Characterization and Modeling of Atomic Memory Operations in Arm Based Architectures

Víctor Soria Pardos

Director: Adrià Armejach (UPC, BSC-CNS)
Co-director: Darío Suárez (UNIZAR)
Responsible: Miquel Moretó (UPC)

Master in Innovation and Research in Informatics: High Performance Computing

facultat d'informàtica de barcelona (fib)

Universitat Politècnica de Catalunya
(UPC) BarcelonaTech

January 2022

Abstract

Efficient fine-grain synchronization is a classic computer architecture challenge that has been profusely addressed in the past. Load Link and Store Conditional (LL/SC) became one of the few solutions to this problem and today it is still part of the State-of-the-art. However, as the core count keeps growing many Instruction Set Architectures (ISA) start to support other synchronization instructions that scale better like Atomic Memory Operations (AMO). In this work we present a characterization of LL/SC and AMO instructions in two current Arm-based server machines.

Furthermore, Arm has released its Network-on-Chip (NoC) specification enabling different hardware implementations of how AMO are executed in a multicore. Since the adoption of this new standard is still in its first stages, we have modeled six different AMO policies to explore the hardware design trade offs. We find out that there is no single implementation that outperforms the rest. Therefore, we have designed a hardware solution to dynamically select the best configuration obtaining up to 1.15x speed-ups on relevant benchmarks from the Splash-3 benchmark suite.

Acknowledgements

First of all, I would like to thank Adrià Armejach, Darío Suárez and Miquel Moretó, for their support, patience and guidance. Thanks to all my coworkers for solved doubts, their helpful tools and lunch talks. I have to also thank my Friends for being there through thick and thin.

Finally, I would like to thank and dedicate this work to my parents for their support and love.

The project has been carried out within the Computer Sciences - High Performance Domain-Specific Architectures group of the BSC in collaboration with the Computer Architecture group of the Universidad of Zaragoza (gaZ).

Contents

Chapter 1

Introduction

This chapter describes the historic evolution of semiconductor chips paying special attention to the emergence of multicore architectures. We explain the main challenges of current processors when facing multicore synchronization. Next, the main contributions of this thesis are summarized. Finally, we outline the organization of this document.

1.1 Context

During the last 40 years, the microchip industry has experimented an spectacular progress in microchip manufacturing. Every two years the top microchip manufacturers have been able to halve the transistor size, following what is known as the Moore's Law [34]. These advances not only improved transistor size, but also decreased their power consumption and their switching time. Thus, new chips incremented the switching frequency increasing performance [12]. However, this trend finished a few years ago because of the power density reached by the new manufacturing processes [9]. This effect was named as the "power-wall", which imposes challenges on heat dissipation, precluding the adoption of higher frequency chips.

Despite industry hitting the power-wall, Moore's Law continued delivering. However, instead of increasing frequency, new chips incorporated more computing elements to exploit those extra available transistors. In the High Performance Computing (HPC) area, which solves scientific and engineering problems with high computing demands, this extra computing elements translated into Central Processing Units (CPU) with a higher core count and internal memory capacity. Nevertheless, parallel computing is more complex to program than

sequential computing. Apart from breaking the problem into independent parts or tasks to be executed concurrently, parallel computing requires program correctness. Frequently, independent tasks have dependencies that must be respected. Thus, parallel systems feature synchronization mechanisms to orchestrate those tasks. These dependencies make exploiting parallelism hard because they limit the maximum achieved speed-up. Furthermore, parallel computing is limited by the portion of code that is paralelizable, as stated by Amdahl's law [2].

Besides, as the amount of cores integrated in a chip increases, the interconnection delay to communicate them becomes a bottleneck. Whilst transistor switching time improves on every new process technology, wires have kept a constant delay and only small decreases have been achieved thanks to new metal layers and design partitioning [7]. Short distance wires have a controllable delay, but interconnections between different spread components have an exorbitant cost, taking several cycles to transverse these distances [7]. These limiting factors, have driven designer efforts to create new scalable interconnection designs based on Networks-on-Chip (NoC) [8]. The main idea behind NoC is that each component in the system has its own router to send and receive messages. Rather than using long and slow wires, connections are split using shorter links connected through routers. Consequently, systems based on NoC support higher frequencies, have higher bandwidth and are more scalable.

