atomic swap operation, which reads the previous value of the lock and then stores one, has an acquire annotation to impose the ordering of the following instructions inside the critical section. On the other hand, the *release* function atomically stores zero to the lock. It has a release annotation to ensure the visibility of all prior memory accesses before its execution. This implementation of acquire and release semantics is recommended by [12] and has already been discussed in section 2.2.

Listing A.1: Release and acquire custom implementations.

```
1   void acquire(volatile uint32_t * lock)
2   {
3     asm volatile (
4
5         "li t0,1                    \n"
6         "again:                     \n"
7         "amoswap.w.aq t0,t0, (%0)   \n"
8         "bnez t0, again             \n"
9
10    : /* output operands */
11    : /* input operands */
12      "r"(lock));
13  }
14
15  void release(volatile uint32_t * lock)
16  {
17    asm volatile (
18
19        "amoswap.w.rl x0, x0, (%0)\n"
20    : /* output operands */
21    : /* input operands */
22      "r"(lock));
23  }
24
```

We aim to compute the number of cycles required to execute the synchronization functions described earlier, which is almost the same time required for executing the atomic RISC-V instructions inside them. This can be easily accomplished by generating an executable file that uses the *acquire* and *release* functions and

simulating it with QuestaSim.

During simulation, the *amoswap.aq/rl* instructions are captured by our coherence protocol, which triggers the invalidation of the entire data cache and the write-back of dirty lines. The time required for invalidating a clean set can be extracted from the miss handler FSM in fig. 4.4 and is equal to two cycles. Each dirty cache line causes a write-back to the main memory, whose latency depends on the interconnection traffic and control. As a result, the number of cycles for executing our synchronization mechanism is made up of three parts: the time required for invalidating 256 clean sets (constant and equal to 512 cycles), the time required for writing back each dirty cache line, and finally the time for the atomic access.

Several experiments were done to measure the number of cycles required to execute the *acquire* and *release* as a function of the number of dirty lines. The test was repeated ten times and the average was computed. The results are presented in tabular form in table A.1. It is important to highlight that only one core is awakened and thus the *acquire* function always reads zero, no lock contention is included in these measurements.

The numbers demonstrate a direct proportionality with the number of dirty lines, and that the average time for writing back a modified cache line (including FSM control) is about 40 cycles. The maximum latency is clearly related to the cache size. These results confirm our expectations and highlight the significant overhead of synchronization mechanisms.

Dirty lines	**Acquire cycles**	**Release Cycles**
0	573	575
1	612	614
10	982	975
100	4486	4545
1000	41537	41273
2000	82227	82592

Table A.1: Number of cycles required for executing an *acquire* and *release* function of the number of dirty lines.

A.2 Random memory accesses

The algorithm implemented by our custom benchmark is depicted in fig. A.1. Each core initially performs a random number of read and write operations on randomly selected private variables, which are stored in a static vector for each core. Then, both cores execute their critical section where they access another set of private variables and perform read and write operations on a shared vector. The critical

section is protected by the *acquire* and *release* functions previously described, which use the same lock to ensure mutual exclusion. The program uses the parameters in table A.2 to control the size of the critical section, the size of the "non-critical section", and the number of iterations of the algorithm. The number of accesses to private/shared variables, as well as their addresses within the static vectors, are randomized.

We implemented the *Random Read/Write* algorithm to estimate the performance penalties of our coherence protocol. This algorithm allows us to compute the first-order effect of self-invalidation, which is caused by the time required to execute a synchronization operation, specifically our *acquire* and *release* functions. In many programs, some sections can be executed in parallel while others must be serialized, making it a common scenario. By iterating multiple times, we obtain several interleavings between synchronizations that improve the generality of our benchmark.

Since we are primarily interested in memory accesses and traffic, we use read and write operations as the main computing element, along with random number computations. This approach enables us to estimate the second-order cost of the coherence algorithm, which is associated with the resulting miss ratio generated by conservatively invalidating the entire cache.

