Informatics

Atomic Instruction und Cache-Unterstützung in VADL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Simon Himmelbauer
Matrikelnummer 12044925

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 29. Oktober 2024

Simon Himmelbauer

Andreas Krall

Atomic Instruction and Cache-Support for VADL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Simon Himmelbauer
Registration Number 12044925

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, October 29, 2024

Simon Himmelbauer

Andreas Krall

Erklärung zur Verfassung der Arbeit

Simon Himmelbauer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Oktober 2024

Simon Himmelbauer

Acknowledgements

First and foremost, I would like to thank my parents and my brother for their unconditional mental and financial support as well as always being available when I needed them.

Furthermore, thank you to all my friends and library companions who cheered me up when work or other things in life were simply too much. I want to specifically thank my friends Johanna and Jeff as well as especially my girlfriend Emma for holding through with me during these turbulent times.

Special thanks go to my supervisor Prof. Andreas Krall for providing me the opportunity to work on this project and always being available for feedback and further advice. Last but not least, I want to thank the entire VADL team for providing assistance, particularly in the beginning of the project.

Kurzfassung

Die Entwicklung von neuen Prozessoren für domän-spezifische Anwendungen stellen eine herausfordernde Aufgabe dar, die mit vielen Komplexitäten und Kosten verbunden ist. Prozessorbeschreibungssprachen wie die Vienna Architecture Description Language (VADL) ermöglichen schnelle "design space exploration", da AnwenderInnen die Struktur und das Verhalten ihrer Architektur beschreiben und die VADL Generatoren automatisch Compiler, Linker, Assembler, Simulatoren und sogar synthetisierbare Hardware in einer Hardwarebeschreibungssprache erzeugen können.

Allerdings ist VADL noch weit entfernt, um alle Funktionen moderner Prozessoren ab zu bilden. Erstens unterstützt es keine Übersetzung von virtuellen zu physischen Adressen, was zum Starten aktueller Betriebssysteme benötigt wird. Zweitens kann VADL atomare Instruktionen nicht modellieren. Allerdings sind diese fundamental für Mehrkernprozessoren, insbesondere für die Implementierung von parallelen Algorithmen für den Befehlssatz einer bestimmten Architektur. Letztlich besitzen fast alle gängigen CPUs in PCs, Handys und Einplatinencomputer mehrere Cacheebenen, um Speicher-Latenzzeiten zu verkürzen und dadurch die Ausführung zu beschleunigen. VADL bietet jedoch keine Möglichkeit zur Modellierung von Caches und Cache-Hierarchien. In dieser Arbeit erweitern wir VADL und die generierten Simulatoren, um all die vorhin genannten Aspekte zu integrieren.

Um die genannten Funktionalitäten zu demonstrieren, implementierten wir atomare Instruktionen für unsere RISC-V RV32 und AArch64 VADL Spezifikationen. Außerdem erweiterten wir VADL, sodass Caches und beliebige Cache-Hierarchien definiert werden können. Für diesen Zweck präsentieren wir einen neuen Cachesimulator, welchen wir von Grund auf an die Anwendungsfälle von VADL angepasst haben. Um den Einfluss von Caches besser verstehen zu können, kann der Simulator sowohl funktionale als auch hardwarespezifische Aspekte, wie zum Beispiel Kohärenzzustände und Nachrichten, welche über den Interconnect gesendet werden, simulieren. Wir validierten die Genauigkeit des Cachesimulators durch einen Vergleich mit gem5, ein im Forschungsumfeld etablierter Allzwecksimulator. Zu guter Letzt implementierten wir den Sv32 Adressübersetzungsalgorithmus der RISC-V Architektur in VADL.

Abstract

Designing new processors for domain-specific applications can be a cumbersome task due to the high complexity and costs involved. Hence, processor description languages, such as the Vienna Architecture Description Language (VADL), enable rapid design space exploration because users may describe the structure and behavior of their architecture and the VADL generators automatically produce compilers, linkers, assemblers, simulators and even synthesizable hardware in a hardware description language.

However, VADL is still away from allowing to design all features modern processors provide. Firstly, it does not support translation from virtual to physical addresses necessary for properly booting modern operating systems. Secondly, VADL cannot model atomic instructions. However, they are fundamental for multi-core processors, particularly when implementing parallel algorithms on the given instruction set architecture. Finally, almost all CPUs, found in PCs, phones and even single-board computers, employ multiple levels of caches in order to reduce memory latencies and hence, speed up execution. However, VADL does not offer a way to model caches and cache hierarchies. In this work, we extend VADL and its generated simulators to integrate all these aforementioned aspects.

To showcase our proposed features, we implemented atomic instructions for our RISC-V RV32 and AArch64 VADL specifications. Furthermore, we extended VADL to allow defining caches and arbitrary cache hierarchies. For this purpose, we present a new cache simulator designed from scratch tailored towards the use cases of VADL. In order to understand the impact of caching, the simulator is capable of simulating high-level functionality and low-level aspects such as coherence states and messages sent via the interconnect. We validated parts of its accuracy by comparing it to gem5, which is a general purpose simulator well-established in academia. Finally, we implemented the Sv32 address translation scheme of the RISC-V ISA in VADL.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

In a world where Moore's Law slowly converges toward its limit, manufactures follow the trend of packing more and more cores onto the same die. When designing an out-of-order multi-scalar and/or multicore architecture, additional aspects become relevant:

1. What memory model do architects want to implement?
2. What instructions do they require in order to provide stronger guarantees than given by the memory model?
3. How does all of this affect the cache coherency between cores?

We explain the relevance of these questions in the following paragraphs. We begin with why a CPU even needs a memory model? Modern processors implement a variety of techniques to keep their pipeline busy and therefore, increase its performance. Examples are multiple layers of caches as well as out-of-order and speculative execution. For instance, a CPU might prefetch certain memory loads in order to hide their latency as much as possible. In addition, CPUs commonly have multiple layers of caches in order to avoid costly accesses to main memory. These caches are often tied to a specific set of cores (e.g. Level 1 caches are usually tied to a single core, while the level 2 cache is shared). This means that a store to a certain memory location is not necessarily visible to other threads because they might access an older value in their local cache. These tricks work fine in single-threaded use cases because they are designed to be transparent to the underlying process. However, when working with parallel algorithms, the effects of these techniques become apparent. So how can a software engineer implement a parallel algorithm correctly? This is where the memory/concurrency model comes into action: The memory model of an architecture describes how memory accesses can be reordered (e.g. the memory model might guarantee that the CPU keeps all stores in the same order). The concurrency model often relates to the underlying microarchitecture. For instance, x86 implements the total store order (TSO) model which originates from the fact that each core has a FIFO write-buffer to hide the latency of commencing a write directly to memory [NSH$^+$20, p. 3]. This is why the memory model and the underlying caching architecture are closely related which answers why question one and three matter to a CPU architecture designer.

To answer the second question, consider our software engineer who now has a memory model explaining how the underlying microarchitecture affects the correctness of a parallel algorithm. What if it turns out that the provided guarantees are too weak? Modern architectures provide so-called memory fences or barriers to handle this problem. A fence instruction prevents the CPU from ordering a memory access across its location, just like an actual fence. Weaker architectures (weak refers to the memory model) often provide different types of fences, varying in strength but also in performance costs. Thus, it is again up to the architecture designer to define what fences are necessary which again largely depends on the memory model as well as underlying caching mechanism. We observe that these three concepts closely interact with each other.

Last but not least, concurrent languages and multicore machines additionally require support for atomic operations. For instance, take two threads A and B which each increment a value stored at some memory location n number of times. The expected result is $2n$. Each increment operation consists of three steps: Loading the value (read), incrementing it (modify) and storing the result back (write). Now take thread A that loads the initial value 0 and then stops. Next, thread B executes the increment operation n times. Now A continues, increments 0 and stores the result (one) back to memory. Hence, the final value (n) of thread B is not considered and simply overwritten. We observe that the final result does not equal to $2n$. Note that the strength of the memory model does not impact the correctness here because even in case our CPU executes the program in program order and has no caches, which in consequences means every store is visible in a total order, the execution previously described may still occur. Instead, engineers require a certain class of atomic operations that combine all three steps (read, modify and write) into a single observable step. The most common classes are fetch-and-modify (-Add, -Sub, etc.) and compare-and-swap (CAS).

Following these design considerations, we focus on the job of a CPU architect. The Vienna Architecture Description Language (VADL) is a processor description language (PDL) that allows developers to design a CPU on all levels, from the instruction set and ABI down to the underlying microarchitecture. Currently, VADL misses support for the following aspects:

- **Atomic instructions**: As mentioned above, some concurrent and parallel algorithms require certain synchronization primitives to function correctly. This requires the architecture to provide certain instructions such as memory fences/barriers or read-modify-write (RMW) instructions. Thus, VADL must support a way to prevent reorderings in case of memory barriers as well as a way to define atomic operations.

- **Caches and Memory Hierarchy:** VADL currently does not support any type of CPU cache. However, they are imperative for fast CPUs, even in the case of single-threaded processors. Furthermore, when designing atomic instructions on a microarchitectural level, their effect becomes more interesting in combination with caches. A flexible mechanism to define caches could also allow VADL developers to more easily experiment with different hierarchies, sizes and coherence protocols. Furthermore, simulation facilitates design space exploration (DSE) [BKP20].

- **Memory Model** A memory model is required in order to properly define what kind of synchronization instructions are necessary. For instance, a sequentially consistent architecture does not require any form of fence instructions, while a more relaxed model might want to provide different types of barriers, each with different strengths and guarantees.

1.2 Aim of the Thesis and Expected Results

The goal is to lay a foundation for defining out-of-order multi-scalar and multicore processors in VADL. This means VADL must support the following aspects:

- **Definition of caches and declaration of memory hierarchy**: The goal is to declare the total and line size, associativity, replacement policy and coherence protocol of a cache. As part of this thesis, one idea is to configure caches via annotations, a syntactical mechanism provided by VADL.

- **Definition of address translation**: While not strictly necessary for multicore processors, we want to add support for defining address translation from virtual to physical addresses. Since memory management units (MMUs) themselves utilize caches such as the translation lookaside buffer (TLB), this topic conveniently matches the task of implementing a caching subsystem.

- **Definition of atomic instructions**: This specifically includes fetch-and-modify, compare-and-swap and load-reserved/store-conditional. The goal is to support all atomic operations provided by ARMv8 AArch64 [Arm21, C3.2.6, C3.2.12.1 - C3.2.12.4] and RISC-V RV32 (Standard Extension for Atomic Instructions, Version 2.1) [RIS19].

- **Declaration of memory model**: The architect must specify what memory model is desired for the instruction set architecture. It is important that this model is then fully respected by both the instruction set and underlying microarchitecture. Ultimately, a developer should be able to define an arbitrary axiomatic or operational model. However, this thesis focuses on a set of specific models.

In order to meaningfully test these features, the instruction set simulator (ISS) and cycle accurate simulator (CAS) will be extended to support these language features. For the ISS, it is sufficient to handle atomic instructions and address translation because microarchitectural details such as caching are not relevant here. On the other side, the CAS must support all aspects of the hardware.

Needless to say, the effects of caching behavior and atomic instructions become more apparent in multicore scenarios. However, both simulators currently do not support multithreading and adding multithreading support is specifically not in the scope of this thesis. However, we can measure how emulating the cache hierarchy affects the throughput of the CAS. In addition, the simulator should be capable of tracking cache-specific information such as miss rate.

1.3 Research Questions

RQ1: *How can we extend VADL to enable developers to fully model and implement atomic instructions such as Fetch-And-Modify, CAS and memory fences (as provided by ARMv8 and RISC-V)?*

As mentioned above, we want to offer VADL developers the possibility to specify arbitrary atomic instructions. Our idea is to study the set of operations provided by common ISAs, such as x86, ARM, RISC-V and Power, to help us derive what features are necessary to cover their use cases.

RQ2: *How can memory models and cache coherence be efficiently and correctly implemented in a simulator?*

This question primarily concerns the cache and write buffer simulation. Correctness is particularly important for design space exploration because developing hardware prototypes for a new chip design is costly. Needless to say, accuracy is a broad term. In the context of this thesis, we decided to implement a functional cache simulator, based on the definition from Brais et al. [BKP20]. Hence, we want to simulate behavioral events such as cache misses, coherence state changes and interconnect topographies. Some simulators go into more detail and consider detailed timings such as how long a message needs to traverse the interconnect. However, we think this has a significant impact on the performance of the simulation and hence, believe that this additional overhead is not worth for the purposes of VADL.

RQ3: *What performance impact does cache emulation have on the cycle accurate simulation?*

A cache simulator might have a considerable influence on the overall performance of the simulator. Consider a simple system with just an instruction cache: The CAS requires constant interaction with the cache simulator because the cache must be consulted for every single instruction fetch. Hence, we want to evaluate how the cache simulation impacts the overall runtime of our CAS.

1.4 Methodology

We begin with an onboarding phase in order to become familiar with the current state of the project which includes VADL itself, the VADL frontend, VADL intermediate representation (VIR) and the instruction set simulator (ISS) as well as cycle accurate simulator (CAS). Next, we design two to three proof-of-concept specifications in VADL to support address translation, caches and atomic instructions so that they comply with the goals described in section 1.2. Regarding RQ1, deciding on the most concise and intuitive version should be based on an empirical study, as conducted by [PSZ21]. However, this is not in the scope of this thesis. Instead we compare our design to existing solutions and base our preference on personal opinions. This version will be integrated into VADL in addition to both the ISS and CAS receiving the necessary support to simulate caching, address translation and atomic instructions (RQ2).

As mentioned in section 1.2, one of the stated goals is the ability to implement all atomic instructions of ARMv8 and RISC-V in VADL. The latter can be validated using the official RISC-V test suite[1]. While ARM provides a validation framework as well, it is unfortunately not publicly available [MRSM16]. Hence, we verify their correctness by hand. Next, we evaluate the impact of simulating the caching subsystem by comparing its throughput to a version without caching (RQ3). Furthermore, we verify the functionality of the cache hierarchy by measuring cache-specific data such as hit and miss rate and comparing them to a state-of-the-art simulator such as gem5 [BBB+11, LAA+20] (RQ2).

1.5 Structure of this Thesis

Chapter 2 introduces the reader to theoretical concepts of caches, memory translation and atomic instructions. In addition, we present an overview of VADL. Chapter 3 summarizes current research on the topics of PDLs and cache simulation. In Chapter 4, we state our language design choices for defining caches, write buffers, memory hierarchies, atomic instructions as well as address translation in VADL. We also present our cache and write buffer simulator. Next, we discuss our VADL specifications of atomic instructions and performance findings in terms of cache simulation and address translation in Chapter 5. Finally, we propose further improvements and extension to VADL in Chapter 6 and conclude our thesis in Chapter 7.

[1] `https://github.com/riscv-software-src/riscv-tests`

CHAPTER 2

Background

This section introduces the reader to several concepts fundamental to understanding the goal of this thesis. Note that many sections talk about memory accesses. We use the terms 'reads' and 'loads' as well as 'writes' and 'stores' interchangeably.

2.1 Vienna Architecture Description Language (VADL)

The Vienna Architecture Description Language (VADL) is a PDL for designing all aspects of a processor, ranging from the instruction set architecture (ISA), application binary interface (ABI) to the microarchitecture (MiA). It was developed at the Technical University of Vienna as part of a research project [Sch20, Mih23, Sch22, Gra21, HHH^{+}24, HK23]. The following Listing provides an overview of some of the features part of VADL:

```
constant MLen = 32

using BitsM = Bits<MLen>
using SIntM = SInt<MLen>
using UIntM = UInt<MLen>

function lessthan (a: SIntM, b: SIntM) -> Bool = a < b

import rv32i::RV32I

instruction set architecture RV32IM extending RV32I = {}
application binary interface ABI for RV32IM = {}
user mode emulation UME for CPU = {}
assembly description Assemble for RV32IM = {}
micro processor CPU implements RV32IM with ABI = {}
micro architecture FiveStage implements CPU = {}
```

Listing 2.1: Overview of the Vienna Architecture Description Language.

Lines 11-16 define the different aspects of a CPU architecture. Note that depending on the use case, not all definitions are required. For instance, an ISS does not require a MiA specification, while the compiler generator may optionally use the additional information to improve instruction scheduling for instance. Needless to say, meaningful cycle-accurate simulation relies on a MiA. Lines 1-9 in Listing 2.1 present some structural features of VADL. For instance, constant definitions can be used to easily switch between 32- and 64-bit address width while keeping the same ISA definition. The **using** keyword can be compared to C's **typedef**. VADL also provides functions as seen on Line 7. Here, the function `lessthan` takes two arguments a and b both of type `SIntM` and returns a value of type `bool`. Note that functions must be pure in VADL. Hence, any side effects are forbidden, such as reading or writing from/to memory. We additionally want to highlight the strict separation of ISA and MiA. All instructions, registers and memory are defined generically in an ISA definition, while the MiA is only concerned with execution of said instruction set. This enables CPU engineers to write multiple MiAs for the same ISA.

The VADL frontend then takes a VADL specification and allows to generate a compiler backend, simulators as well as a hardware description language, which can be synthesized on an FPGA. To be more precise, the VADL frontend currently supports generating an LLVM compiler backend, instruction set simulator (ISS), a cycle accurate simulator (CAS) as well as Chisel for hardware description. However, the general design goal of VADL and its frontend is to support arbitrary backends. Hence, the VADL generators accommodate a common intermediate language called the VADL intermediate representation (VIR) which conceptually follows static single assignment (SSA) form. This section introduces some of the core features of VADL.

2.1.1 Instruction Set Architecture

An ISA definition fundamentally consists of the following construction:

```
instruction set architecture SYS = {
    using Byte = Bits<8>
    using Index = Bits<5>
    using Address = Bits<32>

    program counter PC : Address
    memory MEM : Address -> Byte
    register file X : Index -> Byte
    // ...
}
```

Listing 2.2: Basic instruction set architecture (ISA).

The example above defines an ISA with the name `SYS` and showcases some general aspects of VADL. `Bits<N>` is a primitive type representing a bit vector of size N. VADL also provides specialized integer types `UInt<N>` and `SInt<N>` to represent unsigned integer (with modulo arithmetic) as well as signed integers using the two's complement. The **using** directive allows to alias a type with a more relevant identifier in the given context. For example, in Listing 2.2, Line 4 defines a specific `Address` type. Thus, using this new type provides more semantical information than using the primitive `Bits<32>` directly. Furthermore, this allows developers to more easily reuse the ISA definition. For instance, if users want to support an address width of 64, they will need to change only one line of code. Having to alter the size only at one location additionally tends to be less error-prone. Line 6 defines a program counter named `PC` having the type `Address` (`Bits<32>`). Next, the architecture needs some memory, which is called `MEM` and defined on Line 7. Observe that `MEM` is actually a function mapping an address to a single `Byte` (defined as `Bits<8>` on Line 2). Register definitions are similar, however, the argument describes an index i to access the ith register. Thus, the size of the argument also implies the amount of registers of the given register file. In our case, index is a bit vector of size five and thus, the architecture has $2^5 = 32$ registers.

Next, in order to complement the ISA with an instruction, the specification first requires information about the structure of an instruction. Consider the following example of a format definition:

```
instruction set architecture RV32I extending SYS = {
    format FormatB : Byte = {
        arg : Index,
        res : Index,
        opc : Bits<6>
    }
    // ...
}
```

Listing 2.3: Format definition in VADL.

The **format** keyword defines a format with the name `FormatB` and the type `Byte`. Hence, this type of instruction consists of 16 bits. These bits are separated into three fields: `arg`, `res` and `opc`. Recall that `Index` is defined as a bit vector of size five. Thus, all fields summed up result in 16 bits. Note that the order of the fields matters in how they are layed out in memory, as seen in Figure 2.1. We also want to highlight another feature of VADL: An ISA can be based on another ISA specification using the **extending** keyword. Compare this to the approach of RISC-V which is separated into several extensions [RIS19]. VADL allows to define the base architecture as `RV32I` and an implementation including integer multiplication and division (M-extension) can be provided separately by defining a new ISA via **instruction set architecture** `RV32IM` **extending** `RV32I`. Next follows a definition of our first instruction:

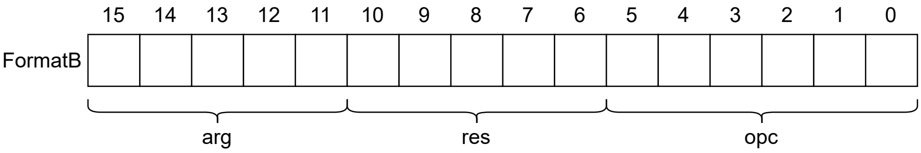

Figure 2.1: Memory layout of our example format definition `FormatB`.

```
1 instruction LOAD : FormatB = {
2     let addr = (0x1000'0000 as UInt<32>) + (X(arg) as UInt<32>) in {
3         X( res ) := MEM( addr )
4     }
5 }
6 encoding LOAD = {
7     opc = 0b0010
8 }
9 assembly LOAD = (mnemonic, " ", register( res ), ",", register( arg ))
```

Listing 2.4: Instruction definition in VADL.

Lines 1-5 define the semantics of an instruction called `LOAD` using the previously defined `FormatB` as a structure. All fields of the format are directly accessible within the definition. The Listing 2.4 shows how the address is calculated from a fixed base address and an offset stored in register `X(arg)`. On line 3, the value stored at the address is loaded and saved in register `X(res)`. Observe how the `arg` and `res` fields are used as parameters within the definition. Recall that our format definition also contains the field `opc`. Needless to say, the CPU instruction decoder must be able to uniquely identify the appropriate instruction. Generally, instructions have a fixed structure (defined via a format in VADL) and a part of it, called the *opcode*, identifies what instruction to execute [PH17, p. 198]. Hence, we added an `opc` field in Listing 2.3. Line 7 establishes the static structure of an instruction in the **`encoding`**. In theory, these steps are sufficient for a simulator or hardware generator. However, the compiler generator also requires information about the structure of the assembly language, defined on Line 9. A VADL developer simply specifies the syntax and the VADL generators automatically produce the appropriate grammar. However, many instructions share a similar structure in practice and hence, manually defining the assembly string for each instruction leads to a lot of boilerplate code. VADL provides additional helper functions to avoid this situation: The **`mnemonic`** builtin uses the identifier of the instruction for matching. For instance, assume our ISA also had two instructions named `ADD` and `STORE`. We could write the following code:

```
assembly ADD, LOAD, STORE =
    (mnemonic, " ", register(res), ",", register(arg))
```

Listing 2.5: Handling multiple instructions with a single **`assembly`** definition.

The **register** builtin matches the field of a format to a register. The VADL frontend includes a semantical analysis to check on what register file the field is used [Sch22]. For instance, consider an architecture with a separate set of address and data registers, called `A` and `D` respectively. Next, assume an instruction `LOAD D1, A2` that loads the value stored at address `A2` in memory and saves the data to register `D1`. Without the aforementioned analysis, the assembler would need to assume that any register file is valid. Hence, it would accept `LOAD D1, D2` for example. The analysis ensures that the VADL frontend only generates rules for registers that are actually used by the specific instruction.

2.1.2 Application Binary Interface

This section presents how CPU engineers may specify an application binary interface (ABI) in VADL. We continue with the example introduced in Section 2.1.1.

```
 1 application binary interface ABI for RV32I = {
 2     alias register a0 = X(0)
 3     alias register a1 = X(1)
 4     alias register a2 = X(2)
 5     alias register a3 = X(3)
 6     alias register sp = X(31)
 7
 8     [ alignment : Bits<8> ]
 9     stack pointer = sp
10     return address = X(29)
11     frame pointer = X(30)
12     return value = [ a0 ]
13     function argument = [ a{0..3} ]
14     caller saved = [ a{0..3}, X(4), X(5), X(6), X(7) ]
15     callee saved = [ X(8), X(9), X(10), sp ]
16 }
```

Listing 2.6: ABI definition in VADL.

First, we introduce another VADL concept on Line 7 called *annotations*. They equip the subsequent element with additional attributes. What type of annotations are supported depends on the specific object. In our example, the stack pointer definition has the annotation `alignment`, specifying that the stack pointer must always be aligned to eight bits. Listing 2.6 also shows definitions of several typical elements of an ABI which should mostly be self-explanatory. Observe that VADL also supports aliasing registers commonly used by architectures [RIS19, p. 137]. For instance, `a0` and `sp` map to the registers `X(0)` and `X(31)` respectively. Aliases additionally help save some boilerplate code, for instance, when defining function arguments on Line 13. Here, `a{0..3}` expands into `a0, a1, a2, a3`.

```
    micro processor CPU implements RV32I with ABI = {
        start = 0x8000'0000
        stop = PC = 0xeeee'eeee

        exception invalid = {
        }

        startup = {
        }
    }
```

Listing 2.7: Microprocessor definition in VADL.

2.1.3 Microprocessor and Microarchitecture

The idea of the micro processor section is to describe general behavior that are independent from a MiA. As seen in Listing 2.7, this includes defining the start address of execution, a stop condition (mostly relevant for simulation), exception handling as well as including startup code to execute before the user-supplied binary. Last but not least, we can specify a MiA for our micro processor `CPU`, as seen in Listing 2.8.

```
micro architecture P3 implements CPU = {
    stage FETCH -> ( fr : FetchResult ) = {
        // ...
    }

    stage DECODE -> ( ir : Instruction ) = {
        // Previous stage result accessible via FETCH.fr
    }

    stage EXECUTE = {
        // Previous stage result accessible via DECODE.ir
    }
}
```

Listing 2.8: Microarchitecture definition in VADL.