As the adoption of NoC spreads, new possible topologies are proposed to connect the different components in a system. A plethora of topologies exist, being the most popular the ring, the mesh, the crossbar, the hypercube and the torus. The common approach when designing a NoC chip is to distribute components in the network by placing one or several cores at every NoC crosspoint, forming a cluster. Each core is connected to one private first level of memory cache (L1). All the L1's are connected to a cluster-shared second level of memory cache (L2). Finally, it is common to place one slice of the shared third level of cache (L3) close to each cluster. Not all the NoC crosspoints have this structure, because there are other kinds of components that need to be placed on the chip, such as memory controllers, IO devices, DMAs, etc. Figure 1.1 depicts such a solution for a torus interconnection.

Another two fields that have experimented changes with the adoption

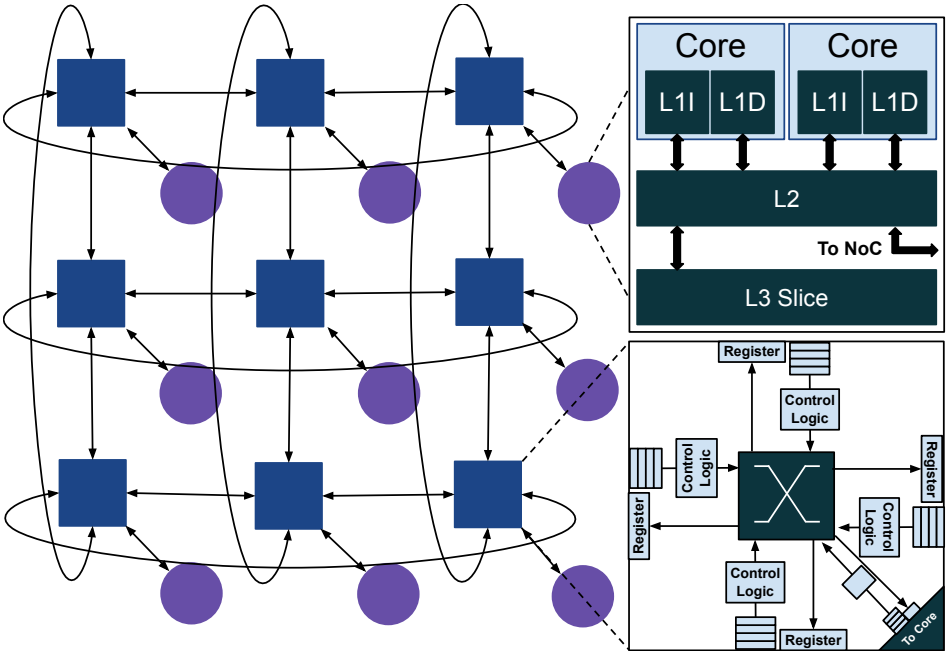

Figure 1.1: Torus 18-core diagram. Each crosspoint hosts a slice of shared L3 cache, private L2 and L1 caches and two cores.

of NoCs are Consistency and Cache Coherence. The former specifies how the cores see the memory updates, while the latter guarantees that each core reads the most recent version of the value stored in a given address. In snoop based protocols, this two conditions are satisfied broadcasting writes to all caches. In directory based protocols, all request are sent to the directory which orders the memory requests and selectively sends snoops to caches. NoCs have eased the transition from snoop-based to directory-based cache coherence protocols. The directory is now a distributed structure spread among the L3 slices. Thus, the centralization of memory request is no more a bottleneck. Now two different request can be processed in parallel in different directory slices. Moreover, NoCs typically use different physical channels to send requests, responses, snoops and data, enabling a higher message level parallelism between the two endpoints of a link.

As we have seen, multicore architecture evolves with new technologies. In this work, we aim to revisit the current foundations of parallel synchronization relaying on the most recent developments.

1.2 Contributions

This thesis presents three contributions in the field of parallel synchronization. First, it analyzes the current state-of-the-art of parallel synchronization instructions doing a characterization. We limit the scope of this characterization to Arm-based systems for simplicity and availability. Second, we perform a design space exploration of modern parallel synchronization directives based on Atomic Memory Operations (AMOs) and evaluate them. We find that there is no single implementation that suits all cases. Third, based on this observation, we propose a mechanism to dynamically switch between implementations.

1.2.1 Characterization of Synchronization Instructions on Arm-Based Architectures

Parallel synchronization received a lot of attention in the parallel computing community back in the 80's and 90's [10] [18]. Nevertheless, as solutions were developed and integrated into final products, the interest on this topic has been relegated to academic discussion (parallel synchronization history is reviewed in Chapter 2). However, as the core-count of chips increases and new technologies like NoCs are adopted, synchronization is again starting to gain relevance.