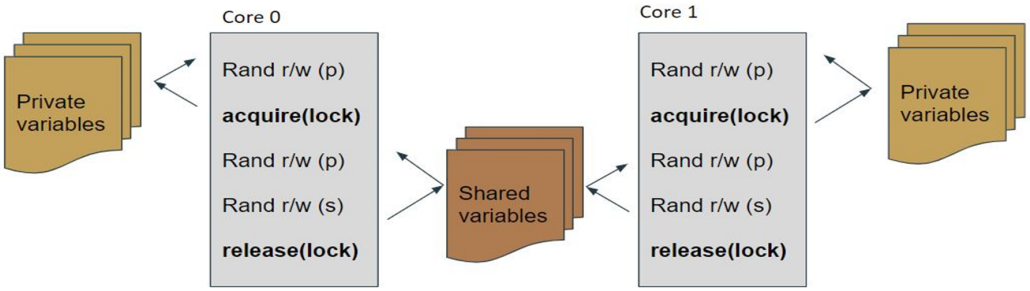

Figure A.1: Random read/write algorithm for performance estimation.

NUM_ITERATIONS	Number of iterations of the algorithm.
MAX_PRIV	Size of private vectors and also the maximum number of read/write to it.
MAX_SHARE	Size of shared vector and also the maximum number of read/write to it.

Table A.2: Parametes used in the random read/write program.

The results of the first experiment are shown in fig. A.2. The pie charts display the distribution of the algorithm's execution time divided into four segments for each core: time spent inside the *acquire* and *release* functions, time spent inside the critical section, and remaining time spent outside the critical section. In this experiment, we ran only a single iteration, and we varied the sizes of the critical and

non-critical sections from left to right, which impacted the total execution cycles and allowed us to adjust the frequency of synchronization operations. To ensure reasonable results, we maintained the critical section time to be about one-fifth of the non-critical section time.

For comparison, we compute the execution time in the ideal case where synchronizations return immediately, in zero cycles. In this case, the execution cycles depend on the interleaving of the two critical sections and oscillate between two limits. The best case occurs when the critical section of one core is executed during the non-critical section of the other core, and the worst case occurs when one core needs to wait for the entire critical section of the other core before executing its critical section, resulting in serialization.
Mathematically:

$$\begin{aligned} bestCase = max(&csTimeCore0 + nonCsTimeCore0, \\ &csTimeCore1 + nonCsTimeCore1) \end{aligned} \tag{A.1}$$

$$\begin{aligned} worstCase = max(&csTimeCore0 + csTimeCore1 + nonCsTimeCore0, \\ &csTimeCore0 + csTimeCore1 + nonCsTimeCore1) \end{aligned} \tag{A.2}$$

In table A.3 is shown the ideal execution times and the performance penalty of our implementation. It also reports the number of spins, i.e. the number of times the lock is read before being acquired. A value of two means that the lock is read only once per core, indicating that every core successfully locked its critical section at the first attempt. This number is useful because says how close the comparison is to the best or the worst-case situation.

The results highlight how the frequency of synchronizations impacts performance. When two synchronizations (an *acquire* and a *release*) are executed every approximately 2500 cycles (first column of fig. A.2), the penalty concerning the ideal case ranges from 102% to 63% (first row of table A.3). A spin number of three shows that the comparison is closer to the worst case. By reducing the frequency of synchronizations by a factor of almost 30, the penalty is reduced to a maximum of 16%, and the execution time is actually better than the ideal worst case (last column of fig. A.2 and last row of table A.3). Similar considerations can be made from fig. A.2, where it can be seen that the synchronization time slices decrease from left to right.

The same experiment was repeated by iterating the algorithm ten times to improve the generality of the test, allowing for multiple possible interleavings of critical sections. The results are summarized in fig. A.3 and compared with the ideal case in table A.4. In this case, the ideal number of spins is twenty, since there