This definition only presents a small subset of the MiA features. Our example implements the `CPU` micro processor in a three-stage pipeline MiA called `P3`. The pipeline is separated into three stages named `FETCH`, `DECODE` and `EXECUTE` where each stage can access the output of the previous one.

2.2 Caches and Cache Coherence

In a modern processor, reading or writing to main memory usually belongs to the slowest aspects in the pipeline of a CPU. Thus, modern compilers try to generate code which minimizes the amount of accesses to main memory by keeping values in registers. However, accessing main memory becomes unavoidable at some point and in order to bridge the gap, CPUs provide multiple levels of caches that contain a subset of main memory. The size of a cache is considerably smaller than main memory (e.g. the Cortex-X1 has an L1 data cache of size 64 kB [Arm20, p. A6-74]). How does a system determine what data to store in the cache? The concept of caching is based on **temporal** and **spatial locality** [PH17, p. 738]. The former assumes that a value that has recently been accessed is likely to be accessed again, while the latter assumes that data closely located has a high probability to be accessed as well. For example, consider an array A in the programming language C and the common case of iterating through all elements in A. Recall that an array always represents a continuous part in memory. Instead of loading each element individually from main memory, a cache may fetch a larger region surrounding the first element and thus, accessing subsequent elements in A will be much faster because they are already contained in the cache. This block of data, fetched from memory at once, is called the **cache line**. Its size can vary among microarchitectures, but many, such as the Cortex X1, have a cache line size of 64 bytes [Arm20].

There exist several approaches for managing cache lines. One is to store them in an array-like manner as seen in 2.2 which is called a **direct-mapped** cache. To determine the location of a cache line, the controller considers the index of an address which directly maps to an entry in the cache. The size of the index depends on the amount of cache lines. For instance, a cache with 1024 entries requires $\log_2(1024) = 10$ bits. However, the mapping from address to index is surjective. In other words, multiple addresses map to the same index. The tag in combination with the index uniquely identifies a memory region and thus, the controller must store the tag in addition to the data in order to check whether the cache line corresponds to the requested memory region. If the tag does not match, the cache line needs to be evicted and the requested data must be fetched from main memory. Next, as mentioned above, a cache line contains several bytes. The offset precisely determines a specific element within the cache line. The size of the offset directly correlates with the size of the cache line. For instance, a cache line of size 64 bytes requires $\log_2(64) = 6$ bits. Finally, the size of the tag consists of the remaining parts of the address: Size of address $-\log_2$(Number of cache entries) $-\log_2$(Size of cache line). To continue with our example, this corresponds to $32 - 10 - 6 = 16$ bits. [PH17, p. 754-764]

The advantage of a direct-mapped cache is its constant lookup time and thus, it scales well with the size of the cache. However, consider a program which accesses different memory regions but all addresses map to the same index. Then the single cache line in the cache needs to be constantly evicted, while other indices of the cache remain completely unused. An alternative approach is to manage its cache lines in a set instead of an array. Thus, the index becomes irrelevant and only the tags are considered. This

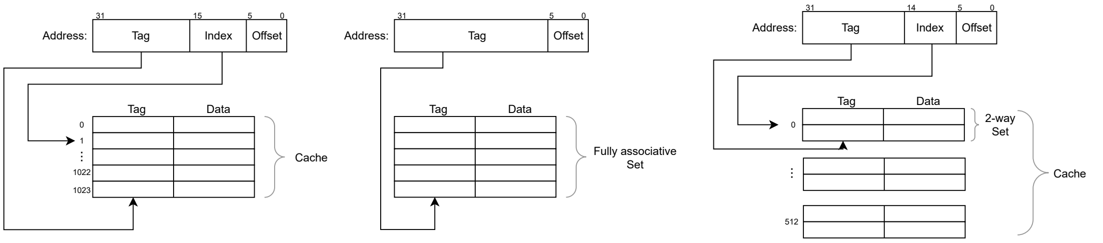

Figure 2.2: Basic examples for a cache of size 64 KiB in three different variations from left to right: Direct-mapped, fully associative and 2-way set associative. An address is separated into a tag, index and offset. Sizes of each parameter depend on the configuration of the cache. Index maps to the entry in the cache, tag determines whether cache line corresponds to the same memory region. Offset corresponds to offset within cache line. Figure inspired by [PH17, p. 762]

.

type of cache is called **fully associative** and allows full utilization of the entire cache because the controller only needs to evict a cache line if the set is full. This adds more complexity though: The decision on what cache line the cache controller decides to evict (called the **replacement policy**) might impact the performance of the system. For example, consider a worst-case situation where the CPU alternatingly accesses two memory regions (mapping to two different cache lines) and the controller always replaces one with the other. Here, this scenario completely eliminates the benefits of a cache. CPU vendors have come up with several schemes. Basic ones include round-robin, random, least-frequently used (LFU) or least-recently used (LRU). For instance, the Cortex X1 uses pseudo-LRU for its L1 data and instruction caches [Arm20, p. A6-74]. Since LRU is difficult to implement in hardware and particularly does not scale with the amount of cache lines, this replacement policy is often approximated, hence the name pseudo-LRU [PH17, p. 861]. There exist also more thorough and complex schemes, such as Static and Dynamic Re-reference Interval Prediction (SRRIP/DRRIP) [JTSE10], Sampling Dead Block Prediction (SDBP) [KTJ10] and Signature-based Hit Predictor (SHiP) [WJH⁺11].

A fully-associative cache might seem clearly superior to its direct-mapped counterpart because it is able to fully utilize the entire cache. However, implementing a fully associative cache requires comparing the tags of all cache lines which incurs additional overhead and might be infeasible to achieve in hardware. However, there exists a way to combine both ideas: Instead of having one large set, the cache can be divided into several sets and each set corresponds to a certain index. Each set can then hold multiple cache lines with the same index. This is called a **set associative** cache. For instance, a cache with sets of size 4 is called a 4-way set associative cache. Which configuration is most desirable highly depends on several factors. For instance, a fully associative cache might be feasible if the cache is not very big and/or a cache miss induces a high penalty [PH17, S. 860].

Last but not least, caches can be distinguished in terms of how they handle writes.

Thread 1	Thread 2
`A = 1`	`while(A == 0)`

Figure 2.3: Example execution for cache coherency. `A` is initialized with 0. Inspired by [NSH+20].

A **write through** cache forwards all writes directly to memory, while a **write back** configuration writes a value to the cache and thus, the modified data will be written back to memory once the cache line is evicted. [PH17, S. 862]

2.2.1 Cache Coherence

Modern CPUs usually have several layers of caches where each layer tends to be bigger but also slower to access. For instance, the Intel Ice Lake Client Microarchitecture has a L1 access latency of five cycles, while L2 already needs 13 cycles [Int24, p. 2-21]. This is called the cache or memory hierarchy because if a requested memory region does not exist in the level 1 cache, then it has to be fetched from the second level. If it also does not exist there, it needs to be loaded from the third level and so forth. Next, consider multi-core CPUs where another property of a cache becomes relevant: A cache may be shared by multiple cores or is local to a core. For instance, modern CPU cores usually have their own L1 data and instruction caches but the L2 cache could be shared by all cores. Observe that this introduces another problem: Consider the example program in Figure 2.3. Thread 2 begins execution and since A is initialized to 0, the loop will be continuously executed. Thread 1 then stores $A = 1$. In a system that only consists of main memory, thread 2 will see the value written by 1 and exit the loop. Next, consider a system where each core has a local cache. Thread 2 remains stuck in the loop. Why? Both threads have a copy of A in their local caches. Since 2 executes first, $A == 0$. Once thread 1 stores $A = 1$, this value is only updated in its local cache and thus, not visible by 2. Even if thread 1 decides to evict the cache line at some point and writes $A = 1$ back to memory, thread 2 will not realize that the value has been modified. This example shows the necessity for the system to provide additional guarantees, which are called **coherence invariants**. There exist different variations of these properties, such as token coherence [MHW03]. However, they are conceptually similar and we stick to the definition from [NSH+20, p. 13]: The authors describe two safety invariants called Single-Writer, Multiple-Read (SWMR) and Data-Value invariant. The former describes that at any point of time either one core may exclusively read and write to a location or multiple entities may only read from the same location. This concept allows us to separate memory accesses into single-writer and multiple-reader epochs. The Data-Value invariant states that the value of a memory location at the end of an epoch is equal to the value at the beginning of the succeeding epoch. Both invariants solve our previous example: The first epoch starts when thread 2 acquires read-only access to A where $A == 0$. Once thread 1 attempts to write $A = 1$, thread 2 loses access to A and thus, a new single-writer epoch begins. Finally, thread 2 tries to read from A again, leading to a new read-only epoch. Due to our Data-Value invariant, the value of A must be 1 because the write epoch ended with $A == 1$. Thus, thread 2 exits the loop. This results in so-called **cache coherence** and both the SWMR and Data-Value property comprise the safety invariants of **cache coherence**.

In order to provide cache coherence, cache and memory controllers implement a finite state-machine (FSM) associated with each cache line. One of the simplest protocols is the **MSI** protocol. The name corresponds to the three main states that a cache line can be associated with. This protocol closely matches the SWMR invariant where the *modified* state describes that the cache controller is allowed to read and write, while the *shared* state denotes read-only access. All caches are connected via a bus to the memory. If a cache requires access to a memory region, they send a request on the bus, including the access type (read-only or read-write) as well as the address. The other cache entities monitor the bus to react accordingly. For instance, if some caches have a certain memory location in the *shared* state and some cache then requests read-write access, the other caches will see those messages and evict their copies of the same location. Thus, these type of protocols are called snooping cache coherence protocols. Another common protocol family are directory coherence protocols [LLG$^+$90]. The main problem with a snooping-based system is that all messages are broadcasted to all entities and every cache controller has to process these messages, limiting scaling. Directory coherence protocols follow the idea that there exists a global view on the state of a memory location (the directory) and thus, requests are sent directly to the directory. The directory then forwards the message accordingly. For instance, if a cache A requests read-write access, while another cache B currently owns the location, the directory will forward the message from A to B, which then evicts the cache line and sends it to A directly.

2.2.2 Common Cache Coherence protocols

The design of the cache coherence protocol heavily depends on the structure of the cache hierarchy. Relevant aspects include the amount of levels, whether the cache should be write through or write back, inclusive-or-exclusive and whether the protocol must support non-uniform memory access (NUMA) configurations. All these properties particularly determine what transitional states are required. Note that there are usually two sets of states: One for the cache and another for the memory controller. To keep this section easier to understand, we want to stick to the states of the former. We begin by introducing some of the more common protocols in their base configuration:

MSI: This protocol consists of three states *Modified*, *Shared* and *Invalid*. The first allows read and write access, while the second one is restricted to read-only access. Note that among all caches only one controller can be in the *Modified* state for the same cache line. However, multiple readers may access the same data in a read-only manner and therefore, the *Shared* state can be obtained by multiple entities at the same time. Note that the cache lines in the *Shared* state are always clean, which means a cache controller may simply evict the cache line without needing to write it back to memory. Needless to say, a cache holding a line in a *Modified* state needs to write the data back to memory before eviction.

MESI: Another common protocol (e.g. implemented by the Cortex X1 [Arm20]), which adds an *Exclusive* state, is called **MESI**. To understand the motivation for this version, consider the following case: Assume the CPU first reads from a memory region and then writes to the same location. Hence, a cache using only **MSI** first asks for read-only access on the bus, which puts the cache line in the *Shared* state. However, then a write is performed and thus, the controller must ask for write permissions on the bus. Observe that this adds additional latency to the write instruction even though the cache technically already contains the cache line which causes needless traffic on the bus. The goal of the *Exclusive* state is to resolve this issue. The memory controller explicitly keeps track of whether some cache currently has read-only-access to a specific cache line, which is not relevant in the **MSI** protocol (it must only know whether a cache has write permissions). Hence, once a cache requests read-only permission and the controller knows that no other cache currently has access to the same line, the controller explicitly informs the requester that it is currently the only user of this cache line. Thus, the cache may put the line in the *Exclusive* state. The advantage now is that once the CPU writes to the location, it may silently upgrade the cache to the *Modified* state without consulting the bus. However, this also means that if a cache evicts a line in the *Exclusive* state, it must explicitly notify the memory controller on the bus. The reason is that the latter cannot distinguish between whether a cache line is in an *Exclusive* or *Modified* state due to the cache having the ability to silently upgrade the line.

MOSI: This protocol adds the *Owned* state to the FSM. We briefly explain the motivation: When a cache has a cache line in the *Modified* state and another core requests read-only access, the cache downgrades the state to *Shared* and has to send the data to both the requester as well as to the memory controller. The reason is that the *Modified* state implies ownership of the data. By downgrading its own state to *Shared*, the ownership gets lost and thus, it is vital to explicitly transfer the data to the memory controller. Observe the following potential issues: The cache might need to send the data twice, depending on whether caches and memory controller are connected to the same bus [NSH+20, p. 128]. Furthermore, the memory controller commits the data to memory which might incur a bottleneck in case memory must be frequently accessed. These disadvantages are avoidable by keeping the ownership information: Recall that the *Modified* state implies ownership. Thus, if another cache requests read-only access, the cache line downgrades to the *Owned* state. It is now also read-only but dirty. This means our cache keeps ownership information and only if eviction of this cache line is required, the cache controller needs to explicitly write the data back to the memory controller.

MOESDIF: This is the protocol mentioned in the amd64 programmer reference [Adv24a, p. 192]. The additional letters **D** and **F** refer to *Dirty* and *Forward* respectively. The former is similar to the *Exclusive* state as the cache line is also read-only and can be upgraded to *Modified* without consulting the interconnection first. However, the cache line in the *Exclusive* state is clean, while this is not the case for *Dirty*. In case of an eviction, both states mandate the cache to notify the other entities but the *Dirty* state additionally requires the cache line to be sent back to memory. The *Forward* state is a microarchitectural optimization. Considering a simple **MSI** protocol, a read request must be handled by the directory or memory controller. Thus, data is fetched from memory, even though another cache might already have the same cache line which could provide the data much faster. The *Forward* state allows a cache to respond to a read request from another entity.

2.2.3 Write/Store Buffers

In the previous section, we discussed how caches reduce latency of memory accesses while keeping coherence. We want to highlight another hardware technique that is relevant for the topics discussed in this thesis. As described in the previous sections, writing to memory can be quite costly, even when using a **write-back** cache. Recall that in order to write data the cache needs to request write permissions from the memory controller. This state is exclusive unlike read-only access, where multiple threads may share the same memory region at the same time. Thus, if multiple writers need to write to the same location, they need to constantly take each other's permissions to access the line. The AMD64 Programmer's manual mentions that writes may also stall memory reads, presumably due to limited ports to the memory, which can potentially stall the entire execution pipeline [Adv24a, p. 202]. The system can reduce the impact of said situations using a **write** or **store buffer**. When executing a store, the value is not sent to the cache directly, but instead stored in the write buffer. In fact, the cache does not even require to contain the affected memory region. The actual store occurs at a later point for several reasons, such as when the store buffer is full. This might raise the question whether a store buffer simply defers the problem to a later time because the system has to commit to memory at some point anyway. In essence, the question is valid but deferring to a later point may have several advantages: For example, the system can commit all stores at the same time which can be more efficient, especially when there are several stores affecting the same cache line. In addition, the system might write back values step-by-step if it observes that memory/cache contention is low at the moment. Hence, as long as the CPU generates less writes than the write buffer can commit to memory, the latency of a store can be completely eliminated [PH17, p. 769]. In addition, consider a thread writing to the same location as a previous store that is still contained in the buffer. The write buffer can simply overwrite the value of a previous store to the same location, reducing the amount of writes that actually have to be committed to the cache and memory. This optimization is called **write coalescing**.

An important question to consider is how write buffers affect the correctness of a multi-core system. Recall that a cache coherence protocol is actually necessary to maintain assumptions of a memory-only system. Take a system where each core has a private store buffer. In a single-threaded context and for true dependencies (read-after-write), if the write has not yet been committed to memory, the read returns the value from the buffer. This mechanism is called **store forwarding** [TA13]. For a multi-threaded environment,

Thread 1	Thread 2
1: `x = 1`	1: `r1 = y`
2: `y = 1`	2: `r2 = x`

Figure 2.4: Litmus test, `x` and `y` are initialized to 0. [MHAM11]

Thread 1	Thread 2
1: `x = 1`	1: `while(flag == false) goto 1`
2: `flag = true`	2: `r1 = x`

Figure 2.5: Litmus test, `x` and `flag` are initialized to 0 and false respectively. [MHAM11]

Cycle	Thread 1	Thread 2	Coherence State of x	Coherence State of flag
1	`flag = true`		Read-only for 2	Read-Write for 1
2		`while(flag == false)`	Read-only for 2	Read-only for 2
3		`r1 = x`	Read-only for 2	Read-only for 2
4	`x = 1`		Read-Write for 1	Read-only for 2

Figure 2.6: Possible cache coherent execution of litmus test 2.5. Thread 2 reads flag $= true$ at cycle 2 and $x = 0$ at cycle 3. Taken from [NSH+20].

consider the example in Figure 2.3, which we used to explain the motivation behind cache coherence protocols. Thread 2 may only continue if it sees $A \neq 0$. When thread 1 stores $A = 1$ to its private write buffer, the value is not yet visible to 2. Nevertheless, the value must be committed to cache or memory eventually, making it visible to all threads. However, Section 2.2.1 explains how cache coherence protocols are an abstraction over a memory-only system and hence, cache coherence still preserves a total order over all memory accesses. But since write buffers are private to each core, a core observes its own writes before other hardware threads in the system. Therefore, each core might observe a different order of loads and stores. Whether a certain order is allowed depends on the memory consistency model specified by the ISA. Refer to Section 2.3 for more information.

2.3 Memory Consistency Model

It might seem that cache coherence is sufficient to model multi-threaded programs. Consider the example in Figure 2.4: We are interested in the possible values of $r1$ and $r2$. Needless to say, the result depends on the order of execution which is non-deterministic in a parallel environment. The obvious cases are $r1 = r2 = 0$ and $r1 = r2 = 1$ where thread 2 has executed its code before thread 1 and vice versa respectively. Another possible execution is that thread 1 runs Line 1, then thread 2 executes its program and then thread 1 continues to execute Line 2 which yields $r1 = 0 \land r2 = 1$. What might come as a surprise is that $r1 = 1 \land r2 = 0$ is also a possible result. The reason is that modern hardware, thanks to out-of-order execution, can decide to reorder the store at Line 1 of thread 1 after Line 2 which is valid from an isolated single-threaded perspective. Of course, one might argue that this example is impractical because the program relies on non-deterministic behavior even when assuming every line is executed in-order. Consider a more practical example in Figure 2.5. Thread 2 loops endlessly while a flag is equal to $false$. This flag could represent several practical scenarios such as a mutex to a resource or notifying another thread that new data is available for further processing. Taking the latter case, thread 2 is supposed to see the newly available data once it gets notified via the flag. However, Figure 2.6 shows a valid cache-coherent execution which yields $r1 = 0$.

Observe that a cache coherence protocol is a way to hide the abstraction over the memory subsystem in order for the caches to behave like a cacheless system [NSH+20, p. 21]. However, it does not specify how memory accesses to different locations are related to each other. This is where the **memory consistency model** or just **memory model** comes into affect. A memory model describes all possible values for a certain input that reading from memory can yield. These results are called *MC executions* [NSH+20]. Unlike in a deterministic single-threaded environment, there can be multiple options, similarly to the example given in Figure 2.4. Example models are sequential consistency (SC), total store order (TSO), the C11 Memory Model or Linux Kernel Memory Model (LKMM). Observe that a memory consistency model exists for both hardware and software. When translating from one model to another, the compiler or tool must ensure that the target model disallows all executions forbidden by the source model. The target model might be stronger in the sense that it disallows more executions than the source model. Consider an example of compiling C code using C11 atomics to the RISC-V architecture. First of all, the compiler must ensure to reorder reads and writes only as long as it does not violate the C11 memory model. However, this might not be sufficient because the hardware's weaker model allow it to reorder these operations during execution. Therefore, the compiler needs to insert special fence or barrier instructions at certain points. We introduce these instructions in Section 2.3.4.

Memory models can be formally defined. There exist several methods which can be divided into operational or axiomatic models [AMT14]. Some ISAs use these tools to model their architecture's memory behavior such as RISC-V [RIS19, p. 191]. However, a lot of formalism can be hard to understand and hence, vendors often also supply so-called **litmus tests**. These are small multithreaded code snippets that explain what outcomes are allowed and what are not [NSH+20, p. 262]. Figures 2.4 and 2.5 show simple examples for litmus tests.

2.3.1 Sequential Consistency

Sequential consistency (SC) was originally defined by Lamport [Lam79] and consists of two properties:

1. Each read and write must be committed in the order defined by the program (also called program order).

2. All memory operations are serviced via a single FIFO-queue.

The second property implies an interesting observation: There exists a single global view on memory and hence, all threads see all memory operations in the same order. In other words, consider every pair of loads-loads, loads-store, store-load and store-store occurring in the program (program order). SC ensures that all threads observe these pairs in the same order. While being very strict, it is also the most intuitive to understand. The result of these properties is that sequential consistency only allows executions which are formed by some combined sequential execution. Considering our litmus test from Figure 2.4, the result $r1 = 1 \land r2 = 0$ is disallowed by SC. The load from y must happen before the load from x by property (1). $r1 = 1$ and $r2 = 0$ implies that $y = 1$ happens before $x = 1$, considering that both variables are initialized with 0. However, this yields a contradiction because $y = 1$ before $x = 1$ does not respect the program order of thread 1.

2.3.2 Total Store Order

Another common memory model is total store order (TSO), provided by x86 for example. In general, TSO is quite similar to SC with one notable exception: A load may be reordered before a store unless the store-load pair accesses the same memory location because TSO must still preserve read-after-write dependencies. Thus, threads can observe their own write operations before other threads. However, once a write becomes globally visible, all other threads observe the same order [NSH+20, p. 39-44]. One might ask what benefits does TSO have over SC since the differences are rather subtle. The important aspect is that allowing loads to be reordered before a store enables the use of a FIFO write buffer, explained in Section 2.2.3. Thus, writes do not need to be committed to cache/memory immediately which can improve throughput. Note that the buffer may not use write-coalescing as TSO still needs to preserve store-store order unless the implementation can guarantee the preservation via some other mechanism.

2.3.3 Weak/Relaxed Memory Model

There exists no general notion of a weak memory model, but more like a family of weak memory models. SC and TSO are often called 'strong' memory models because they still preserve per-thread memory operations [NSH+20, p. 55]. Weaker or relaxed memory models allow both the hardware and the compiler to more aggressively optimize code execution. However, when certain orderings are required, additional fence instructions must be inserted which might not be necessary on an architecture such as x86. RISC-V, ARMv8 and IBM Power are prominent examples for having variations of more relaxed memory models. Even among them, the strength of their respective memory model varies. For instance, RISC-V provides multi-copy atomicity, ARMv8 defines its architecture as 'other-multi-copy atomic' and IBM Power does neither [RIS19,Arm21,Ope21]. Multi-copy atomicity and other-multi-copy atomicity describe that all writes, once they are globally visible, are observed in the same order by all threads. However, threads may see their own writes in a different order in order to preserve program order [Arm21, RIS19, p.

```
(1) li t1, 1
(2) sw t1, 0(s0)
(3) fence
(4) sw t1, 0(s1)
```

Figure 2.7: Example of a memory fence in the RISC-V ISA. Code snippet from the "PPOCA" store buffer forwarding litmus test [RIS19, p. 166].

B2-224, p. 185]. This property is inherently fulfilled by TSO (see Section 2.3.2) but for instance, the memory model of RISC-V still allows write coalescing or write reorderings which would be forbidden under TSO.

The Power ISA is an interesting case which does not enforce multi-copy atomicity. Thus, different threads may observe reads and writes in different orders and hence, there exists no global order of memory operations. On the one side, this makes writing parallel software more difficult because developers and compilers need to be more explicit about when synchronization is required. On the other side, microarchitecture designers have more freedom to speed up execution. For instance, multi-copy atomicity essentially enforces that a write buffer must be private to each thread (at least from a behavioral perspective). However, a Power core is allowed to read a value from another core's write buffer before it is committed to global cache/memory. This leads to a situation where a thread may observe memory operations in a different order than other cores.

2.3.4 Memory Fences

In the previous section, we introduced several common memory models used by more popular ISAs. This helps us understand how a core may reorder loads and stores as well as how they are observed by other threads. But what if the memory model of the hardware does not provide sufficient guarantees for a particular algorithm? In order to enforce stronger constraints, ISAs provide so-called **memory fences**. As the name might imply, a fence instruction ensures that all memory-related operations occurring before the fence in program order are observed before the ones occurring after the fence in program order. For instance, consider the code snippet in Figure 2.7. (3) shows the use of a fence instruction as provided by RISC-V. Assume that `0(s0)` and `0(s1)` are both initialized to 0 and correspond to different memory locations. Without a fence, the memory model of RISC-V would allow another core to observe the store of (4) before (2). The fence ensures that (2) is always observed before (4). On a side note, an analogous example for x86 does not require a memory fence because TSO implicitly preserves store-store ordering.

Depending on the strength of the consistency model of an ISA, different types of fences might be provided. This is particularly the case for architectures with weaker memory models because depending on the situation, a full memory fence might not be necessary. For instance, the RISC-V instruction `fence w, w` prevents any write after the fence to be observed before a write before the fence. However, loads may still be reordered freely across the fence. This might reduce the performance impact of an explicit synchronization when a complete order preservation is not actually required.