Our first contribution consists on characterizing the behavior of available synchronization instructions on different Arm-based systems. The goal of this characterization is to find which synchronization instructions work better in different situations and what are their bottlenecks. A secondary goal of these experiments is to gather insights for the design and verification of a model that supports Atomic Memory Operations (AMOs) used for the second contribution. This characterization uses different experiments that include microkernels and real world applications. We use two different machines to compare possible different implementations, the Kunpeng 920 and the Graviton 2.

In our experiments, we find that both machines behave different because of their specific implementations of LL/SC and AMOs. In the case of Kunpeng 920, AMOs have a higher cost than LL/SC. This higher cost makes that HPC applications scale better with LL/SC than with AMOs. However, in those scenarios with high contention AMOs are more efficient than LL/SC, despite their higher cost. In the case of Graviton 2, we find out that AMOs perform always better than LL/SC because of these pair of instructions frequently suffers from transient live locks. In the case of HPC applications we see how AMOs can speed-up 1.25× the execution time.

1.2.2 Modeling of Atomic Memory Operations in Arm-based systems

In last five years, Arm-based systems have expanded their market view to new sectors such as desktop (Apple's M1), server (AWS's Graviton 2) and HPC (Fujitsu's A64FX). As part of this expansion strategy, Arm has released an open standard for building large cache coherent systems based on NoC architectures, named Advanced Microcontroller Bus Architecture 5 Coherent Hub Interface (AMBA 5 CHI). This new standard has been used as the foundation of recent released many-core systems such as Graviton 2, Ampere Altra and Graviton 3.

One of the key features introduced in this new NoC architecture (with respect to the previous AMBA specs) is the capability of performing remote or far Atomic Memory Operations. In several architectures, AMOs are performed only in the first level of cache next to the core. But in AMBA 5 CHI, AMOs can be sent inside a message to the cache coherence directory to be performed there to reduce pollution and further invalidations. We have followed AMBA 5 CHI specs to develop six different cache policies that execute AMOs in six different ways. Then, we have evaluated all six implementation using the same workloads of the characterization. In our experiments, we find out that each application has a different pattern access that fits different policies. Thus, we cannot use a single policy to obtain the highest performance in all applications.

1.2.3 Dynamic Atomic Memory Operations Policy

The scenario opened by the second contribution gives us the opportunity to develop a hardware mechanism that is able to adapt to each application to select the best fitting AMO policy. In this third contribution we compute what is the achievable speed-up using an ideal policy. Next, we describe the design choices we have made to create an AMO policy predictor and what insights have motivated this design. Then, we perform a sensitivity study of the parameters of the predictor. Finally, we evaluate our predictor in the same conditions as the static policies. With our predictor we are able to some benchmarks significantly.

1.3 Thesis Organization

This document is structured in seven chapters that cover the following topics:

- **Chapter 1** presents the context in which this thesis has been made. It also depicts briefly the contributions made in this work.
- **Chapter 2** presents the challenges of parallel synchronization and summarizes the current solutions that have been used to tackle synchronization. It also describes some of the academic and industry proposals to improve current designs.
- **Chapter 3** describes the methodology, tools and procedures used to perform the experiments presented in the thesis. It also explains the modeling environment used to develop our proposals.
- **Chapter 4** presents the performed characterization to two current commercial multicore systems, focusing on the synchronization directives.
- **Chapter 5** describes how we have modeled AMO in gem5, a well-known architectural simulator. It also explains the design space exploration we have performed to find AMO policies within AMBA 5 CHI specs. Finally, it evaluates the model using the same benchmarks used to characterize the real machines.

- **Chapter 6** presents our hardware mechanism to select dynamically the best AMO policy. It contains an evaluation in which the dynamic mechanism is compared against static policies.
- **Chapter 7** closes this thesis summing up all the contributions and publications.

Chapter 2

State of the Art

This chapter describes the main problems architects have to face when designing multicore systems. Then the most common solutions adopted by the industry to implement synchronization are explained. Finally, the chapter summarizes some solutions proposed in prior academic works.

2.1 Problem Statement

With the adoption of multicores two major types of architectures arised due to process intercommunication requirement. On one hand, Shared Memory systems have multiple processors reading and writing to the same shared data. On the other hand, Distributed Shared Memory (DSM) systems use processors that have its own local memory and send messages to communicate with other processors. Shared Memory multicores are the most common type of single-chip multicores, because Shared Memory offers a simple, fast and efficient method of communication. DSM systems with hundreds of processors employ Message Passing to communicate across nodes. The Message Passing Interface (MPI) [36] is the preferred standard to implement Message Passing.