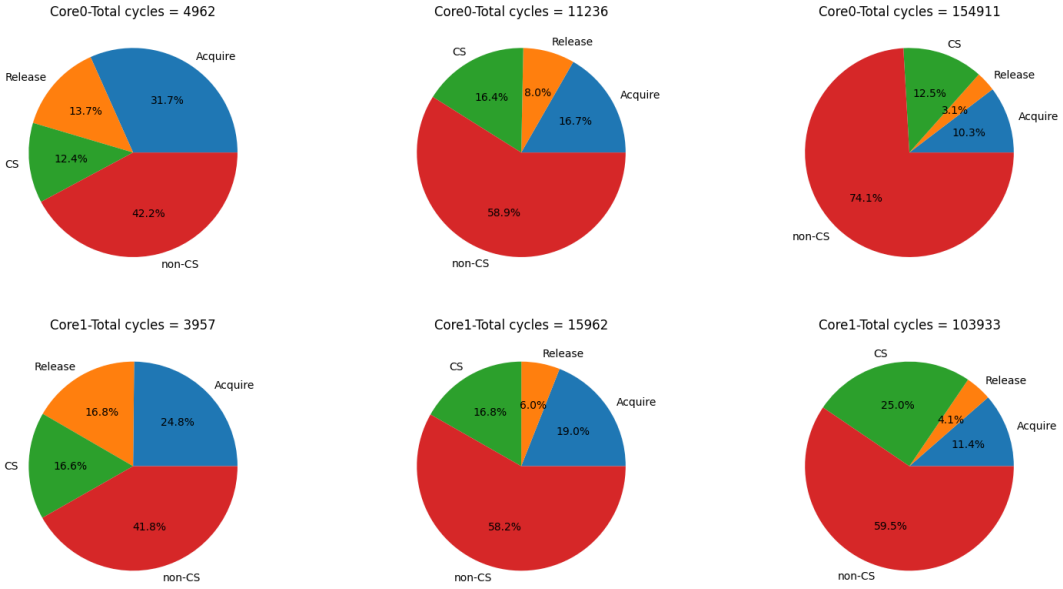

Figure A.2: Time breakdown for the random read/write program, a single iteration of the algorithm. The frequency of synchronizations decreases from left to right.

Total cycles(#spins)	**Ideal best case (penalty)**	**Ideal worst case (penalty)**
5473(3)	2708(102%)	3364(63%)
15962(2)	11974(33%)	13815(15%)
155383(2)	134119(16%)	160113(-3%)

Table A.3: Total number of cycles to execute the random read/write program against the ideal case (one iteration).

are two cores and ten iterations, corresponding to the best case where each core always finds the lock free.

The same trend observed for one iteration is confirmed for ten iterations, revealing a penalty that increases with the frequency of synchronizations.

Overall, the coherence protocol appears to degrade performance significantly compared with an ideal synchronization in zero time, but this impact depends heavily on the frequency of synchronizations and, therefore, on the algorithm itself.

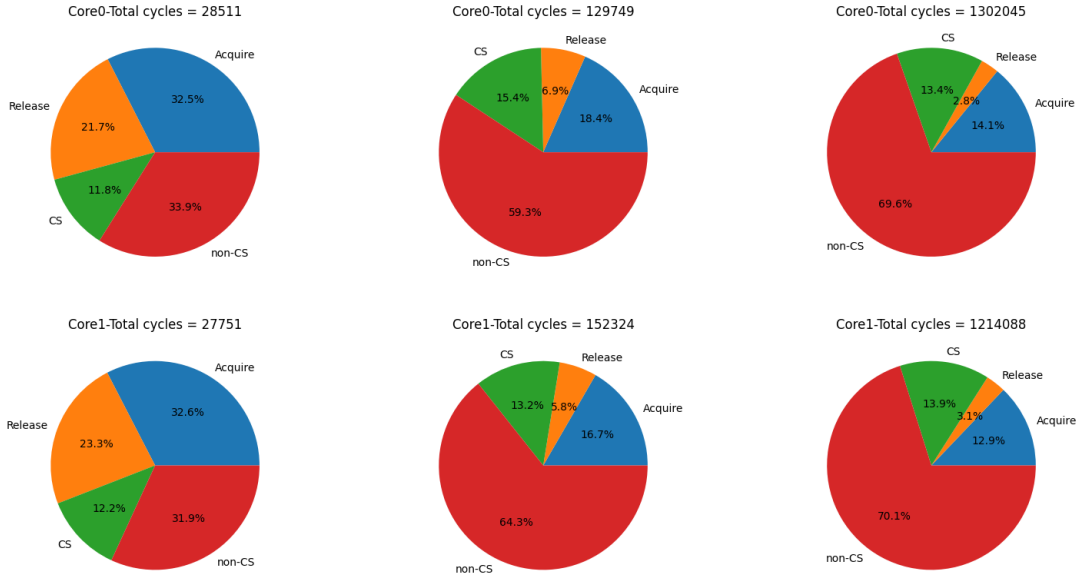

Figure A.3: Time breakdown for the random read/write program, 10 iterations of the algorithm. The frequency of synchronizations decreases from left to right.