2.4 Atomic Instructions

A multicore CPU requires synchronization primitives in order to allow software developers to implement parallel algorithms. The most common family of instructions are read-modify-write (RMW) operations, such as fetch-and-add (FAA) or compare-and-swap (CAS). While these instructions consist of multiple steps, atomicity guarantees that all these steps appear to other threads as if they were executed in a single step. The question is what instructions are sufficient to implement parallel algorithms? This requires a formal way to assess the power of an atomic operation. The consensus problem states that multiple threads propose a value each and then, need to agree on one of those values. An algorithm solving the consensus problem is called a consensus protocol. We specifically focus on wait-free protocols because implementing a parallel wait-free algorithm requires that the atomic primitives of the CPU are wait-free themselves. An algorithm is wait-free if all threads finish execution in a finite number of steps. Next, the consensus number assesses the strength of a consensus protocol. This number corresponds to how many threads are able to agree on one of their proposed values. For instance, a consensus number of two states that the algorithm can only agree on a value for at most two threads. Three or more threads are not supported. Note that the consensus number can be infinite in which case there is no limit for how many threads the consensus protocol can decide upon at most [HSLS20].

Most interestingly, atomic registers are proven to have a consensus number of one. Informally, consider two threads A and B deciding on the value. The question is how can the system determine which thread accessed the register first? Consider the case where A reads from the register and B also executes an operation (either a read or write). There exist two possibilities: In the first one, B executes its operation first and hence, it won the consensus. In the other one, A reads first, then B applies its operation and thus, A won the consensus. However, both scenarios are indistinguishable to B because the side effect of A reading from the shared register is not visible to B. There are also other cases which need to be formally considered, however, the argument is similar, namely that neither thread cannot differentiate the outcome. Therefore, atomic registers have consensus number of one. Thus, even if one was to built a CPU with shared registers between cores, simple loads and stores to those registers would not suffice for implementing parallel algorithms. This shows the need for dedicated atomic instructions. Let us consider the family of RMW operations, such as a FAA operation. This operation has a consensus number of exactly two. Very informally, consider a protocol where each thread stores its proposed value in an array where the id of the thread corresponds to the index in the array. Next, a decision variable is initialized to zero and each thread increments the value via a FAA. Recall that a FAA returns the current value and atomically increments the value by one in the memory. The thread loading the value zero has won the consensus, while all other threads, which must receive a non-zero value by definition of a FAA, have

```
cas:
lr.w t0, (a0)       # Load value from memory
bne t0, a1, fail    # a1 contains expected value
sc.w t0, a2, (a0)   # Try to update. a2 contains new value
bne t0, cas         # Retry if store-conditional failed
li a0, 0            # Indicate success as return value
jr ra               # Return
fail:
li a0, 1            # Indicate failure as return value
jr ra               # Return
```

Figure 2.8: Example for a CAS function using LR/SC instructions provided by RISC-V ISA. Code snippet from [RIS19, p. 50].

lost and thus, the winning thread's value will be chosen. In case of two threads, the losing thread clearly knows who the winning thread is because the array has size two and one element belongs to the losing thread and thus, the other remaining element must contain the decided value. However, in the case of more than two threads, the losing threads cannot determine which thread won the consensus. Thus, FAA only has consensus number two.

Finally, let us consider a CAS operation. A CAS has three operands, an address, an expected value and an updated value. First, the currently stored value is loaded from the given address and compared to the expected value. Next, if they match, the CPU stores the updated value at the same address. All of this happens atomically. We construct a consensus protocol based on a CAS. A decision variable x is initialized to zero. Next, each thread proposes its value in an array where the thread id corresponds to the index in the array. Each thread tries to exchange the value stored at x with its own thread id. Note that we assume that there cannot be a thread with an id zero. Only one thread will be able to replace the initial value stored at x, which is the winning thread. All other threads see the winning thread's id so they also know which thread won the consensus. Thus, the consensus number is infinite, assuming that integers are unbounded.

Finally, we want to introduce universality: A primitive is universal if and only if it is powerful enough to support arbitrary wait-free algorithms. An operation with consensus number n is universal for a system with at most n processes [Her91, p. 125]. Thus, we can conclude that a compare-and-swap operation is sufficient to implement parallel algorithms for any system. ARMv8 and amd64 provide dedicated instructions called `CAS` and `CMPXCHG` respectively. The former is atomic by default, while the latter requires a `LOCK` prefix [Adv24b, Arm21]. While these instructions would be sufficient in theory and thus, can be used to implement other RMW operations such as FAA, most ISAs still provide dedicated fetch-and-add, fetch-and-and, etc. instructions, presumably for performance reasons as they are often required in practice.

Some architectures, most notably RISC-V, Power and ARMv8, support even more powerful primitives called load-reserve (other names include load-exclusive or load-link) and store-conditional (also named store-exclusive) [Arm21, RIS19, Ope21]. We continue with load-reserved (LR) and store-conditional (SC) but the other names can be used interchangeably. An LR instruction loads a value from memory and additionally creates a reservation. A store-conditional stores a value to memory if and only if such a reservation to the specified address exists. This reservation is cleared by stores from the same or other threads but may also be cleared spuriously e.g. due to a context switch. Observe that this functionality can be used to implement a compare-and-swap and other RMW operations such as for RISC-V in Figure 2.8. The question might arise why such a more verbose construction is required if a CAS already provides an infinite consensus number? LR/SC is popular for RISC architectures because they might not have an instruction format for three source operands such as RISC-V [RIS19, p. 48]. Another reason is that, albeit being the same consensus class, LR/SC is slightly more powerful than a CAS because it does not suffer from the ABA-problem [HSLS20, p. 244]. An ABA-problem occurs in the following situation: Suppose thread 1 loads value A from location a and then, pauses before applying a CAS to exchange it for some other value. In the meantime, thread 2 runs and applies the CAS to exchange A for B at location a. Next, it runs another CAS to exchange B back to A. Next, thread 1 continues with its CAS which succeeds because the stored value is still A as expected for thread 1, even though the stored value A does not correlate with the value A that was originally loaded by thread 1. However, if the code used a LR/SC pair to conduct the exchange, thread 1's operation will fail because another store was executed between its load and the store-conditional. ABA-issues commonly occur in memory management scenarios where internal data structures are recycled for reuse [HSLS20, p. 241].

Recall that when an LR-like instruction is executed, the core makes a reservation of the given address. Note that the hardware usually does not remember the entire address but ignores some least-significant bits. Thus, a single reservation covers an entire memory region and the size is defined by the reservation granule [Ope21, p. 1049]. Thus, as long as the address of a subsequent conditional store translates to the same reservation region, the store will succeed albeit the address not being equal. The reservation granule can vary between MiAs [Arm21, p. B2-312] [Ope21, p. 1049].

2.5 Virtual Address Space and Address Translation

When user space programs access memory on modern systems, the addresses do not correspond to physical addresses of the real memory. Instead, they work with virtual addresses which get translated into physical addresses on access. Similarly to how a cache is organized in cache lines, physical addresses are separated into pages. The mapping is defined by a page table which is usually unique for each user space process. Usually, a dedicated register holds the address to the beginning of the current page table. The upper bits of a virtual address are then mapped to a physical address while the remaining bits designate the offset within the page [RIS21, p. 79]. Using a virtual address space has many advantages:

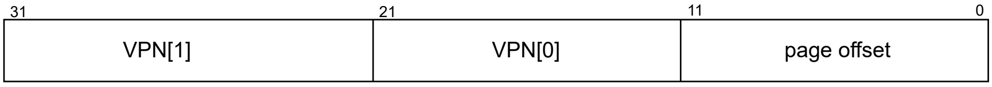

Figure 2.9: Structure of a RISC-V Sv32 virtual address. [RIS21, p. 79]

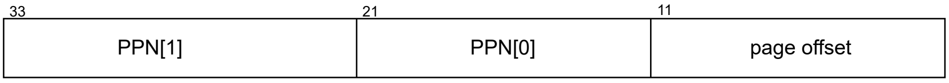

Figure 2.10: Structure of a RISC-V Sv32 physical address. [RIS21, p. 80]

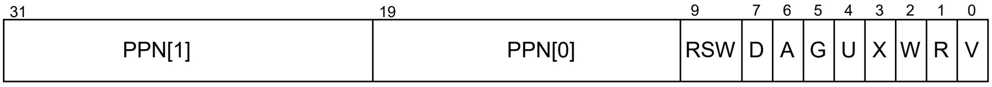

Figure 2.11: Structure of a RISC-V Sv32 PTE. The field *RSW* is reserved for software. Flags are dirty *D*, accessed *A*, global *G*, accessible to user mode *U*, execute *X*, write *W*, read *R* and valid *V*. [RIS21, p. 80]

- **Address space separation**: The virtual address space differs for each process, providing some degree of isolation between multi processes. For instance, this prevents one process to maliciously or unintentionally modify the memory of another process, potentially corrupting it. [PH17, p. 819]
- **Physical memory management**: Since the mapping can be freely configured by the operating system, pages in the physical memory can be freely moved around. Furthermore, operating systems employ a technique called 'swapping', allowing to store less frequently used pages on the hard disk in order to use the memory for other purposes. [PH17, p. 829]
- **Memory Protection**: The type of access to memory regions can be restricted. For instance, the text section containing the executable data should not be writable in order to prevent bugs in software to be exploited, allowing the attacker to dynamically rewrite the machine code. Similarly, memory pages marked writable, such as the heap or stack, should not be executable. macOS on Apple Silicon even enforces a 'write xor execute' scheme on all processes [app].
- **Memory-mapped I/O**: Most modern operating systems utilize the virtual address space to allow processes to map files from the file system into their address space. A mapped file is usually not copied entirely into memory but rather parts of the data once they are actually accessed. In essence, buffer management is delegated to the kernel. This allows programs to automatically benefit from other features mentioned above, such as swapping and memory sharing when several process map the same file into memory. [IEE18]

Virtual memory needs to be supported by the hardware to properly work. The piece of hardware responsible is called a memory management unit (MMU) [Arm21, p. D8-6565]. When the processor wants to access some location in memory via a virtual address, the address must be translated into a physical address. As an example, we want to present the approach of the RISC-V architecture. RISC-V provides several sizes of the virtual address space. We consider the smallest mode, Sv32 for a 32-bit paged virtual memory system [RIS21]. A page is 4 KiB and thus, the 12 least significant bits of a virtual address corresponds to the offset. The remaining 20 bits are called the virtual page number (VPN) which are split into two 10 bits parts. The reason is that RISC-V employs a multi-level page table. If RISC-V did not do this, then a single page table would require at least 2^{20} entries. Since each page-table entry (PTE) uses four bytes, this would lead to a page table of four MiB, even though most entries might not even be used. Now imagine having just a 1000 processes running on the system, where all page tables alone already consume almost 4 GiB. Having a multi-level page table allows to reduce the size of the page table. In the case of RISC-V, splitting the address into two 10 bits parts decreases the size of the page table to $2^{10} = 1024$ entries. Since a PTE needs four bytes, an entire page table conveniently fits on a single page itself.

The translation begins by using the VPN[1] as an index into the page table. The address of the current page table is stored in the `satp` register. Next, look at the page entry as seen in Figure 2.11. The physical page number (PPN) corresponds to the upper bits of a page-aligned physical address but its meaning depends on several flags. Most importantly, the V flag determines whether the entry is valid. If the flag is not set, a page-fault exception is raised. A page-fault does not necessarily mean that a program accessed an invalid address. It could also mean that the kernel swapped a memory page out onto the disk and thus, must now reload the page into physical memory. Next, the processor core checks the permission flags R, W and X corresponding to read, write and execute. If all of them are unset but the page is valid, then the PPN points to a second-level page table. The process is then repeated but now the VPN[0] is used as an index. Otherwise, the PTE entry is a leaf entry. Last but not least, the processor checks whether the permission flags align with the current operation and an exception is raised otherwise. Finally, the PPN is used to reconstruct the physical address.

As observed in Figure 2.11, the PTE contains additional flags, most notable the D (dirty) and A (accessed) flags. The former indicates that a page has been written to, while the latter corresponds to any type of read, write or execute access. Both are set by the hardware and can be used to optimize memory management in the kernel because unlike memory caches, the page table is usually managed by software [PH17]. For instance, the D flag can be used to determine whether a memory mapped file needs to be actually written back to disk. The A flag might be helpful to determine what memory page to evict in case of high memory usage. In addition, the RSW field is ignored by a RISC-V implementation and specifically designed to be used by the operating system.

Observe that accessing memory via a virtual address introduces several additional steps with potential overhead accordingly. Continuing with our RISC-V example, the translation process itself takes two memory accesses in the worst case, one for accessing the root page table entry and a second one for the second-level page table if applicable. Only then can the actual memory access be conducted. In order to minimize this impact, hardware designers have introduced a cache for the translation, called the translation lookaside buffer (TLB) [PH17, p. 834]. The behavior is very similar to a memory cache as the TLB maps a virtual address to a physical one. Since the TLB is not exhaustive, it stores a tag of the virtual address to ensure the entry matches. If an entry cannot be found in the TLB (a TLB miss), the hardware first tries to fetch the entry from the page table. If the entry is actually invalid, a page-fault is raised. An L1 TLB is usually fully associative because misses are rather expensive [PH17, Arm20]. Some microarchitectures, such as the Cortex-X1, additionally store an address space identifier (ASID) to indicate what context the entry belongs to. This allows to switch between processes with different address spaces without having to invalidate the TLB first [Arm20].

CHAPTER 3

Related Work

3.1 Cache Simulation

There exists a wide range of cache simulators, all varying in terms of goals, use case and hence, their design. Brais et al. analyzed more than 20 different simulators [BKP20]. The authors classified them into several categories based on different attributes. For starters, a simulator can be functional or timing. The former merely focuses on the functionality of a cache, while the latter also takes latencies on the interconnect, bus port widths and other microarchitecture-specific properties into account. Next, there exist several ways of how a simulation may be conducted. For instance, some simulators execute or emulate the target program to imitate the behavior of a cache, while others use previously recorded memory-traces which are then replayed by the cache simulator. Finally, some simulators are designed as standalone programs but many have been developed as part of a general simulator, which also replicate other parts of a computer system.

Dinero IV is an older but often cited cache simulator [EH]. It is capable of simulating multiple-cache levels and separate data and instruction caches based on memory traces. Note that Dinero IV can be used only for basic statistical purposes such as cache miss rate. It does not actually store and load data and also does not consider any timing-related information. Hence, cache coherence is also not in the scope of this simulator.

gem5 is a widely popular full-system simulator that supports simulation of several ISAs, types of memory, GPUs and more with a varying degree of accuracy. The core parts are written in C++ but Python is used for setting up the configuration of the simulator [BBB+11, LAA+20]. It also includes two types of cache simulation, called 'classic' caches and Ruby. The former implements a highly-configurable MOESI snooping-based protocol, while the latter is a framework to model arbitrary cache coherence protocols. Ruby provides a domain specific language called *SLICC* and is used for specifying the set of states, transitions and resulting actions in the state machine. gem5 ships with several cache coherence protocols written in SLICC, such as a directory-based two-level MESI. Ruby is designed for accurate timing simulation and hence, can be considerably slower compared to functional simulators.

Ubal et al. developed a simulation framework for CPU-GPU computing called Multi2Sim [UJM+12]. Initially, the simulator was designed for simulating the x86 instruction set in combination with a specific AMD graphics card. However, the simulator has since been extended to support several ISAs and graphics card microarchitectures [ea]. Multi2Sim uses a fixed 6-state coherence protocol. Several attributes, such as associativity, block size and replacement policy, can be configured via configuration files. The simulator separates the cache definition from the interconnect and hence, allows users to define a flexible topography of the cache hierarchy. The simulator itself is functional and records basic statistics, such as amount of memory accesses and cache hit rate.

Cachegrind is a high-precision tracing profiler [cac]. It is distributed as part of valgrind, a popular debugging and profiling tool. Cachegrind's primary purpose is to accurately measure the amount of instructions executed. In addition, the tool allows simulating the behavior on the memory hierarchy. The hierarchy is fixed to a private level 1 data and instruction and a shared level 2 cache. Configuration options are limited to total size, associativity and line size. The authors mention that the behavior approximates an AMD Athlon from 2002. Note that cachegrind does not emulate the given program, unlike gem5 or Multi2Sim. The program is still executed by the host CPU natively. Hence, this also limits design space exploration (DSE) because other microarchitectural properties or even ISSs cannot be simulated.

Jaleel et al. developed the cache simulator called CMP$im [JCLJ08]. Unlike previously mentioned simulators, it is not execution-based but instead relies on instrumentation using the Pin analysis tool [LCM+05]. CMP$im inserts instrumentation code for every memory access which is then redirected through the cache simulator. Users may define typical attributes such as cache size, associativity and replacement policy but also configure the hierarchy in terms of levels whether a level is shared or private as well as inclusion policy. According to the authors, CMP$im employs a MESI-based cache coherence protocol.

pycachesim is another functional cache simulator written in Python [pyc]. It can simulate arbitrary inclusive memory hierarchies with support for several replacement policies, arbitrary associativity and cache line sizes as well as write policies, such as write-through and write-allocate. However, its focus lies on simulating single-core hierarchies and hence, does not consider cache coherence at all. There exists no state-tracking for cache lines except for whether they are dirty. The simulator is designed to be used in Python but the core has been written in C to ensure high performance.

Next, MultiCacheSim is a simulator focused on coherence simulation [mul]. The simulator allows allocating an unbounded amount of caches that can be accessed concurrently. The project includes an implementation of the MSI and MESI coherence protocols but specifically mentions its openness for extension. In addition, it is written in pure C++ and does not require any external dependencies. One drawback of this simulator is that it does not allow defining a memory hierarchy. Hence, the simulation only differs between reads serviced from the cache or remotely. This means that more complex multi-level cache coherence protocols cannot be sufficiently simulated.

zsim is a highly parallel simulator with the stated goal to efficiently simulate even 1024-core CPUs while retaining a high-level of accuracy [SK13]. The authors explain that a lot of cache simulators pass messages in order to handle synchronization between conflicting cache accesses. However, this introduces a significant overhead and thus, they opted for a shared-memory design instead. One challenge of developing a parallel CAS is that accuracy might suffer because parallel execution is non-deterministic and hence, could impair the accuracy of a real system. A purely single-threaded simulator, such as gem5, does not suffer from this issue. However, this approach also limits the scalability of a system. Note that multicore systems, even with 3-digits core counts, have become more common and therefore, scalability will be even more relevant. The authors of zsim showed that a high level of accuracy can be achieved even with a parallel design. They validated their simulator against a real 6-core Intel Xeon CPU using the full SPEC CPU2006 suite and according to their results, their average L1 data cache misses per thousand instructions (MPKI) is off by 1.14. While they also briefly mention a comparison to other simulators, they do not compare themselves to gem5 as it was validated against a different CPU architecture. Furthermore, as mentioned by [BKP20], note that validation against real hardware may lead to over-generalization and hence, simulating other types of systems could lead to varying degrees of accuracy. However, we should mention that zsim is limited to x86-64 simulation and importantly, does not allow configuration of the cache coherence protocol, which we deem important for our purposes of VADL.

McSimA+ is another x86 timing simulator with a focus on accurately simulating manycore MiAs [ALOJ13]. While similar to zsim, McSimA+ focuses on simulating asymmetric architectures where the same die contains two different sets of cores. One is usually a set of high performance cores, while the other consists of slower but more energy-efficient ones. The cache simulation part of McSimA+ is highly detailed and provides three coherence protocols used in real MiAs. For instance, the 'DRAM-dir', found in the Alpha 21364, stores the directory in memory but additionally employs a cache situated in the memory controller. Furthermore, McSimA+ respects the topology of the interconnect in its timing simulation.

Miller et al. developed Graphite, a parallel and distributed simulator for manycore systems [MKK+10]. Similarly to zsim and McSimA+, the goal of Graphite is to simulate manycore machines efficiently, while utilizing the power of a multicore host. However, the authors take a different approach by distributing simulation threads across multiple machines, which allows them to employ more parallelism at the cost of sacrificing accuracy. The authors specifically mention that they do not strive for cycle-accurate simulation. Most noteworthy, the simulated program is not required to be designed for distributed execution, as any pthreads-based program is compatible. Graphite is divided into modules which can be separately configured and exchanged. The memory module is responsible for imitating the cache hierarchy and coherence. The authors implemented two cache coherence protocols: A directory-based MSI protocol, which can be fixed to a certain number of sharers, as well as the LimitLESS protocol [CKA91]. The memory subsystem uses a fixed hierarchy, consisting of a private L1 and L2 cache. However, they claim that thanks to Graphite's flexible architecture, a new module using another memory configuration can be easily developed and used instead. Graphite uses a message-based architecture for distributing events (e.g. a cache coherence request) and communication occurs via TCP between hosts. The authors investigated the performance overhead of the cache coherence protocols using a benchmark that runs almost perfectly parallel and hence, little sharing is needed. However, read-only code, such as shared libraries, need to be shared across all hosts. Hence, the authors limited the directory to fixed number of sharers (e.g. 4) which means that w.l.o.g. if a fifth sharer requests some read-only data, the memory controller is forced to evict the data from the previous four sharers. Thus, the authors were able to observe almost perfect scaling with the number of threads until it increases above the set number of sharers in the directory. While the paper also includes an evaluation of the cache miss rates, the authors do not mention any validation of their cache simulation.

Carlson et al. used Graphite as a foundation for their own simulator, called Sniper [CHE11]. Graphite models the timing of a system by assuming fixed latencies for instruction execution, cache misses and failed branch predictions. Graphite specifically does not take into account that superscalar execution may hide some of these latencies. The authors believe that this model is not accurate enough but needless to say, true cycle-accurate simulation requires considerably more computational power. Hence, they propose a middle ground called interval simulation. The idea is that miss events (branch mispredictions, cache and TLB misses) divide the execution into intervals. The simulator keeps track of instruction windows that are necessary for tracking the simulated execution time. These windows match the size of the reorder buffer of a CPU. This allows Sniper to more accurately determine latencies. For instance, if the instruction cache reports a miss, then the penalty is added to the simulated time. If the CPU triggers a misprediction, a fixed latency plus the depth of the reorder buffer will be added as the latter must be completely drained and refilled. Regarding the cache simulation, Carlson et al. also made several contributions to Graphite. Most notably, they added a cache hierarchy with shared caches, while the caches in Graphite were completely private. This causes additional overhead in the simulation but it should be noted that the authors of Sniper focus on running the simulator on the same machine, while Graphite is specifically designed for distributed computing across multiple hosts.

Ren et al. developed HORNET, which is a multithreaded and highly-configurable cycle-accurate simulator [RLC+12]. Their focus lies on variable memory hierarchies and interconnect topologies. HORNET ensures coherence either via a simple MSI protocol or a distributed shared cache.

Finally, SiNUCA is a simulator with a focus on validation [AVD+15]. Instead of comparing the simulator directly with real hardware, they developed a suite of microbenchmarks that were used to reverse engineer undocumented behavior of the target machines which include an Intel Core 2 Duo and Sandy Bridge processor. It only provides a MOESI cache coherence protocol. However, the authors claim their simulator is highly flexible and can be easily extended for other ISAs and MiAs.

3.2 Processor Description Languages

Processor description languages (PDLs) have been invented for designing hardware architectures. A specification may consist of several aspects such as instruction set architectures (ISAs), microarchitectures (MiAs) or other software- and hardware-related features. A PDL specification can then be used to generate hardware prototypes, test programs, compilers and/or simulators. According to [MD08], PDLs can be separated into three categories. On one side, structural PDLs operate on the RTL-level, making them suitable for cycle-accurate simulation and hardware synthesis. On the other side, behavioral PDLs define the semantics of an ISA but do not concern themselves with details of a hardware implementation. Thus, they are suitable for ISSs and compiler generation. Mixed PDLs include both structural and behavioral features.

Expression is a mixed PDL supporting a wide range of processor types, including digital signal processors as well as VLIW and superscalar architectures [HGG+99]. The structural part is separated into individual components, which can be a pipeline stage, data path, port, cache or memory among other features. Since the language is inspired by Lisp, properties are described as nested lists. Caches can be configured in terms of line size, associativity, replacement policy, write policy and access time. We think it is noteworthy that the language features a special operation mappings section, which allows handcrafting certain ISA-specific compiler optimizations. For instance, a multiplication, where the immediate value is a power of 2, can be replaced by a corresponding shift-left instruction. Expression follows several automation methodologies such as DSE, compiler, simulator and hardware generation.

LISA is also a mixed PDL. The behavioral section allows to define instruction encoding, syntax description and semantics. The latter is based on the C language. The structural side of LISA provides ways to describe the timing behavior of registers and individual instructions. The main goal of the language is to allow designing on a high level of abstraction, which is the behavioral side of the system, but go into more detail as needed. If necessary, LISA provides all constructs to enable cycle-accurate simulation. However, based on our research, LISA is not capable of defining caches nor memory hierarchies.

nML is an industry-proven PDL [FVPF95]. It consists of a structural part which defines storage elements of the architecture. The instruction set is described with an attributed grammar using AND- and OR-rules. Depending on the generated artifact, the language allows defining additional attributes. For instance, the *syntax* attribute correlates to the assembler syntax of an instruction, while the *action* attribute describes its semantics. While nML does not support cache definitions, users can describe networks and buses which are called *transitory* storage. Unlike with static storage, the system cannot read from a transitory component more than once.