Even though Shared Memory systems can provide efficient communication, the correct use of these systems demands deep knowledge by the programmer on different synchronization techniques. Synchronization is used in modern multicore systems to orchestrate how different threads execute in parallel (i.e. forcing a Critical Section (CS) in which only one thread executes a part of the code, while other cores execute other parts or wait [20]). There are many synchronization directives that have been developed through the years, such as locks,

barriers, semaphores and transactions, among many others.

All these synchronization directives require special support implemented in hardware, usually through specific instructions. Most of these instructions implement atomic Read-Modify-Write (RMW) operations that read a memory location, perform an operation with the read data and, finally write again the memory position with the result in an indivisible way. As we will see in Section 2.5, numerous hardware synchronization mechanisms have been proposed throughout the years, but only a few of them have been adopted by real systems. The most common solutions implemented are Load Link and Store Conditional (See Section 2.2), Atomic Memory Operations (See Section 2.3), and Transactional Memory (See Section 2.4).

2.2 Load Link and Store Conditional

In the past, the number of cores integrated on a single chip was low enough to adopt simple solutions that did not required big changes in the architecture. This is the case of Load-Link and Store-Conditional (LL/SC), a pair of instructions that together achieve atomic RMW semantics. First, the programmer uses a Load-Link (LL) to get the value that will be modified atomically. This instruction not only returns the current value of a memory location, but also flags the memory position as accessed by a LL. Then, the programmer can modify the read value as needed. Finally, to update the memory position a Store-Conditional (SC) is issued. This SC only succeeds if the flag left by the LL has not been removed. To ensure that no other core updates that memory position, the system must clean the LL flag on every store performed.

LL/SC have been adopted by many Instruction Set Architectures (ISA) like MIPS, Power, Arm and RISC-V. The most common implementation of LL/SC is based on cache coherence invalidation protocols. When a write operation is performed by a core, the cache coherence protocol sends invalidations to other caches that hold the cache line to gain the exclusive access to it. Therefore, when a LL brings the block to the cache in a Shared state, if any other core writes to the same cache line, then the cache line will be invalidated. Thus, on a SC the cache line will no longer be present and it will fail. Other implementations, instead of issuing the invalidations when

the SC is executed, issue the invalidations when the LL is executed. This simplifies the execution of the SC because the block is already in exclusive state on the L1 cache. However, this approach can generate live-locks when several threads issue LLs that invalidate each other's cache lines.

Arm defines in its architecture that the LL/SC handling mechanism must be carried by Exclusive Monitors [22]. Each core has a local monitor that is associated with it. The local monitor can be constructed to hold the exclusive state for a particular address. Any exclusive store is treated as if it matches the address of the previous exclusive load. Thus, only one LL/SC pair can be handle at a time and other memory accesses are usually forbidden or restricted in number.

As can be derived from the previous example, LL/SC do not guarantee always forward progress for all cores, but for one of them. For example, when two cores want to increment a shared counter, both would emit LL, perform an addition and emit a SC. But only a single core can increment the shared counter, while the others fail.

One of the key advantages of LL/SC is that performing the computation associated with the RMW operation has free cost in hardware, because the Arithmetic Logic Unit (ALU) of the core can be reused. It also allows programmers to perform different sequences of integer/float operations. However, as the computation between a LL and a SC increases, the probability of failure for the SC also increases. Moreover, as the core count increases the performance of LL/SC can degrade as we will see in Chapter 4.

2.3 Atomic Memory Operations

Atomic Memory Operations (AMOs) are instructions that directly encode a basic operation that is performed with RMW semantics. The main difference with LL/SC is that the operation is done in an indivisible way. Some AMOs such as a swap (exchanges two values) or fetch-and-add (increment the value of a memory position) guarantee forward progress of all threads or cores that execute those instructions. Following the example seen in the previous subsection, two threads that want to increment a shared counter would emit a fetch-and-add instruction and both would succeed.

AMOs are widely adopted by most ISAs such as Power, Arm, RISC-V or x86. Analogously to LL/SC, AMOs can be implemented in such a way that, they reuse the hardware that already exist on a core. During the operation the core blocks the cache line, so invalidations are deferred until the operation completes. This is the approach Intel took to implement its instruction XCHG. This implementation, although correct and intuitive, sacrifices performance because of the complex control required between the core and the cache.

Another well known approach is to first obtain the cache block in Modified (M) state and then perform the operation inside the first Level of Cache. This requires an specific ALU and pipeline to perform AMOs. This solution can be found in the Lowrisc L1 cache [33].