Total cycles(#spins)	**Ideal best case (penalty)**	**Ideal worst case (penalty)**
29027(26)	13045(122%)	16431(77%)
152324(34)	118094(29%)	138070(10%)
1302561(70)	1081412(20%)	1250083(4%)

Table A.4: Total number of cycles to execute the random read/write program against the ideal case (ten iterations).

Bibliography

[1] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. «A primer on memory consistency and cache coherence». In: *Synthesis Lectures on Computer Architecture* 15.1 (2020), pp. 1–294 (cit. on pp. 1, 4, 9, 25, 33).

[2] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. «Spandex: A flexible interface for efficient heterogeneous coherence». In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 261–274 (cit. on p. 1).

[3] Jonathan Balkind et al. «OpenPiton: An open source manycore research framework». In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 217–232 (cit. on pp. 1, 13, 32, 38).

[4] Mark Wyse et al. «The BlackParrot BedRock Cache Coherence System». In: *arXiv preprint arXiv:2211.06390* (2022) (cit. on pp. 1, 13).

[5] Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. «An efficient, self-contained, on-chip directory: Dir1-sisd». In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 317–330 (cit. on pp. 1, 2, 14, 15, 46).

[6] Alberto Ros and Stefanos Kaxiras. «Complexity-effective multicore coherence». In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 2012, pp. 241–252 (cit. on pp. 1, 2, 14, 15, 46).

[7] Alberto Ros, Carl Leonardsson, Christos Sakalis, and Stefanos Kaxiras. «Poster: Efficient self-invalidation/self-downgrade for critical sections with relaxed semantics». In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 2016, pp. 433–434 (cit. on p. 1).

[8] Rui Zhang, Swarnendu Biswas, Vignesh Balaji, Michael D Bond, and Brandon Lucia. «Neat: Low-Complexity, Efficient On-Chip Cache Coherence». In: *arXiv preprint arXiv:2107.05453* (2021) (cit. on pp. 1, 14, 15, 17, 45, 46).

[9] Stefanos Kaxiras and Georgios Keramidas. «SARC coherence: Scaling directory cache coherence in performance and power». In: *IEEE micro* 30.5 (2010), pp. 54–65 (cit. on pp. 2, 14).

[10] Joseph Devietti, Benjamin P Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. «RADISH: always-on sound and complete Ra D etection in S oftware and H ardware». In: *ACM SIGARCH Computer Architecture News* 40.3 (2012), pp. 201–212 (cit. on pp. 2, 14).

[11] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. «Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset». In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99 (cit. on pp. 2, 14).

[12] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 201912132*. `https://riscv.org/specifications/`. 2019 (cit. on pp. 2, 7, 17, 18, 29, 48).

[13] F. Zaruba and L. 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 FDSOI Technology». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: `10.1109/TVLSI.2019.2926114` (cit. on p. 10).

[14] *CVA6: OpenHW Group RISC-V Advanced Development Platform*. `https://github.com/openhwgroup/cva6`. 2021 (cit. on pp. 11, 25, 31, 33).

[15] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. «Simics: A full system simulation platform». In: *Computer* 35.2 (2002), pp. 50–58 (cit. on p. 14).

[16] Alvin R Lebeck and David A Wood. «Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors». In: *ACM SIGARCH Computer Architecture News* 23.2 (1995), pp. 48–59 (cit. on p. 14).

[17] Litmus Tests. *litmus-tests-riscv*. `https://github.com/litmus-tests/litmus-tests-riscv` (cit. on pp. 28, 30).

[18] inria. *DIY Toolkit*. `https://diy.inria.fr/` (cit. on pp. 28, 30).

[19] CTSRD-CHERI. *CHERI-Litmus: A Collection of Litmus Tests for CHERI*. `https://github.com/CTSRD-CHERI/CHERI-Litmus`. 2021 (cit. on p. 30).

[20] OpenHW Group. *CVA6-SDK release*. `https://github.com/openhwgroup/cva6-sdk/releases`. 2019 (cit. on p. 32).