ISDL is a behavioral PDL focused on VLIW architectures and supports assembler, compiler and ISS generation [HHD97]. Its focus clearly lies on tool generation and hence, allows inclusion of C code as part of an ISDL specification which will be injected into the according artifact. An instruction definition is separated into an encoding, semantics as well as timing information. In addition, the language enables defining constraints which is important for VLIW architectures where grouping several instructions is possible.

MIMOLA was one of the earliest PDLs [Mar84]. Its purpose originated in the need to develop specialized chips that could execute a specific algorithm in hardware [MD08, p. 35]. Unlike other hardware description languages at the time, like VHDL or SystemC, MIMOLA was primarily designed for hardware synthesis. The language allows defining programs in PASCAL-inspired syntax. The program can be an instruction-set interpreter in which case arbitrary code can be executed by the synthesized hardware. MIMOLA also allows to describe structural aspects of a program. MIMOLA itself has no built-in notion for registers or memory and hence, the programmer is responsible for defining modules which model their features accordingly.

ArchC is a PDL based on the hardware description language SystemC [ARB+05]. The authors state that SystemC is too generic to enable automatic tool generation and hence, they increased the abstraction level with ArchC to achieve this. Both ArchC and SystemC consist of a set of C++ libraries in order to enable hardware modeling. Hence, the language follows an object-oriented approach. Resources (memory, register banks, pipeline, etc.) are declared as member variables of an architecture. The constructor can then be used to setup additional properties of these components. Caches can also be defined and configured including cache line size, associativity, replacement policy. The constructor can then be used to establish a connection between the different caches as well as memory, which yields the memory hierarchy.

Sail is a more recent, purely behavioral PDL [ABC+19]. The language is primarily designed for describing the semantics of an ISA. It enables generation of simulators, ISA tests and theorem-prover-definitions for Coq, Isabelle or HOL4. Furthermore, engineers can model memory consistency model in Sail, which can then generate rmem files. rmem[1] is a tool incorporating an operational model to describe memory model semantics. Using an ELF executable or litmus test, the tool enumerates all legal executions.

[1] https://github.com/rems-project/rmem

CHAPTER 4

Implementation

This section explains the design choices made to support atomic instructions, caches and write buffers in VADL as well as the necessary changes to the frontend. Before talking about the implementation, we want to highlight a certain aspect of this section. While VADL provides documentation and loads of examples, including a mature Aarch64 and RV32IM implementation, it lacks a formal specification. This thesis does not aim to formalize VADL, however, we want to contribute first steps in this direction by providing a specification for the changes made to the language in the context of this thesis. If the VADL frontend deviates from the specification, then this is considered a bug. As an example as well as a first specification, we consider our definition of *implementation-defined behavior*, slightly adapted from the C17 standard [ISO18]:

VADL Specification 1
Implementation-defined behavior is unspecified behavior where each implementation should document how the choice is made.

In other words, if a VADL specification contains implementation-defined behavior, the consequences can vary from generator to generator.

Last but not least, we want to highlight that VADL only knows about *bits* but has no concept of *bytes* or *words* because ISAs often assume different sizes. However, most architectures commonly used have agreed on these definitions at least being a multiple of eight. While VADL does not impose such a restriction (e.g. defining a word to be the size of 11 bits is possible), we implicitly assume 1 byte = 8 bits in this thesis for simplicity.

4.1 Atomic Instruction Support

In Section 2.4, we explain why an ISA needs to provide atomic instructions in order to correctly implement parallel algorithms. Thus, VADL must be able to allow designers to

Abbreviation	Description
LR	Load-reserved
SC	Store-conditional
CAS	Compare-and-swap
SWAP	Swap
FAM	Fetch-and-modify
AM	Atomic modify

Table 4.1: Common definitions for several atomic primitives.

define atomic primitives. We begin by explaining our design rationale and then discuss how our solution is implemented in the VADL frontend.

4.1.1 Language Design

We have shown that LR/SC are the most powerful primitive because they do not suffer from the ABA-problem and their consensus number is infinite. Hence, VADL must clearly allow defining these type of instructions. Nonetheless, most ISAs support other dedicated atomic instructions as a convenience. Before designing a solution for the VADL language, we look at several ISAs to understand the primitives they provide and should be able to be implemented by VADL. Since many instructions are similar but might have different mnemonics or names in general, we define our own notion for several atomic primitives, defined in Table 4.1. We distinguish coarsely between basic types of operations. FAM and AM are similar as both atomically apply an operation to the memory. The former additionally loads the old value (usually into some register). We do not distinguish between the type of modification applied in an AM and FAM because, in our view, the actual modification function is not relevant for the general semantics of these instructions.

We consider several architectures in Appendix A, including RISC-V, Power, Hexagon, amd64 and ARMv8, and map their provided instructions to our notion of atomic primitives. We observe that most ISAs specify LR and SC instructions. amd64 is a notable exception. As seen in Tables A.1, A.5 and A.6, RISC-V, Power and ARMv8 provide a rich set of FAM and AM instructions. RISC-V and ARMv8 specify dedicated mnemonics while Power uses a generic instruction where a third operand (called *function code*) specifies the type of modification operation. amd64 handles atomic operations quite differently. First, atomicity is provided via a special **lock** prefix supported on several arithmetic and logical instructions. However, this means that if a developer or compiler wants to implement FAM operations they need to rely on a CMPXCHG instruction except for addition and subtraction due to the existence of the XADD instruction.

From this, we can derive several aspects:

- LR/SC is an essential functionality for most ISAs. Thus, VADL needs to provide first-class support.

```
[ lock ]
instruction FAA : FormatB = {
    MEM( addr ) := X( res ) + MEM( addr )
}
```

Listing 4.1: Fetch-and-add example using the *lock annotation*.

```
[ lock ]
instruction CFAA : FormatB = {
    if zeroflag then
    {
        MEM( addr ) := X( res ) + MEM( addr )
    }
}
```

Listing 4.2: Conditional fetch-and-add example using the *lock annotation*.

```
[ lock ]
instruction DFAA : FormatB = {
    MEM( addr ) := X( res ) + MEM( addr )
    MEM( addr + 1 ) := X( res ) + MEM( addr + 1 )
}
```

Listing 4.3: Double fetch-and-add example using the *lock annotation*.

- FAM and AM operations can be quite diverse. VADL should offer the flexibility to include arbitrary modifying operations and not be limited to a specific set of instructions.

For the second point, we can take direct inspiration from the ISAs presented in Appendix A. Consider amd64 and its **lock** prefix: From a syntactical perspective, it provides a convenient mechanism to mark an arbitrary instruction as atomic without the need to remember additional mnemonics. VADL supports annotations which can be attached to several entities of the language. Using this feature we can syntactically resemble the lock prefix used by amd64 as seen in Listing 4.1. The annotation enforces that all memory accesses within the instruction definition are atomic, which we achieve by locking the appropriate memory locations or cache lines. However, this approach has several disadvantages:

1. Since we need to lock certain memory locations or cache lines, the order of locking matters in order to prevent deadlocks. Consider the definition of a double FAA in Listing 4.3. Our analysis might determine that we first lock location `addr` and then `addr + 1`. Next, assume we have an analogous definition of a double fetch-and-sub (FAS). However, the analysis might determine that we first lock `addr + 1` and then `addr`. We observe that if thread A executes a double FAA and thread B a double FAS on the same address in parallel, the system might deadlock because A manages to lock `addr` and B to lock `addr + 1` but they cannot progress because they need to wait for each other to unlock their addresses.

   An easy solution to this problem is to lock addresses in a uniform manner either by going upwards (lowest address first) or downwards (highest address first). We could ensure that our analysis deterministically locks addresses in the order they are specified and put the responsibility on the developer to ensure this order matches some global order of locking addresses either up- or downwards. However, considering we want to support arbitrary atomic instructions, assume an instruction that takes two addresses to be locked from two registers as an input. In this situation, the chip designer also needs to conditionally ensure their orderings at runtime to maintain the correct order. However, this makes our analysis more complicated because the lock order is not static anymore. In addition, this introduces the problems mentioned in the subsequent paragraph 2.

2. The **`lock`** annotation might be too restraining because it locks all memory locations accessed within the instruction definition. However, the chip designer might want to only lock certain locations used. Another example would be a more complex system having several memory interfaces and only one of them should be locked. In addition, consider a conditional instruction, such as in Listing 4.2, which only executes if a certain flag is set. Here, the lock annotation might needlessly lock the respective memory locations in case the flag is not set. This issue could easily be fixed by locking locations only when they are actually accessed. However, this introduces further semantical questions: When will the lock be freed again? At the end of the current scope/block statement? This might violate atomicity guarantees of the lock annotation if e.g. memory is accessed again unconditionally after the if statement. We think this leads to a lot of implicit behavior which a VADL developer might not expect.

We conclude that while our proposed lock annotation seems very convenient to use, it induces a lot of implicitness that can lead to the aforementioned problems. This led us to a more explicit approach:

```
instruction FAA : FormatB = {
    MEM( addr ).lock
    MEM( addr ) := X( res ) + MEM( addr )
}
```

Listing 4.4: Fetch-and-add example using the *lock method call*.

Unlike our previous example in Listing 4.1, the VADL developer must explicitly provide the address and memory interface via a lock method call. The location will be exclusive until the end of the current scope (e.g. block statement), solving 2. For issue 1, the designer is fully responsible for preventing deadlocks. What about the situation, where an instruction locks two arbitrary locations received as input via two registers? Since the locking is now explicitly part of the control flow, the designer can use an if statement to check which address is lower or higher and lock accordingly. Technically, the lock method call solves our aforementioned issues. Semantically, this approach has several caveats. The developer must know that the **`lock`** call has a side effect that lasts until the end of the current block statement. Hence, a beginner reading the same code listing might not be able to deduce this knowledge purely from its syntactical structure. Let us consider another design:

```
instruction FAA : FormatB = {
    let addr = X( arg ) + offset in
    lock MEM( addr ) in {
        MEM( addr ) := X( res ) + MEM( addr )
    }
}
```

Listing 4.5: Fetch-and-add example using the *lock statement*.

This version is technically equivalent to the method call in Listing 4.4, however, the scope of the lock becomes more obvious in our opinion. To be more precise, the lock is only held in the statement following the **`in`** keyword. Seasoned VADL developers might notice a similarity to the *let statement* as it has similar semantics: A *let statement* assigns a value to a variable which is visible only within the statement after the **in** keyword. Thus, the *lock statement* introduces a similarity to other language features, which should keep the complexity of VADL reasonable. Finally, we provide a formal definition:

VADL Specification 2 (Semantics of *lock statement*)
A ***lock statement*** **`lock`** `m<n>(l)` **`in`** `inner` *locks the memory instance m at location l and all following n bytes before its inner statement. The lock is released it at the end of the inner statement. All memory operations occurring within the range $[l, l+n)$ and while the lock is held, are observed as a single atomic step.*

Note that $<n>$ can be omitted if $n = 1$.

```
 1 instruction set architecture ISA = {
 2     register reservedAddress : Address
 3
 4     instruction LR : FormatB = {
 5         let addr = X( arg ) in {
 6             X( res ) := MEM( addr )
 7             reservedAddress := addr
 8         }
 9     }
10
11     instruction SC : FormatB = {
12         let addr = X( arg ) in
13         lock MEM( addr ) in {
14             if reservedAddress = addr then {
15                 MEM( addr ) := X( res )
16                 X( 0 ) := 0
17             } else {
18                 X( 0 ) := 1
19             }
20             reservedAddress := 0x0
21         }
22     }
23 }
```

Listing 4.6: Example for an LR/SC implementation using existing VADL primitives.

When considering our notion of atomic primitives from Table 4.1, the *lock statement* allows us to implement CAS, SWAP, FAM, and AM operations because we can lock one or more memory locations and apply arbitrary operations before freeing the lock again. Thus, all operations within a *lock statement* seem atomic from an external observer. LR and SC are a bit more difficult to implement with a *lock statement*. The reason is that the other primitives are completely isolated. In other words, their entire effect can be applied within a single instruction semantics. However, LR/SC are separate operations by definition but work only in conjunction with each other. However, the *lock statement* is restricted within the definition of a single instruction. This means we need a way to cross instruction boundaries via some global state. Several ISAs define this state as *reservation state* [RIS19] or *marking* [Arm21]. First, we analyze a set of ideas to model reservations in VADL:

1. **Utilizing existing ISA-VADL primitives**: VADL already provides sufficient functionality to define LR/SC-type of instructions. Listing 4.6 shows an example of how they could be implemented. The `LR` (Lines 4-9) instruction stores the reserved address in a register, while the `SC` counterpart (Lines 11-22) stores the value to memory if and only if the given address matches the address reserved in the register. Note that we take advantage of our newly designed *lock statement* (Line 13) in order

```
1 micro architecture MIA = {
2     [ granularity 28 ]
3     [ reservation manager ]
4     logic reservedAddress
5 }
```

Listing 4.7: Example for an LR/SC reservation manager using the **logic** element.

to ensure that no other processor can access the same memory location between the address checking (Line 14) and the actual store (Line 15). Finally, we reset the register to zero in either case and store a status value to register `X( 0 )` to notify the program about the outcome of the conditional store. Needless to say, resetting the reserved address must be done in any type of memory write in order to comply with the basic semantics of LR/SC.

2. **Utilizing existing MiA-VADL primitives**: The way a processor handles memory reservation could be defined in the MiA section of a specification. VADL provides the powerful **logic** element, shown in Listing 4.7. Line 3 specifies the primary purpose of the logic element, which we decided to call `reservation manager`. All other annotations describe the properties of the element. This example contains the `granularity` attribute on Line 2. Its purpose is to describe the size of a marked region. Recall from Section 2.4, that this property may vary between different MiAs.

3. **Provide first-class primitives for reservations**: Since LR/SC primitives are crucial for many ISAs, VADL could provide first-class functions to support such constructs. Listing 4.8 presents an example with an `LR` instruction (Lines 2-5) which loads a value from memory and specifically requests exclusive access to it. The function call to `loadExclusive` loads the value from cache or memory and additionally registers a reservation. We decided to choose the name `loadExclusive` instead of `loadReserved` because the functionality could also be used for other purposes such as speculative prefetching. The term *exclusive* also provides more clarity in our point of view.

   The `SC` instruction (Lines 7-16) checks whether a certain memory location is still exclusively owned and reserved using another builtin method call `isExclusive`. If yes, the store will be performed. Similarly to the version presented in Listing 4.6, the instruction must lock the memory location first in order to prevent a race condition between the address checking (Line 10) and the actual store (Line 11).

All these presented options are not mutually exclusive and in fact, may complement each other. The reason is that ISAs themselves often define mandatory parts but leave some aspects to individual implementations. For instance, take the POWER ISA: A processor can hold at most one reservation at any time and the size of a reservation is 2^n but n depends on the MiA [Ope21, p. 1049]. Hence, we argue that VADL needs to represent LR/SC on multiple levels. For instance, options 2 and 3 could be used in conjunction, where Listing 4.8 represents the general LR/SC behavior and the `logic` element shown in Listing 4.7 defines MiA-specific aspects.

Within the context of this thesis, we decided to implement option 3. While 1 can already be used since it relies purely on previously existent functionality, we believe that LR/SC are important primitives that should receive first-class support in the Vienna Architecture Description Language (VADL). Finally, we do not further consider option 2 in this thesis, however, it can be added in the future as an extension. Hence, we describe the semantics of `loadExclusive` and `isExclusive` in more detail:

VADL Specification 3 (Semantics of loadExclusive)
The expression `MEM( addr ).loadExclusive` requests exclusive memory access to location `addr` in addition to loading the value stored at `addr`. In addition, the address `addr` will be marked as reserved.

When talking about exclusivity, this means that the cache controller (of the simulator or generated hardware) requests read-only or writable access, depending on the underlying cache coherence protocol.

VADL Specification 4 (Semantics of isExclusive)
The expression `MEM( addr ).isExclusive` returns true if and only if there exists a correlating `loadExclusive`, called with the exact same address `addr`, that happened before and no other globally visible memory write on the same location or cache line occurred in-between. Otherwise, the expression returns false.

Note that `isExclusive` specifically returns false, if there exists no correlating `loadExclusive` call, even though the cache line might coincidentally be owned exclusively. In addition, we impose the requirement that addresses must match. The reason is that LR/SC pairs are often used for implementing RMW operations (e.g. CAS) and hence, usually operate on the exact same address. To our knowledge, there exists no practical algorithm that specifically takes advantage of the granularity of a reserved region (e.g. via an LR/SC pair where the LR loads from address a but the correlating SC stores at the neighboring address $a + 1$).

4.1.2 Integration into VADL Frontend

This section begins with the integration of the *lock statement*. The grammar is shown in Listing 4.9.

Recall the semantics of the *lock statement* from Specification 2. A lock is acquired before and released after the execution of the `inner` statement. This behavior is quite similar to a pattern known as Resource acquisition is initialization (RAII), commonly used in

```
instruction set architecture ISA = {
    instruction LR : FormatB = {
        let addr = X( arg ) in
        X( res ) := MEM( addr ).loadExclusive
    }

    instruction SC : FormatB = {
        let addr = X( arg ) in
        lock MEM( addr ) in
        if MEM( addr ).isExclusive then {
            MEM( addr ) := X( res )
            X(0) := 0
        } else {
            X(0) := 1
        }
    }
}
```

Listing 4.8: Example implementation for LR/SC instructions using builtin method calls.

```
LockStatement
: LOCK
  expression=ConcreteCallExpression
  IN
  statement=Statement
;
```

Listing 4.9: Grammar definition of the *lock statement*.

C++ and Rust. VADL does not support such constructs. Thus, we introduce a new `AstRAIIPass` into the VADL frontend. As the name suggests, the pass operates on the AST because we need to precisely know where the inner statement begins and ends. This information is not available in the VIR anymore. The pass checks for an occurrence of a *lock statement* and resolves it to a *block statement*. The block begins with a *direct call expression* to **`lock`**`(Memory Reference, Address, Size)`, then follows with the inner part of the lock statement and ends with a call to `unlock()`. The latter does not have any arguments because resources are freed in the reverse order of how they were requested. The `loadExclusive` and `isExclusive` methods are also resolved in the same pass, even though they technically do not exhibit a RAII pattern. This concludes the lowering of our atomic primitives to the VIR.

The generated VIR can be translated directly into an ISS. However, the CAS needs to take additional considerations into account: Recall that the CAS shares a lot of code with the hardware generator. This means that unlike the ISS which sequentially executes every

VIR instruction, hardware tries to generally execute more parts in parallel. Needless to say, data and control flow dependencies need to be taken into account. Recall that a lock must be acquired before the inner statement and released immediately afterwards. Thus, we need to ensure that the CAS generator respects this order of operations. The VADL frontend includes a special data structure called instruction progress graph (IPG). This graph is used to arrange VIR operations, such as memory reads/writes or executing an instruction semantics, and generate the pipeline according to the MiA while executing as many VIR operations as possible within a single clock cycle. Each node in the graph represents a single VIR instruction, while every edge denotes a dependency. There exist three types of dependencies, `OPERAND`, `RESULT` and `REQUIREMENT`. The latter is particularly important for our use case: Locking or unlocking a memory location invokes the builtin **`lock`** or `unlock` process, accomplished via an `InstInstruction`. We define every read/write to a memory instance m as r_m and w_m respectively and an `isExclusive` call to memory instance m as e_m. Let I_{ml} be the set of VIR instructions of types r_m, w_m or e_m occurring between a **`lock`** l of memory instance m and `unlock` u in program order, hence $I_{ml} = \{\, i \mid \forall i \in [l, \ldots, u], t(i) \in \{r_m, w_m, e_m\} \,\}$ where $t(i)$ denotes the type of VIR instruction i. Next, we add a `REQUIREMENT` edge from every $i \in I_{ml}$ to the corresponding **`lock`** l as well as the same type of edge from the `unlock` u to every $i \in I_{ml}$. These adjustments ensure that the semantics of VADL specification 2 are maintained.

Finally, we needed to adapt the `VirControlFlowEliminationPass` which is responsible for eliminating control flow by applying a busy code motion. The algorithm moves every instruction within a condition to the initial basic block. This might introduce additional computations although the involved branch will not actually be executed. In typical compiler optimizations, this is undesirable, however, hardware can execute these instructions in parallel, assuming we disregard the resulting size of the chip. Thus, the computational overhead can be neglected in theory. However, these additional computations matter once they have a side effect that should only apply in certain circumstances. The pass handles this situation by replacing side effect instructions, including reads and writes, with a conditional version that only applies the side effect if the condition is satisfied. This means we needed to adapt the code motion to additionally consider **`lock`**s and `unlock`s:

- A **`lock`** and `unlock` within a conditional branch may only be applied if the condition is true. This means we need to change an `InstInstruction` to a `ConditionalInstInstruction`. The latter did not exist in the VIR before and was added to the intermediate language for this purpose.
- The code motion must still respect the semantics of the *lock statement*, as defined by Specification 2. Hence, if a(n) (un-)conditional read and write instruction or `isExclusive` call from between a **`lock`** and `unlock` gets moved before the **`lock`** (in order to eliminate the basic block of a branch), the **`lock`** must be moved as well.

Finally, we need to ensure that these primitives are provided by the memory subsystem. For the cache simulation, we refer the reader to Section 4.3 for more in-depth explanations.

```
class Memory {
    std::optional<Address> m_linkedAddress;

    Data read(Address);
    void write(Address, Data);
    template<size_t N>
    void lock(Address address);
    void unlock();

    void register_link(Address address) {
        m_linkedAddress = address;
    }

    bool is_linked(Address address) {
        return m_linkedAddress.has_value()
            && *m_linkedAddress == address;
    }

    template<size_t N>
    void clear_link(Address address) {
        if(m_linkedAddress.has_value()
            && (address <= *m_linkedAddress
                && m_linkedAddress < (address + N))) {
            m_linkedAddress = std::nullopt;
        }
    }
};
```

Listing 4.10: Memory interface

In a cacheless system, it is the responsibility of the memory to implement these primitives. The `Memory` class in Listing 4.10 is responsible for simulating memory using an anonymous memory map. It provides a `read` and `write` function where `Data` corresponds to the type of a single unit (e.g. in a byte-addressable system, `Data` corresponds to a `uint8_t`), while `Address` maps to the type of an address (e.g. `uint32_t` for a system with 32-bit memory). The `lock`/`unlock`, `register_link` and `is_linked` functions correspond to the VADL primitives *lock statement*, `loadExclusive` and `isExclusive` respectively. The `clear_link` function acts as a helper function to remove a link in case of a write occurring in-between. The template parameter `N` represents the size of a write in order to check whether the access overlaps with the reserved address. Both `lock` and `unlock` are no-ops because the ISS and CAS are limited to single-threaded simulation and thus, the memory simulation was not designed to be thread-safe. Since multicore simulation is not in the scope of this thesis, adding a proper locking mechanism introduces needless overhead and thus, is not considered here.

In order to support the `loadExclusive` and `isExclusive` primitives, the simulator needs to remember the reserved address in the `m_linkedAddress` variable. Observe that this

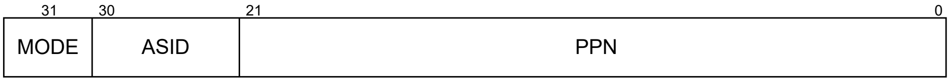

Figure 4.1: Structure of the *satp* register for RV32. [RIS21, p. 73]

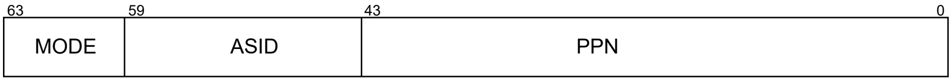

Figure 4.2: Structure of the *satp* register for RV64. [RIS21, p. 73]

means that only the address of the most recent `loadExclusive` will be reserved. However, Specification 4 only enforces that some previous `loadExclusive` must have happened before but not necessarily the most recent one. We looked at some popular ISAs and found that most of them put a lot of constraints on what assumptions software developers can make about reservations. For instance, both Power and RISC-V specifically mention that each processor can hold at most one reserved address at a time [Ope21, p. 1049] [RIS19, p. 49]. ARMv8 defines a conditional store to an address different from the preceding LR as 'constrained unpredictable' [Arm21, p. B2-313]. Thus, hardware manufacturers have engineered more general solutions, such as hardware transactional memory (HTM). However, some of them, most notably Intel [int] and IBM [Ope21, p. 1363], have removed this functionality from modern processors. Hence, we argue that having more than one reservation at the same time is unusual or at least impractical to be implemented in hardware.

4.2 Address Translation

This section presents how we extended VADL to support translation from virtual to physical addresses. The goal was to implement the 'Page-based 32-bit Virtual-Memory System' (Sv32) from RISC-V [RIS21, p. 79] for our RV32 VADL specification. Nevertheless, we also considered other translation schemes, such as from Armv8 [Arm21] and AMD64 [Adv24a]. They generally share a lot of common aspects such as table-walking or memory protection by differentiating between read, write and execute accesses. Our initial idea was to keep memory translation simple for VADL developers. For instance, the structure of a page-table entry could be described via a format definition and annotations could be used to mark important aspects of memory translation, such as what bit represents whether an entry is valid. However, the detailed implementations of these translation schemes differ in subtle ways that we decided that VADL should enable full flexibility in order to accurately model these translation processes. The language already provides so-called processes to write arbitrary code (apart from the usual VADL limitations that no loops or recursive calls are allowed). Thus, we want to present how we implemented the Sv32 memory translation using VADL processes.