A more complex solution consist on executing AMOs on other components of the chip, such as a memory controller or a shared cache bank [18] [21] [14]. This remote execution of an AMO is known as Far AMO. The idea is interesting when several cores access a specific cache line and the block "ping-pongs" from one core to another. Thus, instead of bringing the data to the computation, the computation is sent where the data resides.

Far AMOs were implemented in commercial projects like the Cray T3D [24], T3E [40], and SGI Origin [27], in which Far AMOs were implemented at the memory controllers. TilePro64 [43] and recent GPUs [46] implemented Far AMOs in shared caches. While, this idea was deeply explored in the past, it has not been used in mainstream processors until recently. Power9 has adopted Far AMOs at the memory controllers, while Arm supports different solutions in its Advanced Microcontroller Bus Architecture (AMBA) 5 Coherent Hub Interface (CHI) specs.

Despite AMOs were designed to reduce the latency and NoC traffic, they still cause significant global traffic since operations are sent to a shared, fixed location. Serialization can be another problem because each Far AMOs has both read and write semantics (they return the latest value of the memory position they update). Therefore, consistency needs to be preserved issuing the AMOs only at commit (like a store) and waiting until the value returns (like a load). This is the reason why ISAs like RISC-V, Arm and Power9 have included atomic-no-return instructions, in which the previous data to the update is not fetch to the

core. These instructions reduce the cost of AMOs, because cores are able to commit AMOs earlier due to the weak consistency memory model.

2.4 Transactional Memory

Both LL/SC and AMOs are widely used to implement high level directives like mutexes, spinlocks and barriers among others. However, programming parallel applications using those directives can be complex and requires experienced programmers. Transactional Memory (TM) [19] aims to address the need for a simpler parallel programming model. TM promises good parallel performance and easy-to-write parallel code. With TM, programmers simply demarcate sections of code (called transactions) where synchronization occurs. Then the Hardware Transactional Memory (HTM) system executes those transactions guaranteeing the following properties: atomicity, isolation, and serialisability.

To provide atomicity, the HTM system ensures that transactions are executed under all-or-nothing semantics, either all the code in a transaction is executed or none of it. Isolation is provided by ensuring that no partial results are visible to the rest of the system, results are made visible only when a transaction completes its execution successfully. Finally, serialisability requires the execution order of concurrent transactions to be equivalent to some sequential execution order of the same transactions. To guarantee this properties all TM systems need to perform two important tasks: conflict detection and version management [6].

To detect conflicts, each transaction tracks the memory accesses into two different sets the read-set and the write-set. Then, when the transaction is committing read and write sets are compared to detect fine-grain read-write and write-write conflicts. If a conflict is found, one of the conflicting transactions has to be aborted, the execution state is then rolled back to the point where the transaction started, and the transaction is retried. Otherwise, if no conflicts are found, the transaction commits successfully.

2.5 Related Work

Academic literature has profusely explored hardware and software techniques that reduce the cost of updates to shared data. Apart from the aforementioned solutions, there are other ideas that were not widely implemented in real systems that are related with the topic that we will cover in this section.

One of the first proposals made in 1995 is Dynamic Self-Invalidation (DSI)[28]. DSI is a technique that tries to eliminate invalidation messages by automatically invalidating private cache line copies. This must be done before a conflicting access by another processor triggers an invalidation message. The directory is the component that identifies which blocks should use self-invalidation by maintaining a history of its sharing pattern. When servicing a request for a cache block, the directory uses an extra bit to signal if the block is likely to be invalidated in the future. Then, the self-invalidation is triggered in two scenarios: when a synchronization instruction is executed or when the tag of the cache line is evicted from a FIFO queue. The main weakness of this idea is the mechanism that triggers the auto-invalidation, because the FIFO queues need to be sized properly for each workload and synchronization instructions can evict blocks needed in the next cycles.

Active Memory Operations [50] explore the idea of fine grained updates of AMOs. Unlike traditional Cache Coherent writes that require to invalidate sharers to achieve one private copy with read-and-write permission, the Active Memory Operations can modify the data without obtaining exclusive state. To maintain coherence the updates are sent to the directory which will forward the update to all the sharers with a local copy. This approach substitutes invalidations by update messages and reduces the writer-consumer latency. Although the approach is tested in some microbenchmarks, the impact of the optimization is unclear for real world applications.

Similarly, Sharing/Timing Adaptive Push [35] (STAP), is a complex dynamic mechanism that preemptively sends data from producers to consumers to minimize critical path communication latency. To do so, it detects three types of data sharing patterns: Producer-Consumer, Broadcast and Migratory Exclusive Ownership (a.k.a. cache-line

ping-pong effect). These patterns are detected using additional and expensive hardware at the L1 and L3 caches, and performing extra communication between these two structures. The main advantage is that updates can be forwarded directly to consumers.