First, consider the components of the Sv32 system. The *satp* register defines the

```
1 instruction set architecture ISA = {
2     [ translate VMEM ]
3     memory MEM : Address -> Data
4 
5     process VMEM(addr: Address, type: VADL::AccessType)
6         -> (out: Address) = {
7             out := addr + 0x8000
8     }
9 }
```

Listing 4.11: Toy example for applying a memory translation scheme to an ISA specification. Additionally showcases the use of the builtin `AccessType` enumeration.

configuration of the address translation process. Figures 4.1 and 4.2 show the structure for RV32 and RV64 respectively. The *Mode* field determines what translation scheme should be used. As of this writing, RISC-V specifies four different variants Bare, Sv39, Sv48 and Sv57 for RV64 as well as Bare and Sv32 for RV32 [RIS21, p. 75]. For instance, zero corresponds to no translation and protection, while a value of one enables the Sv32 virtual memory system in the case of RV32. The *address space identifier (ASID)* determines the current address space identifier in order to avoid flushing the translation lookaside buffer (TLB) in case of context switches. Finally, the *physical page number (PPN)* field encodes the address of the current page table. Since page tables have a fixed size and alignment of four KiB, it is sufficient to store the 22 most significant bits. The translation procedure translates a 32-bits virtual to a 34-bits physical address. Their structures as well as the translation process itself are presented in Section 2.5. We implemented the algorithm in VADL which we present in Appendix B.

Listing 4.11 presents a minimum example where the translation function `VMEM` adds a fixed offset to the input address `addr` (Line 7). The output value `out` is then passed on to the memory object `MEM`. The *translation* annotation on the memory definition enables address translation. Its second property defines the name of the responsible translation process, which corresponds to `VMEM` in this case (Line 2). Every read or write access within an instruction definition invokes said procedure before it accesses the actual memory. We added a new pass called `VirMemoryTranslationReplacementPass` to the frontend, which is responsible for injecting these process calls. In addition, we adapted the `VirSimulatorBehaviorPass` in order to inject a call to the translation process before the next instruction is fetched from memory. Needless to say, the `VMEM` process must be able to differentiate between these types of accesses. Note the second argument of `VMEM` which is of type `VADL::AccessType`. It is a builtin enumeration that contains the three variants, `Read`, `Write` and `Execute`. The VADL type checker ensures the process is called with the correct variant, representing the underlying access. While VADL already contains several builtin definitions, builtin enumerations are a novelty which required some adjustments to the frontend. Currently, only VADL processes can be used as an argument to the *translation* annotation and its signature must match a fixed structure: The first argument as well as the output value must be of the same type as the input argument to the corresponding memory object. The second argument must be the builtin `VADL::AccessType`.

The Sv32 implementation, shown in Listing B.3, provides a more thorough example for a memory translation scheme. We named the translation process `VMEM` as well, which acts as the entrypoint for the translation by checking the mode of the *satp* register. If the mode is zero, the virtual address (first argument) is directly mapped to the physical address. Otherwise, the `SV32` translation procedure will be invoked. Note that we limit physical addresses to 32-bits for simplicity purposes and due to signature limitations mentioned above.

4.3 Cache

Another aspect of this thesis is to extend the MiA section of VADL to support caches and arbitrary memory hierarchies. We begin with the language side and then consider the simulation of the cache hierarchy.

4.3.1 Language Design

```
[ evict roundRobin ] // Replacement policy = round robin
[ prot snooping_wb_mesi ] // Cache Protocol
[ entries = 1024 ] // index = 1024 entries
[ blocks = 64 ] // 64 blocks per cache line
[ nSet = 2 ] // 2-way set-associative
[ attachedTo L2 ]
[ dataCache ]
[ instructionCache ]
cache L1 : Bits<32> -> Bits<8>
```

Listing 4.12: Cache configuration in VADL.

The VADL language already provides a basic syntax for defining caches. Listing 4.12 shows an example definition of an L1 cache. Readers familiar with VADL might notice that the syntax is almost identical to a memory definition. A cache is basically a function that receives an address as an input and yields some data as an output. As described in Section 2.2, caches exist in a variety of forms and configurations, most notably how the cache controller keeps caches consistent. Cache model languages, such as Ruby, try to consider all of these aspects. However, this also implies a high level of complexity, even for simple cache coherence protocols. Thus, we decided to prefer configurability and simplicity over flexibility, inspired by Multi2Sim concise configuration options [ea, p. 114]. We believe this allows VADL developers to easily add caches to their MiA without requiring deep knowledge on this topic. All configurable attributes are shown in Listing 4.12, which can be set via annotations.

`entries` on Line 3 refers to the total number of cache lines in the cache. `nSet` on Line 5 defines the associativity of the cache. For instance, a value of two means that each set contains two entries. In case this option is not provided, VADL defaults to a value of one which corresponds to a direct-mapped cache. If `entries` = `nSet`, we have a fully associative cache. Note the correlation between these two options. Both determine the number of sets in the cache. Using our example in Listing 4.12, since the cache has a total of 1024 cache lines and each set has two entries, there exist 512 sets in the cache. `blocks` determines the number of blocks of a single cache line. A block corresponds to a single addressable unit. In the case of our example, a 32-bit address returns an 8-bit value. Thus, a single block corresponds to 8 bits and 64 blocks denote a cache line size of 64 bytes. All three mentioned options directly impact the sizes of the index and tag. Continuing with our example, we previously determined that our cache has 512 entries and thus, our index requires $\log_2(512) = 9$ bits. One cache line contains 64 blocks and thus, we have an offset size of $\log_2(64) = 6$ bits. The remaining most significant bits construct the tag. Thus, the tag has size $32 - \log_2(512) - \log_2(64) = 17$ bits.

Next, observe the `evict` annotation on Line 1. This setting allows VADL developers to select a replacement policy in case a set does not have an empty entry available. We decided to provide three policies, random, round-robin and least-recently used (LRU). Next, Line 2 defines the cache coherence protocol used by the controller. We currently support six options: `snooping_wb_msi`, `snooping_wb_mesi`, `snooping_wb_mosi`, `directory_wb_msi`, `directory_wb_mesi` and `directory_wb_mosi`. Each value refers to either a snooping- and directory-based version of an **MSI**, **MESI** or **MOSI** protocol (see Section 2.2.2). `wb` is an abbreviation for *write back* (see Section 2.2). Finally, we need to specify how cache misses should be handled. This is achieved via the `attachedTo` annotation where the VADL developer may provide the identifier of a fallback source. This can either be another cache or memory which allows to construct arbitrary cache/memory hierarchies. This covers the basic configuration of a cache in VADL. A cache may be used similarly to a memory instance. For instance, the statement `let x = L1<4>(0x1F) in` reads four bytes at address 0x1F from our example cache in Listing 4.12 and `L1(0x1F) := 0x42` stores the value 0x42 at the same location.

Now recall that VADL strictly separates the ISA from the MiA definition. Since caches are a microarchitectureal aspect, they cannot be directly used in the ISA. In order to redirect memory accesses within instruction definitions, we decided to handle this use case via the `dataCache` annotation as seen in Listing 4.12: There can only be at most one data cache in the MiA section. The VADL frontend will then redirect all memory usages within instruction definitions to the cache marked as entrypoint. Similarly, we added another annotation called `instructionCache` to instruct the VADL frontend to fetch instructions from the marked cache instead of from memory directly. Note that unlike in our example, data and instruction cache can be separate definitions which is common in most MiAs.

4.3.2 Implementation

Before talking about the actual implementation, we want to introduce our terminology for certain components in the memory hierarchy:

- A **cache controller** or just **cache** manages one instance of a cache, including its data and states. It is connected to one specific interconnect. If a cache wants to fetch data, it needs to consult the interconnect. The response may originate from another cache or the memory/last-level cache (LLC) controller in case no other cache has the currently requested data. The exact procedure depends on the cache coherence protocol.

- An **interconnect** represents a network of arbitrary topology.

- A **memory** or **LLC controller** manages the global state of data in memory and additionally, represents the gateway to memory or the upper level. Despite its name, the memory/LLC controller can be connected to another cache too. For instance, in a system with a L1 and L2 cache, the L1 memory controller is also a L2 cache controller.

Next, we describe our desired functional as well as non-functional requirements. The goal is to be able to simulate the following cache coherence protocols:

- Snooping MSI, MESI and MOSI protocols with write-back support

- Directory MSI, MESI and MOSI protocols with write-back support

While the difference might seem subtle, snooping- and directory-based protocols differ substantially. Recall from Section 2.2 that in a snooping-based protocol, all entities observe all coherence messages via a bus. On the contrary, directory-based protocols send messages directly to the recipient. It should be noted that some implementations do not require Point-to-Point connections [NSH$^+$20, p. 180]. Nonetheless, directory protocols tend to be more explicit. For instance, the protocols presented in [NSH$^+$20] send explicit invalidation messages in case a cache requests mutable access, while other entities have shared copies in their caches. In a snooping-based protocol, this invalidation occurs implicitly because all messages and requests are observed in a global order which does not apply to a point-to-point interconnect. Hence, we define the following non-functional characteristics:

- **Extendability**: As described in Section 2.2, the vast amount of design possibilities of caches make it difficult to design a simulator that allows to simulate arbitrary cache configurations. Nonetheless, we strive for a software architecture that allows adaptations of the simulator. This particularly applies to implementing new cache-coherence protocols.

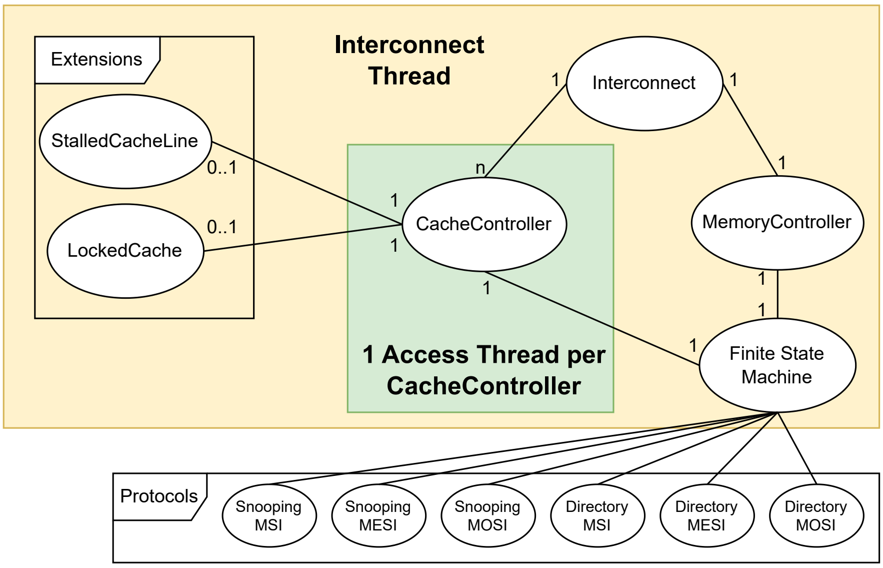

Figure 4.3: Overview of the cache simulator architecture. The yellow rectangle represents the interconnect which is responsible for distributing messages by invoking the message handler of each *CacheController* as well as *MemoryController* (see Algorithm 4.3). The consequence of a message is determined by the finite state-machine (FSM), which corresponds to one of the designated coherence protocols. Note how the *CacheController* is also surrounded by a green rectangle. This corresponds to the thread reading/writing from/to the cache. Only one thread may access one instance of the *CacheController* but the amount of caches are unbounded.

- **Performance**: Cycle accurate simulation is already quite slow compared to an ISS. Thus, we want to keep the impact of the cache simulation low.
- **Thread-safe**: While VADL does not yet support designing multicore architectures and hence, neither the ISS nor CAS simulators are multithreaded, the cache simulator must support a multithreaded environment to be future-proof.

In order to fulfill the second point, only low-level compiled languages, such as Rust, C(++) and Go can be considered. Since the ISS and CAS are already written in C++, we decided to choose the same language in order to keep build system and maintainability simple. Furthermore, C++ provides templates which are a construct to parameterize classes and functions at compile-time. Since the CAS is individually generated for each VADL specification, the VADL frontend is able to configure the cache simulator at compile-time. Hence, we decided to use templates as a configuration mechanism to potentially allow the C++ compiler to even more aggressively optimize the code.

In order to support a flexible cache simulator, we designed an architecture that separates general cache management aspects from the coherence protocol itself. A simplified version of the architecture is shown in Figure 4.3. We separate the system into cache-, interconnect- and memory-related parts. Recall that the former does not necessarily refer to actual memory but to the gatekeeper from the cache to the next level, which could be memory but also another independent cache. The actual state transitions are handled by the individual coherence protocols, implemented as FSMs. In order for the cache controller to understand the semantics of a given state, the coherence protocols must provide certain primitive functions:

- `is_readable`: Whether the given state allows read-access.
- `is_writable`: Whether the given state allows write-access. Note that `is_writable` $\implies$ `is_readable`.
- `is_wait_for_data`: Whether the given state denotes a transitional state waiting for the data to arrive.
- `requires_notification`: Whether the memory controller must be notified about eviction. Note that this does not imply whether the data needs to be sent back as well. For instance, a MESI protocol with an *Exclusive* state must explicitly notify the memory controller about eviction. The cache might have silently upgraded the line to *Modified* and hence, the controller cannot distinguish between these two states.
- `is_promotable`: Whether the given cache can be upgraded from read-only to read-write without explicitly sending a request on the interconnect.
- `request_read`: Retrieve the follow-up state to gain read access.
- `request_write`: Retrieve the follow-up state to gain write access.
- `request_evict`: Retrieve the follow-up state to evict a cache line.

The first two points are used for handling access to the cache line. For instance, when a thread wants to write certain data to some location, the controller can check with `is_writable` whether the given state allows writing without actually knowing about the underlying state. `requires_notification` tells the controller whether it must send the data back to the upper level cache or memory before evicting the line. As explained in Section 2.2.2, the **MESI** protocol includes an *Exclusive* state, which is read-only but allows upgrading to a writable state without consulting the interconnect first. The `is_promotable` method enables this specific use case.

Next, we present the basic procedure for reading from the cache, presented in Algorithm 4.1. Based on the index i and the tag t extracted from address a, the cache line is retrieved on Line 4. The tag of l does not necessarily match t but l is guaranteed to correspond to the entry at index i (direct-mapped) or one of the entries in the set located

Algorithm 4.1: Basic algorithm for read access.

```
    input  : Address $a$
    output : Result $r$
 1  $i \leftarrow \text{index}(a)$;
 2  $t \leftarrow \text{tag}(a)$;
 3  $o \leftarrow \text{offset}(a)$;
 4  $l \leftarrow get\_cache\_line(i, t)$;
 5  if $l.invalid \vee l.tag \neq t$ then
 6      if $l.state.requires\_notification()$ then
 7          interconnect.push($Evict(l, a)$);        // Notify about eviction
 8          $l$.state ← $l$.state.request_evict();
 9          $l$.wait_until($l$.state.invalid_state());
10      end
11      interconnect.push($Read(a)$) ;        // Request read access for $a$
12      $l$.tag ← t;
13      $l$.state ← $l$.state.request_read();
14      $l$.wait_until($l$.state.is_readable());
15  end
16  $r \leftarrow l[\text{o}]$;
```

at i (set-associative). If the cache is set-associative and does not contain a line matching the tag t, then some line in set i is selected by the replacement policy. Nonetheless, the simulator needs to check whether the tag matches on Line 5. If yes, it can commence reading from the offset o within the cache line and return the value on Line 16. Otherwise, the system needs to evict the line and ask for read-only access on the interconnect (Line 11-14). In addition, it might be necessary to notify the memory controller or other entities about the eviction. Hence, the cache sends a corresponding message, which might contain the current cache line, via the interconnect before requesting new data (Line 6-9). For instance, this applies if the currently contained memory region is in the *Modified* state. Observe that the system does not check on Line 5 whether the cache line is actually readable. Recall that `is_writable` $\implies$ `is_readable`. In other words, our algorithm assumes that there exists no write-only state. Thus, since it checks whether the cache line is valid, we can imply readability on Line 16.

The algorithm for writing to the cache is quite similar, as seen in Algorithm 4.2. In addition to checking whether the cache line is valid and tags match, the simulator needs to ensure that the cache line permits writing. Using the `is_promotable` primitive, the controller can check whether it may upgrade the cache line for free (Lines 6-7). Otherwise, it checks whether the line needs to be evicted (Line 9). Unlike in the read-case, the tags might match but the cache does not have write permissions. Consider the *Owned* state of the **MOSI** protocol. This state is read-only but dirty and hence, requires its data to be written back when evicted. However, if our algorithm wants to upgrade a cache

Algorithm 4.2: Basic algorithm for write access.

```
input  : Address $a$, Data $d$
$i \leftarrow$ index($a$);
$t \leftarrow$ tag($a$);
$o \leftarrow$ offset($a$);
$l \leftarrow$ get_cache_line($i, t$);
if $l.invalid \lor l.tag \neq t \lor \neg l.state.is\_writable()$ then
    if $l.tag = t \land l.state.is\_promotable()$ then
        $l$.state $\leftarrow$ $l$.state.request_write();
    else
        if $l.tag \neq t \land l.state.requires\_notification()$ then
            interconnect.push($Evict(l, a)$);      // Notify about eviction
            $l$.state $\leftarrow$ $l$.state.request_evict();
            $l$.wait_until($l$.state.invalid_state());
        end
        interconnect.push($Write(a)$) ;    // Request write access for $a$
        $l$.tag $\leftarrow$ t;
        $l$.state $\leftarrow$ $l$.state.request_write();
        $l$.wait_until($l$.state.is_writable());
    end
end
$l$[o] $\leftarrow$ $d$;
```

line from *Owned* to *Modified*, eviction is not necessary and would otherwise diminish the advantages of this protocol.

Both algorithms show how we separate cache management from the coherence protocol. The primitives described previously allow the controller to inquire about the permissions of the corresponding state. Furthermore, the `request_evict`, `request_read` and `request_write` functions return the corresponding successor state based on the current one. As a simple example, consider the **MSI** protocol. If the cache line was invalid, `request_write` will return the transitional *InvalidToModified* state. In the case of both tags being equal but the cache line is read-only (*Shared* state), the same function will return the *SharedToModified* state.

In order to simulate the parallel effects of a cache coherence protocol, the simulator spawns a thread per interconnect to handle message distribution across all caches and the memory controller. In an earlier design of the cache simulator, each cache, memory and interconnect had a dedicated message handling thread. However, initial tests have revealed that the synchronization overhead heavily impacted the throughput of the simulation, which stands in contrast to our goal of a high-performance cache simulator. In addition, debugging became significantly more difficult.

In the current architecture, as seen in Figure 4.3, each cache controller may be accessed by

Algorithm 4.3: Basic algorithm handling incoming messages on the interconnect.

```
   input  : Message $m$
1  Procedure handle_message():
2      $i \leftarrow$ index($m$.address);
3      $t \leftarrow$ tag($m$.address);
4      $l \leftarrow$ get_cache_line($i, t$);
5      if $l.exists$ then
6          $l$.transition($m$);
7          $l$.notify();
8      end
9      foreach $s \in M_{rs}$ do
10         handle_message($s$);
11     end
12     $M_{rs} \leftarrow \{\}$;
13 end
```

a single thread. Hence, they are not thread-safe but the number of caches is unbounded. When the thread requests read or write access via the interconnect, the dedicated interconnect thread processes these messages by invoking the message handler of each cache and memory controller, seen in Algorithm 4.3. The `wait_until` function, seen in Algorithms 4.1 and 4.2, puts the initiating thread to sleep while it awaits a state change. How messages are distributed depends on the network topology. In a bus, when the cache sends a message through the interconnect, the message must be processed by all responsible entities (e.g. the cache(s) and/or memory which currently own or have a copy of the requested cache line). The response is then sent back via the bus and must be processed by every entity. The Point-to-Point interconnect can send messages directly to the corresponding entity. Hence, messages have a destination id attached to them. The simple algorithm shown in 4.3 presents the message processing side of a single cache entity. We observe that the algorithm must check whether its cache contains the affected address on Line 5. This is particularly important for bus-based coherence protocols as every entity observes all messages and therefore, they must check whether they are affected by the given message m. For directory-based protocols, it depends on the organization of the directory. As mentioned, some directories group set of cores when tracking sharer and owner information. In fact, directories can be organized without any backing store, called Null Directory, at all [NSH+20, p. 176]. In case the cache contains the corresponding cache line, the message handling is delegated to the underlying cache coherence protocol using the `transition` function. Again, we show how the simulator clearly separates general cache management responsibilities from the protocol. The `transition` function might change the state of the cache line, update its data or even invalidate it (e.g. when another thread has requested write-access). After the transition, the cache notifies other threads about a change on Line 7. This `notify` function directly correlates with the `wait_until` used in 4.1 and 4.2. When `wait_until` is called, the thread awaits a change in the cache line state. An invocation of `notify` triggers the waiting thread to check for the inner condition (e.g. whether the cache line is now writable).

Next, the message handling procedure may process stalled messages (Lines 9-11). We required this functionality for our implementation of some directory-based protocols as they sometimes delay certain messages. Last but not least, we want to highlight the extension mechanism of the simulator. In order to keep the logic of the cache simulator simple as well as remaining open to future changes, extensions can be dynamically added to the controller itself. These extensions must provide certain callback functions which are invoked at several points of the controller. We utilize this mechanism for two features required for VADL:

- **StalledCacheLine**
- **LockedCache**

Both are quite similar but are internally designed for different use cases: The **StalledCacheLine** is an extension that prevents a newly fetched cache line from being evicted until the core has accessed its data at least once. The idea here is that since the message handler runs on a separate thread, the situation might occur that a newly acquired cache line is evicted before the other thread had a chance to actually execute the cache access, potentially preventing forward progression of the entire system. The **LockedCache** similarly prevents a cache line from being evicted. However, the former is optimized for implicitly locking a single cache line, while the latter can support arbitrarily many lines. In addition, a line must be explicitly locked by calling the extension's `lock` function. This feature is necessary in order to support the *lock statement* introduced in Section 4.1.

Finally, we want to briefly showcase the cache coherence protocols we decided to implement:

- SNOOPING_WB_MSI
- SNOOPING_WB_MESI
- SNOOPING_WB_MOSI
- DIRECTORY_WB_MSI
- DIRECTORY_WB_MESI
- DIRECTORY_WB_MOSI

We refer to Section 2.2 for an explanation of all the involved states. The snooping-based protocols communicate via a single shared bus. Furthermore, these protocols require atomic transactions. A transaction is atomic if a request completes (receives a response) before any following request to the same memory region where a region corresponds to the size of a cache line [NSH$^+$20, p. 114]. Otherwise, the coherence protocols would require additional transient states in order to handle the case of interleaved requests. All directory-based protocols use a point-to-point interconnect. All protocols are write-back schemes, hence, the reason they contain **WB** in their names.

4.3.3 Integration into VADL Frontend

```
template<unsigned N, class T>
void read(T& data, uint32_t address);

template<unsigned N, class T>
void write(T data, uint32_t address, size_t mask = ~0U);
```

Listing 4.13: Functions provided by the current `Memory` class.

The cache simulator was designed as a standalone simulator to be used in different environments. The memory interface of the CAS supports the function shown in Listing 4.13. The `mask` parameter provides a way to only write certain bytes of the data. Each bit in the mask corresponds to a byte in the input. For instance, the mask `0b101` writes only the first and third least significant byte but ignores the second least one. The template parameter N defines the number of bytes accessed by the read or write. The memory subsystem supports arbitrary sizes when accessing memory. For a cache, this is a bit more tricky because an arbitrarily sized read or write to a cache might span multiple cache lines. Arbitrary sizes within a single cache line are supported directly by the simulator. What if a memory access crosses cache line boundaries? ARMv8 describes read or write as single-copy atomic, if and only the address is aligned to the size of the access. Informally, single-copy atomicity is defined that if two threads execute two stores S_1 and S_2, the value committed to memory must be either but never a mixture between them. Similarly a load L_1 and S_1 occurring at the same time, L_1 observes either S_1 or the previous value stored at the same location but never a mixture [Arm21, p. B2-222]. RISC-V [RIS19, p. 84] and amd64 [Adv24a, p. 195] have similar behavior. Some ISAs permit MiAs to not support misaligned memory accesses at all [RIS19, p. 25]. If misaligned loads and stores are allowed, the result is usually undefined in multi-threaded environments [Adv24a, p. 195] because a single load might need to be split up into two or more memory reads in case the request crosses cache line boundaries. Hence, we decided to pose no restrictions on what value the cache may return in case of crossing a cache line boundary in the simulator.

VADL Specification 5
A read or write to a cache, which spans across more than one cache line, is implementation-defined behavior, unless all affected cache lines are locked.

If an ISA does not support misaligned memory accesses, then the cache can be designed

```
 1 template<size_t N, class T>
 2 void write(T data, uint32_t address, const size_t mask = ~0U) {
 3     clear_exclusive<N>(address);
 4
 5     if(cache.isWithinCacheLine<N>(address)) {
 6         cache.apply_mutable(address, [&](auto &line, auto offset) {
 7             for(size_t i = 0; i < N; i++) {
 8                 if(((0x1 << i) & mask) != 0) {
 9                     line.store(offset + i,
10                         static_cast<uint8_t>(data >> (8 * i)));
11                 }
12             }
13         });
14     } else {
15         for(uint32_t i = 0; i < N; i++) {
16             if(((0x1 << i) & mask) != 0) {
17                 cache.store(address + i,
18                     static_cast<uint8_t>(data >> (8 * i)));
19             }
20         }
21     }
22 }
```

Listing 4.14: Simplified version of the `write` function in the cache.

in a way that all regular loads and stores always remain within a cache line. The size of a line must simply be greater or equal to the largest possible memory access. For instance, if the machine supports 8-byte loads, then the cache line size must be a multiple of eight. If atomicity for misaligned addresses or larger access sizes are required (e.g. due to vector or floating point instructions), the VADL developer is responsible for locking the affected memory region using a *lock statement* beforehand.

Listing 4.14 shows a simplified version of the `write` method from the memory interface presented in Listing 4.13. On Line 3, the simulator clears the link to an address if the link is within the range $[a, a + N)$ where a denotes the input address `address`. Similarly to the implementation of LR/SC in the memory class (see Listing 4.10), we use an `std::optional` as we decided to limit our simulation to having at most one reservation per core. Lines 5-21 conduct the actual write to the cache. The `apply_mutable` function is provided by the simulator that requests read and write access to the given address and calls the lambda with exclusive mutable access to the cache line. In the lambda call, we check for each byte whether it should be stored. The `offset` parameter provides the offset within the cache line of the given address. Furthermore, we need to consider the case where writes cross cache line boundaries (checked on Line 5). If yes, we need to store each byte individually on Lines 15-20. While this might look similar to the case where a write remains within a single cache line, the bytes written may not appear atomically to other observers. The reason is that in-between each iteration of the loop, another cache may

```
1  template<size_t N, class T>
2  void read( T &data, uint32_t address ) {
3      if(cache.isWithinCacheLine<N>(address)) {
4          result = is_linked(address) ?
5              cache.load<T, true>( address ) :
6              cache.load<T>( address );
7      } else {
8          for(auto i = 0U; i < N; i++) {
9              result |= cache.load<uint8_t>(address + i) << (8 * i);
10         }
11     }
12 }
```

Listing 4.15: Simplified version of the `read` function in the cache.

receive access to the affected cache line and modify its data in-between before the cache line is re-requested on the next iteration. In the other case, the `apply_mutable` function ensures that the cache line is stalled until the given function has finished execution. Hence, the effect of the function is atomic to other entities.

The `read` function is presented in Listing 4.15. First, we check on Line 3 whether loading N bytes from the input addresses crosses a cache line boundary. If yes, the simulator reads each byte individually. Otherwise, we can load the entire value in one step from the simulator. Note that we also check whether an address was registered via a `loadExclusive`. If yes, we specifically request write-access (by passing **`true`** as a second template parameter to the `load` function) in order to obtain the cache line in an exclusive state. We observe that we do not check for a link in case of crossing cache line boundaries. Crossing cache line boundaries usually stem from misaligned memory accesses which is not supported by most ISAs for impracticability reasons [RIS19, p. 49].

In Listing 4.16 we show the `lock` function of the CAS as well as its counterpart, the `unlock` function, in Listing 4.17. Since addresses are unlocked in the reverse order, we can simply put the address on a stack, seen on Line 1 in Listing 4.16. However, we observe that we actually store a set of addresses instead of individual ones. The template parameter N denotes the number of bytes to be locked. This means the memory region $[\text{address}, \text{address} + N)$ will be locked and this range might affect more than one cache line. Therefore, the simulator needs to store all addresses within the range corresponding to all relevant cache lines. The conditional statement on Lines 8 and 9 effectively ensures that we skip duplicates that would otherwise represent the same cache line. In the `unlock` function in Listing 4.17, we take the topmost set of addresses on the stack and unlock the corresponding cache lines (Lines 6-8).

Both snippets additionally showcase the use of extensions. Since we wanted to separate the locking mechanism from the general cache management, the **LockedCache** extension is responsible for this feature. The `extension` method enables access to its underlying

```
1  std::stack<std::vector<uint32_t>> globalLockedAddresses;
2  
3  template<size_t N>
4  void lock(uint32_t address) {
5      std::vector<uint32_t> lockedAddresses;
6      for(auto i = 0; i < N; i++) {
7          auto aligned = align_address<CACHE_LINE_SIZE>(address + i);
8          if(lockedAddresses.empty()
9              || lockedAddresses.back() != aligned) {
10             lockedAddresses.push_back(aligned);
11             cache.extension().inner<1>().lock(aligned);
12             cache.request_writable(aligned);
13         }
14     }
15     globalLockedAddresses.emplace(lockedAddresses);
16 }
```

Listing 4.16: Simplified version of the `lock` function in the cache.

```
1  void unlock() {
2      if(globalLockedAddresses.empty()) {
3          throw std::runtime_error("No address has been locked");
4      }
5      auto addresses = globalLockedAddresses.top();
6      for(auto a : addresses) {
7          cache.extension().inner<1>().unlock(a);
8      }
9      globalLockedAddresses.pop();
10 }
```

Listing 4.17: Simplified version of the `unlock` function in the cache.

functions (Line 11 in Listing 4.16 and Line 7 in Listing 4.17).

The extension itself is presented in Listing 4.18. The simulator stores the address in a two-layered contiguous array. We make use of the fact that we can organize the locked addresses in a similar manner as the cache itself. The simulator traverses the first layer via the index of the address, while the second one corresponds to a set of the cache and hence, we have to check every tag in the second layer. Observe that the extension utilizes a *lazy locking* mechanism. The actual locking is deferred to a later point, namely, in the `before_transition` function. This is a callback invoked by the cache controller when it receives a message on the interconnect and the message affects a cache line in its cache. The function is called before the state machine gets invoked (basically, before Line 6 in Algorithm 4.3). The callback checks whether the address of the cache line is

```
 1 class LockedCache {
 2   std::array<std::array<
 3     std::optional<uint32_t>, N_WAY>, N_ENTRIES> lockedLines;
 4
 5  public:
 6   auto lock(uint32_t address) {
 7     // Add to lockedLines based on index of address
 8   }
 9   auto unlock(uint32_t address) {
10     // Remove from lockedLines
11   }
12
13   auto before_transition(CacheLine &line) {
14     auto entry = find_entry(line.address);
15
16     if (entry.has_value() && line.state().is_writable()) {
17       line.wait_until([&](auto &) {
18         return !entry.value()->has_value()
19             || entry.value()->value() != info.tag;
20       });
21     }
22   }
23 };
```

Listing 4.18: Simplified version of the **LockedCache** extension.

supposed to be locked in addition to checking whether the cache line is writable. The latter condition is important because we only want to stall a message if it would lead to an eviction or downgrade of the cache line. Checking for writability is sufficient, because once the cache has acquired the address in a read-write state, we want to keep this state until the address is unlocked again. Note that this means that when `lock` is called with some address, the cache does not necessarily contain the cache line corresponding to this address. Hence, we need to request mutable access to a cache line on Line 12 in Listing 4.16 to ensure that the lock mechanism is actually enforced.

Nevertheless, if all conditions on Line 16 are satisfied, the simulator stalls the cache line until the address was unlocked again (Lines 17-19). The reason it stalls is because the callback is executed by the message handler thread. Therefore, if the callback does not return, the message handler is blocked and hence, the entire interconnect cannot process any further messages.

We want to emphasize that this approach is not necessarily deadlock-free. Consider the following situation where thread 1 locks two cache lines A and B (e.g. due to an unaligned atomic operation). Recall that the simulator needs to request writable access for both lines. Next, another thread also requests access to line A. Consider the following order of these requests on the bus:

$$W_1(A) \implies D_1(A) \implies R_2(A) \implies W_1(B) \tag{4.1}$$

$W_1(A)/R_1(A)$ denote a write and read request from thread 1 to cache line A. $D_1(A)$ corresponds to the response containing the writable data for thread 1. Note that the execution shown above deadlocks the entire system because thread 1 cannot progress as it awaits the lock for cache line B. However, the message $R_2(A)$ entered the bus before $W_1(B)$ and hence, must be processed in the same order. But thread 1 stalls the processing of $R_2(A)$ due to the lock. The reason is that the protocols presented in this thesis require a pipelined bus. Pipelined means that the total order of the requests on the bus matches the order of the corresponding responses [NSH+20, p. 134].

This issue can be resolved in different ways:

- **Backoff mechanism**: Continuing our example from above, we could extend the simulator to have first-class support for locking multiple cache lines which we call multilock. Then thread 1 could detect that thread 2 tries to acquire a cache line from one of the cache lines in the multilock, which 1 has already locked but it has not yet managed to lock all of those cache lines. If thread 1 enters this situation, it can backoff to give up on cache line A in order to process the message by thread 2 in order to then restart the transaction of locking both lines A and B.

  However, this approach does not guarantee forward progress of thread 1 because it may be continuously interrupted by other threads. In fact, the backoff mechanism can theoretically lead to livelock of the system in case two threads try to multilock both A and B and their messages always intertwine similarly to the execution shown in 4.1.

- **Multi-Request transactions**: The issue can be easily prevented if multiple requests can be atomically sent on the bus. This guarantees that no request from another thread can interfere with the locking of multiple cache lines.

- **Non-atomic transactions**: An alternative to a pipelined bus is a split transaction bus. This means that responses might have a different order on the bus than their corresponding requests. While this does not require adaption of the simulator, the cache coherence protocol itself must be equipped with additional intermediate states to account for this behavior. [NSH+20, p. 132]

Since VADL as a language does not support multicore systems yet, this issue cannot occur in the CAS as of this thesis. Therefore, we decided to leave a solution open for future work.

Finally, we want to briefly mention how we adapted VADL to divert memory accesses to the cache. Recall that instructions are defined in the ISA section and hence, cannot utilize the cache directly. As mentioned in Section 4.3.1, the language provides the `dataCache`

```
[ size = 8 ]                // Amount of slots in the buffer
[ coalescing ]              // Whether coalescing should be activated
[ attachedTo L1 ]      // Backend to write/read data to/from
[ flushPolicy opportunistic ] // Flush if cache line is readable
[ write buffer ]
logic writebuffer
```

Listing 4.19: Example definition of write buffer.

annotation on a cache definition so that the frontend knows what cache it should use as a new target for reads and writes from/to memory within an instruction semantics. We added a new pass to the VADL frontend called `VirMemoryAccessReplacementPass` which conducts the replacement on the VIR code.

4.4 Write Buffers

In Section 2.2.3, we showed the advantages of using a write or store buffer in a microprocessor. As part of this thesis, we extended VADL to support defining such buffers in the MiA section of a specification. We decided to utilize the powerful **`logic`** element of VADL. Figure 4.19 provides an example for a write buffer definition. The annotation `write buffer` corresponds to the primary annotation and marks the logic element as a write buffer component. All others are called secondary annotation and describe several properties of the buffer. The `size` annotation denotes how many entries the buffer contains. The size of an entry implicitly depends on the `attachedTo` annotation which tells the VADL frontend for which component it should buffer stores. This can be either a memory definition, in which case the size corresponds to the data bus width of the MiA, or a cache definition, where the size matches the size of a cache line. We decided against letting VADL developers freely choose the size of an entry because when attaching to a cache, an entry size that mismatches the cache line size, can lead to undesired alignment issues when data from the buffer is written back. Hence, we see no practical reason to allow arbitrary write buffer entry sizes. Next, `coalescing` defines whether the write buffer may coalesce or overwrite a write with an existing entry that matches the same address. This may or may not be allowed depending on the memory model of an ISA. Last but not least, the `flushPolicy` defines the behavior of when data should be flushed from the buffer. We defined a preset of three policies:

- **lazy**: In this mode, the simulator flushes only when absolutely necessary such when the buffer is full. Note that if the `flushPolicy` is omitted, this is the default policy.
- **opportunistic**: In this mode, the precise behavior depends on the attached component. In case of a cache, when new data is written to the buffer, it checks whether the cache line corresponding to the address happens to be in the cache.

If yes, all entries affecting the cache line are written to the cache. The idea of this policy is that if the data already exists in the cache, a write to it might be nearly for 'free'. If the cache line happens to be in a *Modified* state, this will be the case, as in practice, L1 access latency tends to be very low [Int24]. Even if the state does not allow writability, upgrading the cache line might not require a lot of cycles. For instance, if the cache employs a **MESI** coherence protocol, the cache line could be in *Exclusive* state and hence, an upgrade to the *Modified* state comes without consulting the network first. Hence, we believe that this policy might be particularly effective for coalescing buffers. Non-coalescing buffers require special considerations because the write order must be maintained. Therefore, if the system wants to flush one specific entry, all writes ordered before it must be written back first. This might limit the effectiveness of the `opportunistic` policy.

Finally, when using a memory as the attached backend, we assume that the system can always write back to memory without contention. Needless to say, this will be limited by the amount of read and write ports in case the CPU accesses the memory in other ways (e.g. for fetching an instruction).

- **fullCacheLine**: This flush policy flushes an entry once the entry becomes full. Our intuition is the following: Data bus widths are usually larger than the size of a typical load instruction. For instance, the amd64 manual mentions that certain instructions may utilize write-combining buffers which combine 64 bytes in order to transfer them in a single burst to main memory [Adv24a, p. 203]. Since entries in the write buffer are always aligned on the cache line size or data bus width, the `fullCacheLine` policy takes advantage of this by flushing an entry once it is full. In case of a memory, a single transfer over the data bus is required and in case of a cache, the corresponding cache line needs to be requested only once. In fact, one could come up with an additional optimization to not request the data of the cache line, since it will be replaced entirely anyway, but to broadcast the new data from the write buffer over the interconnect. Needless to say, the cache coherence protocol needs to account for this scenario.

  Lastly, Figure 4.4 shows an example of how and when a single entry in a coalescing buffer turns full. Note that in a non-coalescing buffer, a slot can only be full if a single write matches the size of an entry and is aligned. Hence, this policy might be less useful in this case.

Based on the non-functional criteria described in Section 4.3.2, we implemented the write buffer simulator in C++ 20 using only standard library features apart from boost for its logging facility. The Algorithm presented in 4.4 shows how data is read from a buffer.

$|r|$ denotes the size of r. Basically, the algorithm handles every byte of the output value r individually. The reason is that partial writes might lead to fragmentation of data, especially in a non-coalescing buffer. Thus, for each byte and entry in the buffer (Lines 1-2), it checks whether the corresponding entry contains a value for the address $a + i$ (Line 3). If yes, we continue from Line 1. If no entry contains the value, then the buffer

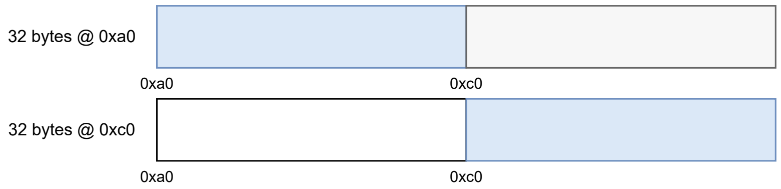

Figure 4.4: Example of how two individual stores are merged into the same write buffer entry. The first store writes 32 bytes at address 0xa0. The grey-shaded area remains unused. The second store writes 32 bytes to address 0xc0 resulting in a merge with the entry of the previous store.

Algorithm 4.4: Basic algorithm for reading from the buffer.

```
  input  : FIFO-Buffer $b$, Address $a$
  output : Result r
1 foreach $i \in \{0, \ldots, |r|\}$ do
2     foreach $e \in reverse(b)$ do
3         if $(a + i) \in e.a$ then
4             $r[i] \leftarrow e[a + i]$;
5             Continue from 1;
6         end
7     end
8     $r[i] \leftarrow \text{LLC}(a + i)$;
9 end
```

must fall back to its attached storage, either a cache or memory (Line 8). Observe that the algorithm must iterate the entries in reverse order, assuming a FIFO buffer. This is necessary for a non-coalescing buffer because a read must return the value from the latest write in order to comply with program order. In terms of runtime complexity, the algorithm is in $\mathcal{O}(|r| * |b|)$ where $|r|$ and $|b|$ respectively denote the size of r in bytes and the amount of entries in the buffer. Both the size of r and b should remain reasonably small in practice but considering this algorithm must be invoked for every read regardless of whether the buffer actually contains any data of the requested address, this complexity might have a noticeable impact on the runtime of the simulation. We evaluate the performance of our implementation in Section 5.

Nevertheless, we propose some ideas to improve the complexity. For coalescing buffers, the FIFO property is not necessary. Hence, the simulator could maintain entries in a hash table instead, where the aligned address maps to the corresponding entry. As all hash table operations are constant, this improves the complexity of the algorithm to $\mathcal{O}(|r|)$. When using a non-coalescing buffer, the order of writes must be maintained. In addition, the same address might have multiple entries. Therefore, the hash table idea could be used as an addition to the FIFO queue. The table establishes a mapping between a specific address and a reference to an entry containing the latest value. Another tiny improvement could be to read more bytes if possible. For instance, if the algorithm finds an entry with a valid value for some address a, the principle of locality states that this entry might also contain data for the subsequent bytes $a + i$, at most until the end of the entry. This idea is based on two assumptions: Firstly, if an architecture allows only aligned memory accesses (e.g. RISC-V), then loading a value from write buffer can never cross an entry boundary (unless the write buffer has an unusual entry size configuration, however, we assume VADL developers use sensible values in this case). Secondly, consider reading data from some address a that was preceded by a write to the same address. From a practical point of view, the size of the read request matches the size of the preceded write request. Hence, a read of a different, particularly larger, size from the same address can be considered an exceptional case.

Algorithm 4.5: Basic algorithm for non-coalesced writing to the buffer.

```
   input  : FIFO-Buffer $b$, Address $a$, Data $d$
 1 $cAddress \leftarrow \text{align}(a)$;
 2 $cEntry \leftarrow ()$;
 3 foreach $i \in \{0, \ldots, |r|\}$ do
 4     if $align(a + i) \neq cAddress$ then
 5         $b.\text{append}(cAddress,\ cEntry)$;
 6         $cAddress \leftarrow \text{align}(a + i)$;
 7         $cEntry \leftarrow ()$;
 8     end
 9     $cEntry.\text{add}(a + i,\ d[i])$;
10 end
```

The write procedure for a non-coalescing buffer is shown in Algorithm 4.5. The basic principle is similar to the read function. The function writes the data d byte-by-byte into an entry of the buffer. Since we need to ensure that each entry is aligned to its size, we keep track of the current entry and its associated aligned starting address, which is calculated using the *align* helper function. Once the address of the current byte crosses an alignment boundary (Line 4), we add the entry to the buffer and create a new one to append to (Lines 5-7). The data itself is copied on Line 9, where $d[i]$ denotes the ith byte of d.

The coalesced version in Algorithm 4.6 is quite similar. The procedure differs on Lines 2 and 6, where it utilizes a helper function called *findOrCreate*. This function finds an entry within buffer b which represents the address a (Line 2) or $a + i$ (Line 6). If it cannot find such entry, it allocates a new one and appends it to b. The ith byte of d is added on Line 8, which - unlike in the non-coalesced version - might overwrite a previously written value.

When looking at the complexity of both procedures, Algorithm 4.5 has a linear runtime in $\mathcal{O}(|r|)$. For Algorithm 4.6, this depends on the data structure of b. If it is an array, then

Algorithm 4.6: Basic algorithm for coalesced writing to the buffer.

```
input  : FIFO-Buffer $b$, Address $a$, Data $d$
$cAddress \leftarrow$ align($a$);
$cEntry \leftarrow$ findOrCreate($b$, $a$);
foreach $i \in \{0, \ldots, |r|\}$ do
    if $align(a + i) \neq cAddress$ then
        $cAddress \leftarrow$ align($a + i$);
        $cEntry \leftarrow$ findOrCreate($b$, $a + i$);
    end
    $cEntry$.add($a + i$, $d[i]$);
end
```

findOrCreate has linear complexity, meaning the total runtime is quadratic in the worst case, where each entry holds only one byte. However, b can be changed to a hash map in a coalesced buffer, which then yields a complexity of $\mathcal{O}(|r|)$. However, both algorithms offer some improvement. For instance, instead of writing each byte individually, an entire block of data can be committed to the entry as long as the address of the highest-order byte of the block stays within the alignment of the entry. Consider the following example: Assume all entries are aligned on a 4-byte boundary. Next, we store four bytes at address 0x6. Hence, the first two and other two bytes go into the entries representing address 0x4 and 0x8 respectively. The algorithms presented in 4.5 and 4.6 require four iterations (one per byte) to commit the write. Our improved version would require only two, since the write consists of two blocks at addresses 0x6 and 0x8. In theory, the improved version could significantly speed up the common case of aligned writes. Recall that an aligned memory access cannot cross an alignment boundary of an entry (assuming a reasonably configured write buffer). Hence, an aligned write always results in constant runtime because the algorithm must commit only a single block.

Finally, we added the possibility to specify the memory model of a memory definition. A VADL engineer may select from a one of the following models:

- Sequential consistency (SC)
- Total store order (TSO)
- RISC-V Weak Memory Ordering (RVWMO)

Listing 4.20 shows an example for how to define these orderings. As mentioned in Section 2.3, the concept of a write buffer is not compatible with all memory models. For instance, coalescing stores violates TSO, but works in conjunction with RVWMO. The VADL frontend verifies whether a certain combination is legal. Note that annotating a memory model is purely for specification purposes and is not considered by any of the VADL generators in the context of this thesis.

```
instruction set architecture FOO = {
    [ ordering sequentialConsistency ]
    memory MEM_SC: Bits<8> -> Bits<8>

    [ ordering totalStoreOrder ]
    memory MEM_TSO: Bits<8> -> Bits<8>

    [ ordering rvWeakMemoryOrdering ]
    memory MEM_RVWMO: Bits<8> -> Bits<8>
}
```

Listing 4.20: Memory definitions with annotated memory models.

CHAPTER 5

Evaluation

In this chapter, we measure several aspects of our work in this thesis. We introduce the reader to our benchmarking environment as well as the key features we wanted to evaluate. Finally, we present our findings as well as conclusions based on our results.

5.1 Benchmark Setup

We consider different aspects of our work. Regarding write buffers and caches, we want to evaluate the impact of the simulation on the runtime using Embench, a collection of small benchmarks suitable for evaluating embedded platforms[1]. We decided to use Embench because they do not require user mode emulation and memory allocations. We used our VADL implementation of the RV32 instruction set, including the A- and M-extensions, and compared four MiAs:

- `P1`: This is a basic single stage MiA without any caches, buffers or other extras.
- `P1-WB`: Similar to `P1` but additionally includes a write buffer.
- `P1-Cache`: Similar to `P1` but additionally includes an L1 data as well as instruction and an L2 cache.
- `P1-Cache-WB`: Similar to `P1-Cache` but additionally includes a write buffer.

In addition, we compare our results to the gem5 simulator [BBB$^+$11, LAA$^+$20]. gem5 is a powerful simulator supporting many features. It mainly differs between two simulation types, *atomic* and *timing*. The former is more comparable to an ISS as memory access latencies are not considered for instance. On the other hand, a *timing* CPU models

[1] `https://github.com/embench/embench-iot`

Component	Name
CPU	Apple M1
Memory	16 GiB
Host Compiler	gcc 14.1.1
Rv32 Cross Compiler	gcc 13.2.0
Benchmark Suite	embench 54fd9a0
RISC-V Compliance Suite	riscv-tests 51de008
gem5	23.1

Table 5.1: Configuration of the benchmarking environment.

Property	Value
Line Size	64 bytes
L1I/L1D/L2 Size	64/64/256 kiB
L1I/L1D/L2 Associativity	4/4/8-way
Eviction Policy	LRU
Coherence Protocol (gem5)	Snooping-based MOESI
Coherence Protocol (VADL)	Snooping-based MESI

Table 5.2: Cache hierarchy setup for `P1-Cache`, `P1-Cache-WB` and `gem5 Timing+Cache`. Based on the Cortex-X1 [Arm20].

latencies of memory and caches more precisely. It should be noted that this type of simulation is still not cycle-accurate but certainly more comparable to our CAS. We defined two configurations. The first is a simple RV32 machine without a cache hierarchy (named `gem5 Timing`), while the other consists of the same memory hierarchy as `P1-Cache` (called `gem5 Timing+Cache`). We tried to match the memory hierarchy configurations of `P1-Cache` and `gem5 Timing+Cache` as closely as possible. The latter implements a snooping-based MOESI protocol [LP24] as part of gem5's 'classic caches', while our CAS uses the `snooping_wb_mesi` coherence protocol described in Section 4.3.1. The exact configuration is shown in Table 5.2 and the corresponding VADL definitions in Appendix D. We derived the parameters and structure of the hierarchy from the Cortex-X1 to showcase a realistic example [Arm20]. We compared mainly two aspects of the simulation:

- **Performance**: In alignment with our design goals described in Section 4.3.2, the simulation should be as fast as possible to keep the impact on the entire simulation low.
- **Accuracy**: Since both our gem5 configuration and CAS share a similar setup, the amount of cache misses and hence, the cache miss rate should be close to each other.