Even though NoCs have been adopted by the industry, core to core communication is transparent to programmers. A notable exception is the SW26010 (the Sunway TaihuLight processor [15]), which exposes the inter-core network to developers for better architecture scalability. pLock [42] is a fast lock designed for architectures that support Explicit inter-core Message Passing (EMP). pLock proposes two techniques: chaining lock, and hierarchical lock, both to reduce message count and mitigate network congestion. The chaining lock when a thread releases a lock instead of freeing it, the thread passes the lock to the next waiting client. The hierarchical lock removes slow long-distance communication using intermediate cores as local servers to avoid long-distance communication. Both ideas combined can reduce the amount of messages and the average latency of messages. However, instead of using cache coherent caches, EMP needs scratchpad caches, which are very complex to program.

MiSAR [29], is a minimalistic synchronization accelerator that is added to each L3 slice. The accelerator is designed to support the classic Pthread directives Mutex Lock/Unlock, Barriers and Conditional Variables through custom instructions. For example a *LOCK* instruction is always sent to the corresponding accelerator. In case the lock is free it returns a free message to the requester. If the lock is not available, the accelerator simply delays the response until the lock is freed. This prevents the requesting core's *LOCK* instruction from being committed, stalling its core until the lock is obtained. The biggest weakness apart from introducing extra instructions on the ISA, is that MiSAR assumes that there is no thread oversuscription or any other process running on the system and thus it can stall cores from committing instructions.

Another high level directive that is targeted in Carbon [25] is task scheduling. Carbon introduces hardware distributed task queues that implement task stealing. In this scheme, each thread has its own queue, where it enqueues a tasks. When the thread finishes executing a task and needs a new task to execute, it first looks at its own queue. When there is

no task available in its own queue, it steals a task from one of the other queues.

Multi-Address atomic operations (MAD atomics) [17] are a set of individual instructions that achieve complexity-effective, non-speculative, non-deadlocking, fine-grained locking for multiple addresses. These instructions target the Dijkstra philosophers problem of taking several locks, giving a predefined order of acquiring the locks to avoid deadlocks based on the address of the variables.

Research has recently turn towards exploiting the capabilities of RMW updates. For example, COUP [49] presents an aggressive reordering of AMOs that exploit the commutativity of this type of operations. However, this approach requires support for Floating Point AMOs. RICH [13], presents a similar approach that targets OpenMP reductions, where the runtime is supposed to delimit which code blocks contain reduction functions, so the hardware is able to optimize them using a hierarchical reduction module present in the different levels of cache.

Chapter 3

Experimental Framework

This Chapter describes the methodology used to carry out our experiments and the workloads used as representative code of real world applications. It includes a short description of the machines we have used in the experiments. Moreover, we describe the simulation infrastructure used to model the different implementations of AMO and a discussion on the microarchitecture that is being simulated.

3.1 Methodology and Environment

In order to run our native and simulated experiments, we have developed a methodology that enables fair comparisons between machines and experiments. In this section, we will describe how we have set-up and run the experiments following that methodology. Moreover, to ensure the software stack and OS environment does not introduce any artifact, we have followed a rigorous set-up of the software environment, which is described in the following subsection. Finally, we close this section listing and describing the tools used in the characterization to understand the behaviour of some applications.

3.1.1 Methodology

The first step to fairly execute a benchmark natively or on a simulated machine is to define a Region-of-Interest (ROI). The ROI delimits which part of the application is going to be captured by the time measuring tools, tracing mechanisms or profiling tools. This way ROIs help us to characterize exactly the execution patterns of applications. Moreover, a good definition of the ROI is fundamental when we running an experiment

in a simulated machine because the cost of simulating extra cycles is very expensive. This is specially important for full system simulations, in which the system calls can generate noise in the results. Therefore, we have excluded from the ROI all the file system calls, data initialization and thread creation and destruction.

For the characterization of commercial multicore machines we have developed an automatic framework to carry out the experiments. The purpose of this framework is to launch all the experiments, pin the threads to physical cores and repeat the experiments several times. Thus, the results of the experiments are isolated from the noise of other processes running on the same system. Using the standard deviation obtained from the experiments we have fixed the amount of repetitions to 10 times. In the case of simulations we have conducted only one repetition due to simulation time restrictions.

Finally, in order to ensure the same conditions for all the experiments we have used the same binaries in all the experiments. This requirement is essential to avoid possible differences on the libraries available in the machines. Further details can be found in the next section.

3.1.2 Software Environment

Since our test machines have different Kernel versions, OS distributions, library versions, and compilers installed, we have used a common environment to compile our applications. This environment is the same our simulator uses to run the applications along with the kernel and file system. We use Kernel 4.15.0 and an Ubuntu image of version 16.04.9.

To compile our applications we have used the Arm HPC compiler (version 20.1), a commercial compiler developed by Arm. We have opted for this compiler instead of GCC or CLANG based on previous results that demonstrate that the Arm HPC Compiler can generate well optimized code [41]. We also tested the benchmarks to verify that this premise is true.

For every benchmark we have compiled two binaries that contain two different set of instructions. On one hand, we have binaries that use exclusively LL/SC to implement locks, barriers, or atomic updates. On the other hand, we have binaries that only use AMOs for the same

LL/SC	-march=armv8-a+nolse -pthread -static
AMO	-march=armv8-a+lse -lpthread -L/lib/aarch64-linux-gnu/atomics -static

Table 3.1: Compiler flags used to obtain LL/SC and AMO binaries

primitives. In the case of libraries like POSIX, by default they are compiled to support only LL/SC. Therefore, we have recompiled those libraries forcing the use of AMOs. The flags we have used to get these binaries are listed on Table 3.1.

To verify that the generated code is free from instructions of the wrong type, in each binary we have used a postprocessing over the binary to disassembly it and then count the instructions of each type. We have also checked manually the assembly of some primitives like the CAS (see Listings 3.1 and 3.2).

```
#0: ldr  x1, [x4]      # Load Lock
    cbnz x1, #0        # Lock != Zero
#1: ldxr x2, [x4]      # Load Lock
    eor  x3, x2, x1    # Lock unchanged
    cbnz x3, #3        #
    stxr w3, x5, [x4] # Write 1 in Lock
    cbnz w3, #1        # STX != Fail
#3: cbnz x2, #0        # LL = 0
```

Figure 3.1: CAS implemented with LL/SC

```
#0: ldr  x1, [x4]      # Load Lock
    cbnz x1, #0        # Lock != Zero
#1: mov  x2, x1        # x10 = expected
    cas  x2, x5, [x4] # CAS Lock by 1
    cbnz x2, #0        # CAS != Fail
```

Figure 3.2: CAS implemented with AMO

Listing 3.1 shows the implementation of a CAS using only LL/SC instructions. Initially the code waits until the Lock variable is equal to 0 (lock is not busy) using a load and a conditional branch. When the Lock is released by other thread writing a 0 on it the CAS builtin tries to capture the lock. First, the thread reads with a LL (ldxr) the lock value, then performs an Exclusive OR and a conditional branch to check that the value did not change during the process. In case of a change, the function comes back to waiting until the value is zero again. Otherwise, the CAS builtin tries to write a 1 using a SC (stxr) instruction. Finally, the builtin checks that the SC did not fail and that the read value was zero with two conditional branches.

Listing 3.2 shows the same builtin using AMO instructions. Again, the

code starts waiting until the lock is released. Once the lock is free, the thread performs a CAS instruction, but first it needs to set the expected value in the destination register. This register is overwritten by the original value of the Lock before issuing the CAS. Thus, after issuing a CAS, the thread checks if the destination register contains a 0 that means that it succeeded.

3.1.3 Tools

During our experiments, we have used several applications for different purposes, one of them is thread pinning. *Taskset* [30] is a simple Linux command that pins threads to physical cores. This is essential in our case because our test machines have multiple sockets that are visible to the OS, but we want to test only on a single chip.

Another tool we have used is *time* [31]. This tool apart from returning the execution time, performs a break down of the amount of CPU time spent in kernel mode, the percentage of the CPU used, the amount of memory used and the number of swap out of the process and number of page faults. This tool has been used as a sanity check, not as a time measuring tool.

When performing a characterization just capturing the execution time and scalability of applications is insufficient to understand and obtain useful insights. Therefore, we have used *extrae* [11] tracing tool to dig into the behavior of applications. This tool creates trace files from the execution events captured during the execution. In this case we have used POSIX events for our traces. In order to visualize the traces we have used the trace visualization tool *paraver* [26]. We have used *paraver* also to visualize our custom made traces generated from simulator events.

3.2 Workloads

Workloads are a major concern when performing experiments, because the use of non representative applications can lead researchers to wrong conclusions. Therefore, we have selected the workloads following this criteria:

- Benchmarks should represent parallel and sequential patterns present in common and HPC applications (such as consumer-producer, group sync, etc).
- Benchmarks should allow a comparison of different synchronization directives implemented across multiple systems.
- Benchmarks should present different scalability patterns to study the effects of AMOs.