Last but not least, we want to evaluate the impact on the simulation when using address translation. We use our VADL implementation of the Sv32 memory scheme presented

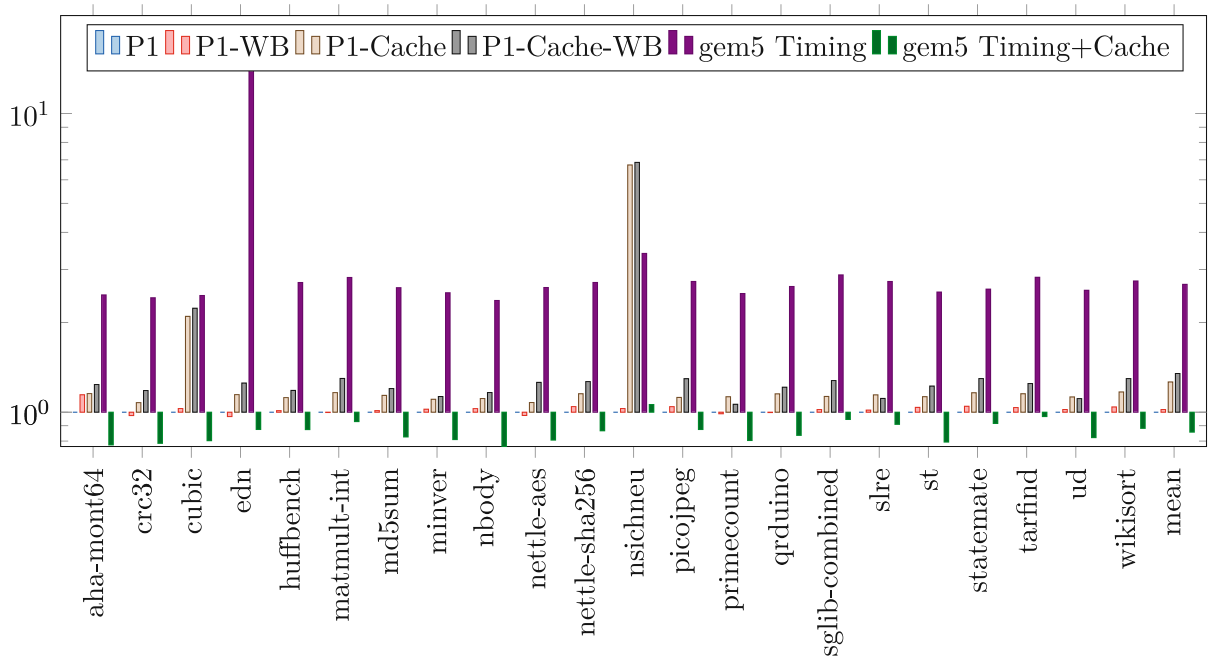

Figure 5.1: Embench runtime of CASs using RV32IAM (relative to P1, smaller is better).

in Appendix B. We compare our results with the same VADL specification without any memory translation as well as a simple gem5 setup using an *atomic* CPU which is more comparable to an ISS. For reference, we also executed Embench on Spike[2], which is a hand-written RISC-V ISS. For memory translation, we map the virtual address to the exact same physical address in the page table which allows us to use the same unmodified binary.

We setup our benchmark environment as presented in Table 5.1. For runtime evaluation, we executed each configuration 25 times per benchmark in order to retrieve statistically relevant results.

5.2 Results

We start with validating the correctness of our cache simulation. First, it is important to account for what aspects of a simulation are used for validation. For instance, a timing simulation needs to consider exact durations of sending a message via an interconnect from one entity to another. Some cache simulators even precisely simulate the energy consumption of accessing the cache. Since our CAS is more on the functional side, we consider the cache miss rate as our main statistical property. We compare our system to the gem5 configuration shown previously. One might think the optimal way to validate the behavior of a cache simulation is by comparing it to hardware performance

[2] https://github.com/riscv-software-src/riscv-isa-sim

counters. [BKP20] discuss that using hardware events are hard to use correctly. Firstly, an event can mean different things across different CPU vendors. For instance, consider an unaligned memory access which crosses two cache lines. This operation triggers a single L1 miss but does this mean that both cache lines were not available or just one of them? Secondly, the authors mention that a mismatch on a certain metric is hard to interpret because hardware is usually not open-source and hence, researchers can only speculate where the mismatch originates. However, when comparing a simulator to another open-source one, researchers can look into the code of the program to potentially understand the source of the mismatch. Lastly, trying to fix the gap between the simulation and hardware performance counters might lead to over-generalization because there is no guarantee that the simulator will also correctly work for other CPU systems. Hence, [BKP20] suggest validating a simulator against other cache simulators accepted in academia which are in turn verified or validated against real hardware.

gem5 is often used for validating the correct behavior of a cache simulator and hence, we decided to do the same [BKP20]. As mentioned above, we compare cache misses and cache miss rates of the L1D cache. Note that we execute gem5 in user mode emulation while our CAS runs in bare metal mode. Hence, the compiled benchmarks are different binaries where one artifact is a static Linux and the other a bare metal executable. However, we use the same compiler (seen in Table 5.1) and same optimization level (-O2). We verified some parts of the compiled code of the benchmarking parts to ensure that the generated code matches. We noticed that the Linux executable has a substantial setup and teardown code which impacts the behavior of the cache simulation of gem5. Hence, we compiled a binary with an empty main function and measured the amount of L1D cache accesses and cache misses. This led us to a total number of 40988 memory accesses and 807 cache misses. We subtracted these values from the results we received when running the regular benchmarks. Hence, some values yield a negative amount of total cache misses which stems from inaccuracies of our estimation. All results can be found in Table E.2 in Appendix E.

At first sight, the mean relative error (36.73 %) seems high, however, note that the mean absolute deviation of 0.0008 % is considerably small. Hence, this makes the relative error hard to interpret because even tiniest deviations in the cache miss rates can have a large impact on the relative error. We tried to increase the number of repetitions within each benchmark, however, the number of cache misses for both `P1-Cache` and `gem5 Timing+Cache` remained stable. This suggests that all memory accesses of the benchmark fit into the entire L1 data cache.

Next, Figure 5.1 presents the runtime of the different setups relative to the `P1` CAS. While the write buffer simulation (`P1-WB`) increases the runtime by a mean factor of around 1.02, the cache simulation (`P1-Cache`) has a slightly higher impact on the overall simulation with a mean factor of 1.26. On the other hand, the timing gem5 configuration is around 2.68 times slower on average, while the configuration with caches is interestingly considerably faster, specifically by a factor of around 0.86 compared to `P1`. We analyzed this oddity further by profiling both configurations of gem5 using the Linux tool `perf`. We looked at several benchmarks and found that the cache simulation is considerably faster than gem5's memory simulation. For instance, the *wikisort* benchmark spends around 222 ms within cache-related functions compared to around 1030 ms spent in the DRAM simulation of the memory-only version. As mentioned previously, the benchmarks seem to be tiny enough so that most of their data fit into the entire L1 data cache. Hence, the memory simulation is barely involved in the `Timing+Cache` configuration (`perf` measured around 0.3 ms). If we include our `P1-Cache` CAS into the comparison, we observe that *wikisort* spends around 258 ms in the cache which indicates that our cache simulator might still be competitive to gem5's implementation. However, this requires further evaluation. In addition, gem5's simulation is more detailed in certain aspects especially when considering the entire system which might also incur additional overhead. However, we want to highlight that the 'classic caches' of gem5 use a fixed cache coherence protocol while our cache simulator provides more flexibility in this regard.

Next, when looking at Figure 5.1, the *nsichneu* benchmark is a significant outlier. We again used the Linux tool `perf` to retrieve a flame graph of the program execution, which revealed that a lot of CPU time is lost during synchronization. For instance, the `P1-Cache` CAS spends around 29 % of the entire execution on reading from the L1 instruction cache. 20.8 % out of the 29 % of the total execution are spent on synchronization and waiting within the reading function. If we compare these numbers to a benchmark where `P1-Cache` performs better, such as *primecount*, reading from the instruction cache consumes only around 11 % of the total runtime. Analyzing flame graphs from other benchmarks revealed that a lot of time is spent reading from instruction cache while the data cache usually has a lower impact. Needless to say, an instruction cache is constantly being used because every instruction fetch accesses it, while the data cache is accessed only in certain instructions (mostly loads and stores). We provide an overview of how much time is spent on reading/writing from/to the L1 instruction and data caches compared to the total runtime in Table E.3. We observe a similar effect on the *cubic* benchmark where the instruction cache consumes around 18 % of the entire execution time. When looking at Figure 5.1, we see that the execution time of *cubic* is also noticably above the mean value.

Our first intuition was whether the memory access pattern of the benchmarks influence the cache simulation. In Table 5.3, we consider the Pearson correlation between absolute runtime and total amount of memory accesses as well as absolute runtime and cache miss rate of the L1 instruction cache. As expected, there is a strong correlation between the amount of accesses and runtime as every additionally executed instruction leads to longer simulation time and every instruction must be fetched from the instruction cache. There seems to be a weak correlation between the cache miss rate and runtime as well. If we consider the results from Table E.1, we see how *cubic* and *nsichneu* benchmarks have a higher miss rate (0.7585 % and 4.5757 % respectively), while the median of all benchmarks is at 0.0024 %.

Recall that our architecture uses a separate thread per interconnect to handle messages and distribute them to all relevant entities. Hence, when some requested data is not

Configuration	Time-Absolute Memory Accesses	Time-Cache Miss Rate
P1-Cache	0.745	0.547
P1-Cache-WB	0.740	0.546

Table 5.3: Pearson correlation between several statistical events of the L1 instruction cache.

in the cache, at least one message must be sent to the interconnect thread. However, it might be necessary to evict a cache line first in order to make room for the newly requested data. If said cache line is dirty, it must be explicitly sent back to the upper level cache but the initiating thread must wait for the interconnect to finish before the actual data can be requested. Furthermore, the same applies to the shared L2 cache. Hence, for a single instruction fetch, the simulator must send four messages (L1I cache eviction, L1I read request, L2 cache eviction and L2 read request) and synchronize on each of them in the worst case. gem5 is limited to a single thread and hence, does not suffer from synchronization overhead [ZCCJ^{+}23].

We infer that a multithreaded architecture might not be beneficial for simulating a single-threaded environment. Nevertheless, it would be interesting to see how our cache simulator scales when multiple threads are involved. As mentioned above, gem5 cannot utilize more than one thread, even for simulating multicore systems [ZCCJ^{+}23]. Unfortunately, the current CAS cannot simulate multiple threads either and hence, we could not investigate the scalability of our cache architecture in this thesis.

Figure 5.2 presents the execution time of our RISC-V VADL specification with and without address translation as well as the RISC-V ISS Spike and a simple gem5 configuration using an *atomic* CPU. We used Spike as a baseline for our measurements. Note that the ISS with address translation enabled has no practical impact on the execution time of the simulator. Our ISS with and without address translation are slower by a factor of 43.88 and 44.09 compared to Spike. Originally, we expected a tiny overhead stemming from additional memory accesses, similarly to a real system. Recall that Sv32 features a two-level table walk. The uppermost 10 bits of the virtual address determine the first-level entry in the table. Since the executable code sections of our benchmarks are densely packed, those uppermost bits are equal and a second level page table is required. Hence, executing a single instruction in the simulator takes three memory reads. Two for accessing the first- and second-level page table entries and one for actually fetching the instruction from physical memory [RIS21, p. 82]. Therefore, real systems employ TLBs to cache page table entries in order to avoid additional memory latencies [PH17, p. 843].

Furthermore, our ISS, even with enabled address translation, is significantly faster compared to gem5, which is slower by a factor of around 98.75 compared to our baseline. In addition, gem5 uses user mode emulation and hence, does technically not even require address translation. Spike is still considerably faster than our generated ISS, however, note that spike is hand-written and thus, does not offer the flexibility VADL provides.

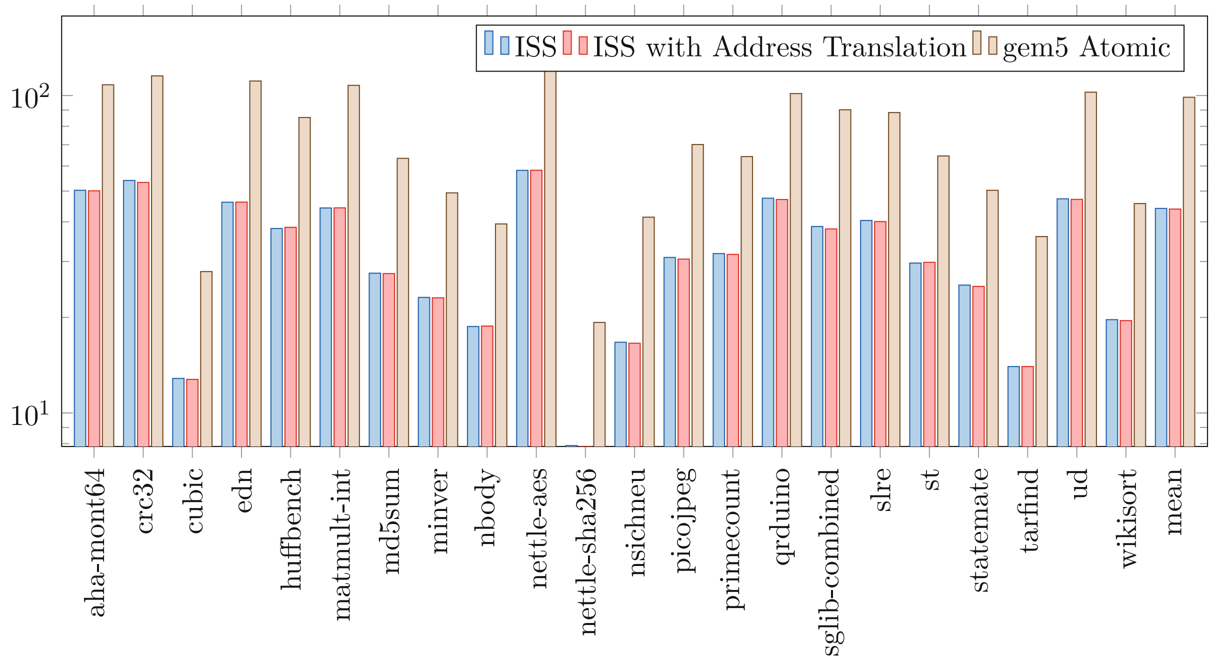

Figure 5.2: Embench runtime of ISSs using RV32IAM (relative to Spike, smaller is better).

ISA	Amount of Instructions	LoC with Models	LoC w/o Models
RISC-V	11	77	136
AArch64	57	277	1078

Table 5.4: Lines of code without comments for implementing several atomic instructions.

Finally, we want to evaluate the expressiveness of our language constructs for atomic instructions, explained in Section 4.1. We implemented the entire RV32 A-extension from the RISC-V specification [RIS19, p. 47], with the exception of the acquire-release semantics bits. We verified the correctness using the test programs *rv32ua-p-** from the `riscv-tests`[3] suite which is maintained by the RISC-V International organization itself. As seen in Listing C.1, our VADL specification requires 77 lines of code without comments. Needless to say, lines of code is only a rough measurement due to not every line of code being equally expressive. We followed a sensible style where we put one statement or definition per line and limit each line to 80 characters. In addition, we utilize VADL's powerful syntactical macro system, called models, to avoid repetitive boilerplate code. The model `AMOtype` defines a typical fetch-and-modify structure, including an appropriate encoding and assembly definition. Only the `LR` and `SC` instructions have been defined separately. Without models, the A-extension specification increases to 136 lines of code.

We also implemented almost all atomic instructions provided by ARMv8 AArch64,

[3] https://github.com/riscv-software-src/riscv-tests

again with the exception of the acquire-release semantics bits. This includes Load-Exclusive and Store-Conditional as well as all operand size version (1, 2, 4, 8 and 16 bytes) [Arm21, C3.2.6, C3.2.12.1 - C3.2.12.]. As seen in Table 5.4, our implementation requires 277 lines of code with extensive use of models. The models particularly helped us to share encoding and assembly definitions for atomic instructions as well as reusing code for the different operand sizes. When expanded, the resulting VADL specification requires 1078 lines of code. While ARM provides an official but internal validation framework [MRSM16], there exists no proper publicly available ISA test suite for AArch64 to the best of our knowledge. Therefore, we were not able to validate the correctness of our implementation.

CHAPTER 6

Future Work

As discussed in Section 5, the synchronization overhead in the multithreaded architecture of our cache simulator is considerable in single-threaded use cases. Firstly, we would like to know whether the advantages of our design become more apparent in multicore scenarios. As of the time of writing, VADL does not support simulating multicore CPUs. However, the cache simulator can be used independently from VADL. Thus, one idea to evaluate the scalability of our architecture is to record all memory accesses of a multi-threaded benchmark using another simulator (e.g. gem5) and replay this recording in our cache simulator. Furthermore, this would enable comparing our simulator to other parallel benchmark suites, such as SPLASH-2 [WOT+95] or PARSEC [BKSL08]. Both are often used for evaluating multi- and manycore simulation performance. Nevertheless, we want to see how the simulation can be improved in single-threaded use cases. A basic approach could be to provide a single-threaded fast path that can be enabled at compile- or runtime. The current architecture, however, heavily relies on a multithreaded foundation. Developing and maintaining potentially two different simulators can lead to unnecessary maintenance burden. Hence, we propose a different solution to redesign the current multithreaded architecture into an asynchronous one. We believe that the amount of changes necessary is small because our simulator already divides some components (e.g. cache, interconnect and cache controller) into individual threads. We can repurpose these into tasks in an asynchronous environment. While this change might seem subtle, it might have several advantages:

- A real-world hardware cache is also asynchronous by design. Hence, an asynchronous architecture could lead to an easier more accurate behavior by simulating the parallel/asynchronous effects of the real system.
- Many asynchronous frameworks have a scheduler that can be both single- and multithreaded. Hence, scaling can be applied as necessary. For instance, our current CAS does not support multithreaded CPUs yet and therefore, scheduling

> the asynchronous tasks on a single thread is certainly more efficient. However, a properly multithreaded simulator might scale better with the number of simulated cores and hence, an asynchronous cache simulator can be scaled on demand.

An alternative approach is to enable integration of other cache simulators or even embed VADL's generated simulator into other frameworks. gem5 is a mature and widely established computer-system simulation framework and due to its modular architecture, open to extension. The VADL frontend could generate a CPU simulator that can be used by gem5 which gives users access to the general power of gem5 such as user mode simulation, detailed memory simulation and simulation of other computer components such as graphics cards. Our write buffer simulation is less impactful on the simulation. Nevertheless, we think that there are possibilities to easily improve performance which we proposed in Section 4.4.

Finally, our cache simulator simulates detailed behavioral aspects of several cache coherence protocols. Currently, our system only tracks the amount of read and write requests as well as whether an access hits a line in the cache. Hence, we mainly considered the cache miss rate for validation. For accuracy purposes, we believe it would be useful to compare additional statistical events, such as state transitions, messages sent via the interconnect or access latencies, to other simulators (e.g. gem5). Since some of the aforementioned properties are already simulated, we believe it should be trivial to add these additional metrics.

On the language side of VADL, there are still several open tasks. While VADL now enables developers to define atomic instructions, this is not sufficient for constructing multicore systems. Most notably, defining the memory model is an important aspect of an ISA. We currently provide a set of predefined models, however, it might be desirable to allow defining arbitrary concurrency models. Sail supports this aspect for example [ABC+19]. But still, it should be noted that describing memory models is a complex matter on its own [NSH+20, p. 253]. Nonetheless, VADL still needs to offer primitives to act on the memory model. An important example are memory fence instructions which ensure that certain loads and/or stores cannot be reordered. This is an interesting use case for VADL because as of this writing, an instruction cannot influence the MiA due to the strict separation between ISA and MiA. However, a fence instruction directly impacts the behavior of the MiA by influencing the order of how instructions may be executed. A possible solution might be to allow overriding instructions in the MiA section of a specification. Compare this to interfaces known from other programming languages such as Java where an interface may define default behavior for certain functions but the implementor may choose to override their implementation. VADL could also provide primitives for handling memory model specific aspects.

However, we believe that a more general feature can be useful in other cases too. For instance, ISAs provide cache management instructions for clearing a cache or evicting certain cache lines. Again, the specific semantics of a cache management instruction depend on the MiA. For example, a cache flush instruction is a no-op in a cache-less system.

Next, we want to note that while VADL now allows specifying atomic instructions, there are different execution approaches in practice. For instance, it is common for interconnects to apply a fetch-and-modify directly without interaction of the Core's ALU. ARM's Cortex X1 differentiates between *near* and *far* atomics where the latter is executed by the L2 or L3 cache if supported [Arm20, p. A6-79]. This is certainly a very specific MiA implementation detail but something that VADL could support in the future as well. In general, VADL does not consider buses or interconnects. In our cache design presented in Section 4.3.1, caches are connected directly with each other. This is fine for purely functional evaluations, however, one of VADL's features is to be able to generate RTL designs. In this case, the topography and type of the interconnect becomes relevant. In addition, proper cycle-accurate simulation must take latencies of the interconnect into account as well. For instance, gem5 [BBB+11] and Multi2Sim [UJM+12] allow defining caches and interconnects separately.

Finally, VADL currently does not support defining TLB. However, the overhead of virtual memory management has a considerable impact on real hardware. It might be possible to reuse the cache definition introduced as part of this thesis. What might be challenging is how the TLB can be integrated into the specification. A TLB is clearly a microarchitectural detail but the address translation process is part of the ISA.

CHAPTER 7

Conclusion

This work presented our extension of VADL to support defining address translation functions, atomic instructions and caches as well as memory hierarchies. Furthermore, we adapted both the ISS and CAS to support these features, including a multi-threaded and detailed cache simulator. The compiler and hardware generators were not considered.

The language design for atomic instructions was derived from current ISAs and what their functionalities require in order to implement them in VADL. We believe that our language constructs provide enough flexibility to add arbitrary RMW and CAS instructions as well as LR/SC pairs. We implemented the entire A-extension of RISC-V RV32 and all atomic instructions for our ARMv8 AArch64 VADL specifications and successfully validated the correctness of the former.

For address translation, we considered several address translation schemes in existing ISAs. While the basic concept (e.g. table walks) remains quite similar, the exact mechanism differs in the minutest details. Hence, we decided to enable users to accurately implement an address translation algorithm using existing VADL language features instead of following a more generic, declarative style. This should also provide the possibility to implement other related features found in modern ISAs, such as pointer authentication.

The cache simulation is certainly the largest contribution in this thesis. Our simulator provides a detailed simulation of several cache coherence protocols, interconnects and replacement policies while maintaining high throughput. We observed that its performance mostly depends on the cache miss rate which we could trace back to synchronization overhead when accessing data from an upper level cache or memory. Last but not least, we integrated (non-)coalescing write buffers into the MiA section of VADL. Users may also choose from a range of flush strategies, defining how the simulator writes entries in the buffer back to cache or memory.

APPENDIX A

Common Atomic Primitives

This chapter presents atomic instructions provided by several common ISAs. We map their mnemonic to our own definition of atomic primitives as defined in Table 4.1.

Instruction	Primitive	Additional notes
`LR.W/D`	LR	
`SC.W/D`	SC	
`AMOSWAP.W/D`	SWAP	
`AMOADD.W/D`	FAM	
`AMOAND.W/D`	FAM	
`AMOOR.W/D`	FAM	
`AMOXOR.W/D`	FAM	
`AMOMAX[U].W/D`	FAM	`U` . . . Unsigned
`AMOMIN[U].W/D`	FAM	`U` . . . Unsigned

Table A.1: Atomic instructions provided by RV32 and RV64 with the A-extension. `W/D` correspond to 4/8 bytes respectively. [RIS19]

Instruction	Primitive	Additional notes
`LL/LLD/LLDP`	LR	4/8/16 bytes version
`SC/SCD/SCDP`	SC	4/8/16 bytes version

Table A.2: Atomic instructions provided by MIPS64. [Wav16]

Instruction	Primitive	Additional notes
`Rd = memw_locked/memd_locked (Rs)`	LR	4/8 bytes version
`memw_locked/memd_locked (Rs, Pd) = Rtt`	SC	4/8 bytes version

Table A.3: Atomic instructions provided by Hexagon. [Qua16]

Instruction	Primitive	Additional notes
`lock ADC`	AM	ADD with Carry
`lock ADD`	AM	
`lock AND`	AM	
`lock BTC`	AM	Bit Test and Complement
`lock BTR`	AM	Bit Test and Reset
`lock BTS`	AM	Bit Test and Set
`lock CMPXCHG`	CAS	
`lock CMPXCHG16B`	CAS	CAS with 16 bytes
`lock DEC`	AM	Decrement by 1
`lock INC`	AM	Increment by 1
`lock NEG`	AM	Two's Complement Negation
`lock NOT`	AM	One's Complement Negation
`lock OR`	AM	
`lock SBB`	AM	Subtract with Borrow
`lock SUB`	AM	
`lock XADD`	FAM	
`XCHG`	SWAP	Does not require explicit `lock` for atomicity
`lock XOR`	AM	

Table A.4: Atomic instructions provided by amd64. [Adv24b]

Instruction	Primitive	Additional notes
`LDADD`	FAM	1, 4 and 8 byte(s) versions available
`LDCLR`	FAM	Not And; 1, 4, 8 and 16 byte(s) versions available
`LDEOR`	FAM	Xor; 1, 4 and 8 byte(s) versions available
`LDSET`	FAM	Or; 1, 4, 8 and 16 byte(s) versions available
`LDMAX`	FAM	Signed Maximum; 1, 4 and 8 byte(s) versions available
`LDMIN`	FAM	Signed Minimum; 1, 4 and 8 byte(s) versions available
`LDUMAX`	FAM	Unsigned Maximum; 1, 4 and 8 byte(s) versions available
`LDUMIN`	FAM	Unsigned Minimum; 1, 4 and 8 byte(s) versions available
`STADD`	AM	1, 4 and 8 byte(s) versions available
`STCLR`	AM	Not And; 1, 4 and 8 byte(s) versions available
`STEOR`	AM	Xor; 1, 4 and 8 byte(s) versions available
`STSET`	AM	Or; 1, 4, 8 and 16 byte(s) versions available
`STMAX`	AM	Signed Maximum; 1, 4 and 8 byte(s) versions available
`STMIN`	AM	Signed Minimum; 1, 4 and 8 byte(s) versions available
`STUMAX`	AM	Unsigned Maximum; 1, 4 and 8 byte(s) versions available
`STUMIN`	AM	Unsigned Minimum; 1, 4 and 8 byte(s) versions available
`SWP`	SWAP	1, 4, 8 and 16 byte(s) versions available
`CAS`	CAS	1, 4, 8 and 16 byte(s) versions available
`LDXR`	LR	1, 4, 8 and 16 byte(s) versions available
`STXR`	SC	1, 4, 8 and 16 byte(s) versions available

Table A.5: Atomic instructions provided by Aarch64. [Arm21]

Instruction	Primitive	Additional notes
`LWAT/LWAD _,_,0`	FAM	Add
`LWAT/LWAD _,_,1`	FAM	Xor
`LWAT/LWAD _,_,2`	FAM	Or
`LWAT/LWAD _,_,3`	FAM	And
`LWAT/LWAD _,_,4`	FAM	Unsigned Maximum
`LWAT/LWAD _,_,5`	FAM	Signed Maximum
`LWAT/LWAD _,_,6`	FAM	Unsigned Minimum
`LWAT/LWAD _,_,7`	FAM	Signed Minimum
`LWAT/LWAD _,_,8`	SWAP	
`LWAT/LWAD _,_,9`	CAS	Value is written when not equal
`LWAT/LWAD _,_,10`	FAM	Bounded increment
`LWAT/LWAD _,_,11`	FAM	Increment if equal
`LWAT/LWAD _,_,12`	FAM	Bounded decrement
`stwat/stdat _,_,0`	AM	Add
`stwat/stdat _,_,1`	AM	Xor
`stwat/stdat _,_,2`	AM	Or
`stwat/stdat _,_,3`	AM	And
`stwat/stdat _,_,4`	AM	Unsigned Maximum
`stwat/stdat _,_,5`	AM	Signed Maximum
`stwat/stdat _,_,6`	AM	Unsigned Minimum
`stwat/stdat _,_,7`	AM	Signed Minimum
`stwat/stdat RS,RA,8`	AM	Stores `RS` to memory at `RA` and `RA` + 1 if `RA` = `RA` + 1
`ldarx`	LR	1, 2, 4, 8 and 16 byte(s) versions available
`stdcx`	SC	1, 2, 4, 8 and 16 byte(s) versions available

Table A.6: Atomic instructions provided by Power64. The ISA does not have dedicated mnemonics for most atomic operations but rather relies on a *function code* operand. _ is used as an operand placeholder. [Ope21]

APPENDIX B

Sv32 Memory Translation

```
format SV32Address : Word = {
    offset [11..0],
    vpn0 [21..12],
    vpn1 [31..22]
}

format SATPFormat : Word = {
    ppn [21..0],
    asid [30..22],
    mode [31]
}

format SV32PTE : Word = {
    v[0],
    r[1],
    w[2],
    x[3],
    u[4],
    g[5],
    a[6],
    d[7],
    rsw[9..8],
    ppn0[19..10],
    ppn1[31..20],
    ppn = (ppn1, ppn0)
}

[ read full ]
register satp : SATPFormat
```

Listing B.1: Definition of relevant Sv32 data structures.

```
process translate_sv32(va: SV32Address, pte: SV32PTE,
        type: VADL::AccessType) -> (ppn: Address) = {
    let invalidRead = (type = VADL::AccessType::Read)
                        & (pte.r = 0) in
    let invalidWrite = (type = VADL::AccessType::Write)
                        & (pte.w = 0) in
    let invalidExecute = (type = VADL::AccessType::Execute)
                        & (pte.x = 0) in {
        if ((invalidRead | invalidWrite) | invalidExecute) then {
            raise {
                // Raise page-fault exception
            }
        } else {
            ppn := (pte.ppn, va.offset) as Bits<32>
        }
    }
}
```

Listing B.2: Permission check for Sv32 translation.

```
process SV32(addr: SV32Address, type: VADL::AccessType )
    -> (ppn: Address) = {
    // Fetch from page-table
    let pteaddr =
        ((satp.ppn as PA) << 12) + ((addr.vpn1 as PA) << 2) in
    let pte = MEM<4>(pteaddr as Word) as SV32PTE in {
        if((pte.v = 0) | ((pte.r = 0) & (pte.w = 1))) then {
            raise {
                // Raise page-fault exception
            }
        } else if(((pte.x = 0) & (pte.w = 0)) & (pte.r = 0)) then {
            // Leaf entry
            let pte2addr =
                ((pte.ppn as PA) << 12) + ((addr.vpn0 as PA) << 2) in
            let pte2 = MEM<4>(pte2addr as Word) as SV32PTE in
            let invalid = pte2.v = 0 in
            let invalidNoReadButWrite = (pte2.r = 0) & (pte2.w = 1) in
            let invalidNoPermissions =
                ((pte2.r = 0) & (pte2.w = 0)) & (pte2.x = 0) in {
                if((invalid | invalidNoReadButWrite)
                    | invalidNoPermissions) then {
                    raise {
                        // Raise page-fault exception
                    }
                } else {
                    ppn := translate_sv32(addr, pte2, type)
                }
            }
        } else {
            ppn := translate_sv32(addr, pte, type)
        }
    }
}

process VMEM(addr: Address, type: VADL::AccessType)
    -> (ppn: Address) = {
    ppn := if(satp.mode = 0) then addr else SV32(addr, type)
}
```

Listing B.3: General Sv32 translation procedure.

APPENDIX C

VADL Implementation of RV32 A-Extension

```
instruction set architecture RV32IAM extending RV32IM = {
  model AMOtype(name: Id, type: Id, op: Ex, funct7: Bin)
  : IsaDefs = {
    instruction $name : Rtype = {
      let addr = X(rs1) in
      let val = X(rs2) as $type in //First operand
      //Protect region from other threads to ensure atomicity
      lock MEM<4>(addr) in
      let memVal = MEM<4>(addr) as $type in //Second operand
      {
        X(rd) := memVal  //Original value always stored in register
        MEM<4>(addr) := $op  //Apply operation and store back to mem
      }
    }

    encoding $name = {
      funct7 = $funct7,
      funct3 = 0b010,   // 4-byte operation
      opcode = 0b0101111
    }

    assembly $name =
      (mnemonic, ’ ’, register(rs1), ’ ’,
       register(rs2), ’ ’, register(rd))
  }

  $AMOtype(AMOADD ; Word ; memVal + val ; 0b00000’0’0)
  $AMOtype(AMOAND ; Word ; memVal & val ; 0b01100’0’0)
  $AMOtype(AMOOR ; Word ; memVal | val ; 0b01000’0’0)
  $AMOtype(AMOXOR ; Word ; memVal ^ val ; 0b00100’0’0)
  $AMOtype(AMOMAXU ; UnsignedWord ;
      if memVal > val then memVal else val ; 0b11100’0’0)
  $AMOtype(AMOMAX ; SignedWord ;
      if memVal > val then memVal else val ; 0b10100’0’0)
  $AMOtype(AMOMINU ; UnsignedWord ;
      if memVal < val then memVal else val ; 0b11000’0’0)
  $AMOtype(AMOMIN ; SignedWord ;
      if memVal < val then memVal else val ; 0b10000’0’0)
  $AMOtype(AMOSWAP ; Word ; val ; 0b00001’0’0)

  instruction LR : Rtype = {  //Load-Reserved
    let addr = X(rs1) in {
      X(rd) := MEM<4>(addr).loadExclusive //Mark address
    }
  }

  encoding LR = {
    funct7 = 0b00010’0’0,   // No Rel/Ac-semantics
    funct3 = 0b010,         // 4-byte operation
    rs2 = 0,
    opcode = 0b0101111
  }

  assembly LR = (mnemonic, ’ ’, register(rs1), ’ ’, register(rd))

  instruction SC : Rtype = { //Store-Conditional
    let addr = X(rs1) in
    //Here, we need to lock the region in order to ensure
    //that no other thread writes between the ‘isExclusive‘
    //check and the actual write afterwards.
    lock MEM<4>(addr) in
    let excl = MEM(addr).isExclusive in {
      if excl then { //Check whether address is still marked
        MEM<4>(addr) := X(rs2)
        X(rd) := 0 //Indicate success
      } else {
        X(rd) := 1 //Indicate failure
      }
    }
  }

  encoding SC = {
    funct7 = 0b00011’0’0,   // No Rel/Ac-semantics
    funct3 = 0b010,         // 4-byte operation
    opcode = 0b0101111
  }

  assembly SC =
    (mnemonic, ’ ’, register(rs1), ’ ’,
     register(rs2), ’ ’, register(rd))
}
```

Listing C.1: RV32 A-extension implemented in VADL.

APPENDIX D

VADL Benchmark Configuration

```
[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 256 ]
[ blocks = 64 ]
[ nSet = 4 ]
[ attachedTo L2 ]
[ dataCache ]
cache L1D : Address -> Bits<8>

[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 256 ]
[ blocks = 64 ]
[ nSet = 4 ]
[ attachedTo L2 ]
[ instructionCache ] // <-- Comment out to disable instruction cache
cache L1I : Address -> Bits<8>

[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 2048 ]
[ blocks = 64 ]
[ nSet = 8 ]
[ attachedTo MEM ]
cache L2 : Address -> Bits<8>
```

Listing D.1: VADL cache configuration of the `P1-Cache`. Based on the Cortex-X1 [Arm20].

```
[ size = 16 ]
[ attachedTo MEM ]
[ coalescing ]
[ flushPolicy opportunistic ]
[ write buffer ]
logic writebuffer
```

Listing D.2: VADL write buffer configuration of the P1-WB.

```
[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 256 ]
[ blocks = 64 ]
[ nSet = 4 ]
[ attachedTo L2 ]
[ dataCache ]
cache L1D : Address -> Bits<8>

[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 256 ]
[ blocks = 64 ]
[ nSet = 4 ]
[ attachedTo L2 ]
[ instructionCache ] // <-- Comment out to disable instruction cache
cache L1I : Address -> Bits<8>

[ evict leastRecentlyUsed ]
[ prot snooping_wb_mesi ]
[ entries = 2048 ]
[ blocks = 64 ]
[ nSet = 8 ]
[ attachedTo MEM ]
cache L2 : Address -> Bits<8>

[ size = 16 ]
[ attachedTo MEM ]
[ coalescing ]
[ flushPolicy opportunistic ]
[ write buffer ]
logic writebuffer
```

Listing D.3: VADL cache and write buffer configuration of the P1-Cache-WB. Based on the Cortex-X1 [Arm20].

APPENDIX E

Additional Results

Benchmark	MemAcc	Misses	Miss Rate
aha-mont64	4542083	40	0.0009 %
crc32	4029015	9	0.0002 %
cubic	10535005	79906	0.7585 %
edn	3541307	48	0.0014 %
huffbench	2669327	64	0.0024 %
matmult-int	3262543	26	0.0008 %
md5sum	2099758	40	0.0019 %
minver	4919146	140	0.0028 %
nbody	14165624	166	0.0012 %
nettle-aes	4467281	68	0.0015 %
nettle-sha256	4017643	132	0.0033 %
nsichneu	2238692	102435	4.5757 %
picojpeg	4475664	162	0.0036 %
primecount	4297050	9	0.0002 %
qrduino	3353854	216	0.0064 %
sglib-combined	2636565	84	0.0032 %
slre	2659985	63	0.0024 %
st	4432389	165	0.0037 %
statemate	1641312	60	0.0037 %
tarfind	1026459	22	0.0021 %
ud	3401997	51	0.0015 %
wikisort	2850992	232	0.0081 %

Table E.1: Results of tracking L1I memory accesses and cache misses of `P1-Cache`.

Benchmark	MemAcc (VADL)	MemAcc (gem5)	Misses (VADL)	Misses (gem5)	Miss Rate (VADL)	Miss Rate (gem5)	Abs. Error	Rel. Error
aha-mont64	12391	12784	6	-17	0.0484 %	-0.1330 %	0.0846 %	63.60 %
crc32	525512	525095	19	40	0.0036 %	0.0076 %	0.0040 %	52.46 %
cubic	1560507	1559903	163	251	0.0104 %	0.0161 %	0.0057 %	35.09 %
edn	1104210	1017783	71	95	0.0064 %	0.0093 %	0.0029 %	31.11 %
huffbench	720181	703952	158	223	0.0219 %	0.0317 %	0.0097 %	30.75 %
matmult-int	1224448	1224853	179	219	0.0146 %	0.0179 %	0.0033 %	18.24 %
md5sum	269657	271273	57	44	0.0211 %	0.0162 %	0.0049 %	30.32 %
minver	841658	842526	18	27	0.0021 %	0.0032 %	0.0011 %	33.27 %
nbody	1583537	1564099	37	52	0.0024 %	0.0033 %	0.0010 %	28.88 %
nettle-aes	892582	881113	183	296	0.0205 %	0.0336 %	0.0131 %	38.97 %
nettle-sha256	567490	556593	17	18	0.0030 %	0.0032 %	0.0002 %	7.37 %
nsichneu	1230808	1230362	6	-13	0.0005 %	-0.0011 %	0.0006 %	53.83 %
picojpeg	1294104	1226815	54	92	0.0042 %	0.0075 %	0.0033 %	44.36 %
primecount	870929	871322	7	-3	0.0008 %	-0.0003 %	0.0005 %	133.44 %
qrduino	782871	687524	34	44	0.0043 %	0.0064 %	0.0021 %	32.14 %
sglib-combined	974347	976109	95	165	0.0098 %	0.0169 %	0.0072 %	42.32 %
slre	869280	869794	15	17	0.0017 %	0.0020 %	0.0002 %	11.71 %
st	484861	482464	46	79	0.0095 %	0.0164 %	0.0069 %	42.06 %
statemate	886417	863478	11	12	0.0012 %	0.0014 %	0.0002 %	10.71 %
tarfind	255512	249555	146	172	0.0571 %	0.0689 %	0.0118 %	17.10 %
ud	665695	658851	19	20	0.0029 %	0.0030 %	0.0002 %	5.98 %
wikisort	661441	662798	193	348	0.0292 %	0.0525 %	0.0233 %	44.43 %
Mean	-	-	-	-	-	-	0.0008 %	36.73 %

Table E.2: Results of tracking L1D memory accesses and cache misses of `P1-Cache` and `gem5 Timing+Cache`. For the latter, the setup and teardown memory operations are not accounted for and hence, some cache misses might yield negative values.

Benchmark	L1 Data Cache	L1 Instruction Cache
aha-mont64	0 %	12 %
crc32	1.6 %	12 %
cubic	1.5 %	18 %
edn	2.2 %	12 %
huffbench	3.1 %	11 %
matmult-int	4.1 %	11 %
md5sum	1.5 %	11 %
minver	2.0 %	12 %
nbody	1.2 %	13 %
nettle-aes	2.7 %	12 %
nettle-sha256	1.6 %	13 %
nsichneu	4.9 %	29 %
picojpeg	3.1 %	12 %
primecount	2.2 %	12 %
qrduino	2.3 %	11 %
sglib-combined	3.8 %	11 %
slre	3.4 %	13 %
st	1.1 %	12 %
statemate	4.4 %	11 %
tarfind	2.9 %	10 %
ud	2.2 %	11 %
wikisort	2.7 %	12 %

Table E.3: Evaluation of `P1-Cache` in terms of how much reading/writing from/to the L1 instruction and data caches consumes compared to the total execution. `perf` was used for measuring.

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ABI application binary interface. 2, 7, 11

AM atomic modify. 36, 37, 40, 84–86

ASID address space identifier. 28, 47

AST abstract syntax tree. 43

CAS cycle accurate simulator. 4, 5, 8, 31, 43–45, 51, 57, 59, 70–73, 77, 100

CAS compare-and-swap. 2, 4, 23–25, 36, 40, 42, 62, 72, 74, 81, 84–86, 99

DSE design space exploration. ix, xi, 3, 4, 30, 33

FAA fetch-and-add. 23, 24, 37–39, 105

FAM fetch-and-modify. 2, 36, 37, 40, 83–86

FAS fetch-and-sub. 38

FSM finite state-machine. 16, 17, 51, 52, 100

HTM hardware transactional memory. 46

IPG instruction progress graph. 44

ISA instruction set architecture. xi, 4, 7–10, 19, 20, 22, 24, 29, 30, 33–37, 40, 41, 46, 47, 49, 57, 59, 62, 63, 76, 78, 79, 81, 83, 86, 99, 101, 105

ISS instruction set simulator. 4, 5, 8, 30, 33, 34, 43, 45, 51, 69, 71, 74, 75, 81, 100

LFU least-frequently used. 14

LKMM Linux Kernel Memory Model. 20

LLC last-level cache. 50

LR load-reserved. 24, 25, 36, 40–43, 46, 58, 81, 83–86, 99, 105

LRU least-recently used. 14, 49

MiA microarchitecture. 7, 8, 12, 25, 30, 31, 33, 41, 42, 44, 48, 49, 57, 63, 69, 78, 79, 81

MMU memory management unit. 3, 27

NUMA non-uniform memory access. 16

PDL processor description language. 2, 5, 7, 33, 34

PPN physical page number. 27, 47

PTE page-table entry. 26, 27, 99

RAII Resource acquisition is initialization. 42, 43

RMW read-modify-write. 2, 23, 24, 42, 81

RTL register-transfer level. 33, 79

RVWMO RISC-V Weak Memory Ordering. 67

SC sequential consistency. 20, 21, 67

SC store-conditional. 24, 25, 36, 40–43, 58, 81, 83–86, 99, 105

SSA static single assignment. 8

TLB translation lookaside buffer. 3, 28, 32, 47, 74, 79

TSO total store order. 2, 20–22, 67

VADL Vienna Architecture Description Language. ix, xi, 2–5, 7–12, 31, 35–49, 51, 56–58, 62, 63, 66, 67, 69–71, 74–79, 81, 92–94, 96, 105, 106, 108

VIR VADL intermediate representation. 5, 8, 43, 44, 63

VPN virtual page number. 27

Bibliography

[ABC+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290384.

[Adv24a] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, March 2024.

[Adv24b] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual - Volume 3: General-Purpose and System Instructions*, March 2024.

[ALOJ13] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. McSimA+: A Manycore Simulator with Application-Level+ Simulation and Detailed Microarchitecture Modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85, 2013. doi:10.1109/ISPASS.2013.6557148.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2), jul 2014. doi:10.1145/2627752.

[app] Porting just-in-time compilers to Apple silicon. https://developer.apple.com/documentation/apple-silicon/porting-just-in-time-compilers-to-apple-silicon. Accessed: 2024-04-15.

[ARB+05] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC Architecture Description Language and Tools. *Int. J. Parallel Program.*, 33(5):453–484, October 2005. doi:10.1007/s10766-005-7301-0.

[Arm20] Arm Limited. *Arm® Cortex®-X1 Core - Technical Reference Manual*, 5 2020.

[Arm21] Arm Limited. *Arm®v8-M Architecture Reference Manual*, September 2021.

[AVD+15] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. SiNUCA: A Validated Micro-Architecture Simulator. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 605–610, 2015. doi:10.1109/HPCC-CSS-ICESS.2015.166.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. doi:10.1145/2024716.2024718.

[BKP20] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. A Survey of Cache Simulators. *ACM Comput. Surv.*, 53(1), feb 2020. doi:10.1145/3372393.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1454115.1454128.

[cac] Cachegrind: a high-precision tracing profiler. https://valgrind.org/docs/manual/cg-manual.html#cg-manual.cg_diff. Accessed: 2024-09-17.

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2063384.2063454.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):224–234, April 1991. doi:10.1145/106974.106995.

[ea] Chris Barton et al. The Multi2Sim Simulation Framework. http://www.multi2sim.org/downloads/m2s-guide-4.2.pdf. Accessed: 2024-09-17.

[EH] Jan Edler and Mark D. Hill. Dinvero IV. https://pages.cs.wisc.edu/~markhill/DineroIV/. Accessed: 2024-09-23.

[FVPF95] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 503–507, 1995. doi:10.1109/EDTC.1995.470354.

[Gra21] Alexander Graf. Compiler Backend Generation using the VADL Processor Description Language. Master's thesis, Vienna University of Technology, 2021. doi:10.34726/hss.2021.79221.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991. doi:10.1145/114005.102808.

[HGG+99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 485–490, 1999. doi:10.1109/DATE.1999.761170.

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, page 299–302, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/266021.266108.

[HHH+24] Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Niklas Mischkulnig, Philipp Paulweber, Tobias Schwarzinger, and Andreas Krall. The Vienna Architecture Description Language, 2024. arXiv:2402.09087.

[HK23] Christoph Hochrainer and Andreas Krall. A pred-LL (*) Parsable Typed Higher-Order Macro System for Architecture Description Languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 29–41, 2023. doi:10.1145/3624007.3624052.

[HSLS20] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming, Second Edition.* Elsevier, January 2020.

[IEE18] mmap - map pages of memory. Standard, IEEE and The Open Group, 2018.

[int] Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors. https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html. Accessed: 2024-07-17.

[Int24] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, 4 2024.

[ISO18] Information technology — Programming languages — C. Standard, International Organization for Standardization, Geneva, CH, July 2018.

[JCLJ08] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. CMP $ im : A Pin-Based OnThe-Fly Multi-Core Cache Simulator. 2008. URL: http://www.jaleels.org/ajaleel/publications/cmpsim_mobs2008.pdf.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1815961.1815971.

[KTJ10] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. Sampling Dead Block Prediction for Last-Level Caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010. doi:10.1109/MICRO.2010.24.

[LAA+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jer'onimo Castrill'on, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago M"uck, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and 'Eder F. Zulian. The gem5 Simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020. arXiv:2007.03152.

[Lam79] Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065010.1065034.

[LLG$^+$90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 148–159, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/325164.325132.

[LP24] Jason Lowe-Power. Classic Memory System coherence. https://www.gem5.org/documentation/general_docs/memory_system/classic-coherence-protocol/, 2024. Accessed: 2024-09-27.

[Mar84] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. In *21st Design Automation Conference Proceedings*, pages 587–593, 1984. doi:10.1109/DAC.1984.1585857.

[MD08] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[MHAM11] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Litmus Tests for Comparing Memory Consistency Models: How Long Do They Need to Be? In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 504–509, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2024724.2024842.

[MHW03] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token Coherence: Decoupling Performance and Correctness. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 182–193, 2003. doi:10.1109/ISCA.2003.1206999.

[Mih23] Hristo Mihaylov. Optimised Processor Simulation with VADL. Master's thesis, Vienna University of Technology, 2023. doi:10.34726/hss.2023.102629.

[MKK$^+$10] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010. doi:10.1109/HPCA.2010.5416635.

[MRSM16] M.S.Hrishikesh, Madhusudhan Rajagopalan, Sujatha Sriram, and Rashmin Mantri. System Validation at ARM: Enabling our Partners to Build Better Systems. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/System%20IP/System_Validation_at_ARM_Enabling_our_partners_to_build_better_systems.pdf?revision=a82b4e93-4118-4a3a-ba5f-70e38f6b4616&hash=88D9B7CF58BE13B124E43EE538D21F4D, April 2016. Accessed: 2024-09-27.

[mul] MultiCacheSim: A coherent multiprocessor cache simulator. https://github.com/blucia0a/MultiCacheSim. Accessed: 2024-09-23.

[NSH+20] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd edition, 2020.

[Ope21] OpenPOWER Foundation. *Power ISA - Version 3.1B*, September 2021.

[PH17] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.

[PSZ21] Philipp Paulweber, Georg Simhandl, and Uwe Zdun. Specifying with Interface and Trait Abstractions in Abstract State Machines: A Controlled Experiment. *ACM Trans. Softw. Eng. Methodol.*, 30(4), jul 2021. doi:10.1145/3450968.

[pyc] pycachesim - A single-core cache hierarchy simulator written in python. https://github.com/RRZE-HPC/pycachesim. Accessed: 2024-09-23.

[Qua16] Qualcomm Technologies, Inc. *Hexagon V60/V61 Programmer's Reference Manual*, March 2016.

[RIS19] RISC-V International. *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*, December 2019.

[RIS21] RISC-V International. *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*, December 2021.

[RLC+12] Pengju Ren, Mieszko Lis, Myong Hyon Cho, Keun Sup Shim, Christopher W. Fletcher, Omer Khan, Nanning Zheng, and Srinivas Devadas. HORNET: A Cycle-Level Multicore Simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):890–903, 2012. doi:10.1109/TCAD.2012.2184760.

[Sch20] Hermann Schützenhöfer. Cycle-Accurate Simulator Generator for the VADL Processor Description Language. Master's thesis, Vienna University of Technology, 2020. doi:10.34726/hss.2021.78460.

[Sch22] Tobias Schwarzinger. Flexible Generation of Low-Level Developer Tools with VADL. Master's thesis, Vienna University of Technology, 2022. doi:10.34726/hss.2023.103246.

[SK13] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485922.2485963.

[TA13] A.S. Tanenbaum and T. Austin. *Structured Computer Organization*. Pearson, 2013.

[UJM+12] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 335–344, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2370816.2370865.

[Wav16] Wave Computing, Inc. *MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual*, June 2016.

[WJH+11] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 430–441, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2155620.2155671.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 24–36, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/223982.223990.

[ZCCJ+23] Niko Zurstraßen, José Cubero-Cascante, Jan Moritz Joseph, Li Yichao, Xie Xinghua, and Rainer Leupers. par-gem5: Parallelizing gem5's Atomic Mode. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023. doi:10.23919/DATE56975.2023.10137178.