The best way accomplish these requirements was to use different benchmarks suites. This way, we decided to use one suite of microkernels named LockHammer [23] and Splash-3 [39] a popular parallel application suite.

3.2.1 LockHammer

LockHammer [23] is a performance evaluation tool for locks, barriers and read-write locks, which can be used to characterize the performance of high core-count systems or compare different synchronization directives. Several basic primitives and well known lock implementations are included in the suite. Table 3.2 list the synchronization directives we have used in our experiments. This list includes a short description of the implemented directives we have tested.

The general structure of the micro-benchmark consists of a for loop that all threads execute in parallel. In the loop each thread tries to capture the lock, once acquired the thread executes a variable number of NOP instructions that represent the Critical Section (CS) of a program and then releases the lock. After releasing the lock the thread can execute again a parameterized number of NOP instructions, which represent the parallel section (PS). Thus, we can model different SC/PS ratios tweaking the number of NOP operations executed inside and outside the critical section. Note that the amount of CS increases with the number of threads instantiated, therefore this is a weak scaling application.

We have selected two configurations to simulate two different high contention scenarios. The first configuration named *serialized* because each thread executes 500 NOP instructions in the CS, and 0 NOP

Synchronization Directive	Abbreviation	Description
Empty	Empty	Null implementation of lock that simulates no synchronization cost
CAS Lockref	CAS-Lock	Basic spinlock implementation that uses CAS instruction with 64 bit word
CAS RW Lock	CAS-RW	Basic multiple reader lock that uses CAS instruction to increment lock
Incdec Refcount	IncDec-RW	Shared incrementable counter
Swap Mutex	Swap-Lock	Basic spinlock implementation that uses SWAP instruction with 64 bit word
JVM Monitor	JVM-Lock	Java virtual machine implementation of a monitor for shared objects. Initial phase tries to acquire the lock with a Spin within a maximum number of iterations. In case of failure, the thread adds itself in a list of waiting threads and stops on a pthread conditional variable
Hybrid Spinlock	HS-Lock	Behaves like a normal spinlock at first, in case of contended lock uses a back-off strategy. Similar to Queued Spinlock from Linux. The MCS Lock is used to provide fairness, since is equivalent to FIFO queue
Hybrid Spinlock Fastdequeue	HSF-Lock	Same as as Hybrid Spinlock with special fast path for thread that acquire the lock after waiting
Optimistic Spin Queue Lock	OSQ-Lock	MCS like lock that uses spin relax instead of sleeping
Queued Spinlock	Queued-Lock	Extracted from Linux 4.13 implementation. Is a complex version of the MCS Lock with several slowpaths
Ticket Spinlock	Ticket-Lock	Linux 4.13 spinlock that uses a shared counter to implement fair CS access
Event Mutex	MySQL-Lock	MySQL 5.7 implementation of mutex. Uses CAS instruction to obtain the lock, an uses delays as back-off method
TBB Spin RW Mutex	TBB-RW	Threading Building Block Reader-Writer lock, that is fast, unfair, with back-off and writer-preference

Table 3.2: Lock description of LockHammer Micro-benchmark Suite

instructions in the PS. Therefore, all the work is done in the CS and all the work needs to be serialized (in the case of RW locks this is not true). With this configuration, on each thread acquisition, all threads in the system will fight to access the CS. However, one problem of this scenario is that the latency between L1 and L2 is not 0. Therefore, threads that release a lock will keep lock cache line in M state in the L1. Since the PS is 0 instructions and invalidations need some cycles to arrive from L2 to L1, threads that release a lock are able to gain the CS again. This happens more frequently in locks that have pauses or sleep directives.

There are some lock implementations that constantly read the variable to check if has been updated, like in spinlocks and RW locks. In these locks the problem is attenuate, because a thread can receive an invalidation just after gaining ownership of a cache line. So when releasing the lock, it might lose the ownership of the block.

Therefore, we have designed a second configuration that tries to avoid this problem. We named this configuration *unserialized* because it uses 250 NOP instructions in the CS and PS. Therefore, half of the work is now split in a parallel region, which is short enough to have high contention when using high number of threads. If we revisit the previous problem we see now that a thread releasing a lock, needs to execute 250 NOP instructions before trying to acquire again the lock. This creates a window of time big enough to wake up a sleeping thread and send a lock acquire that invalidates the cache line.

3.2.2 Splash-3

Splash-2 [47] is a benchmark suite that comprehends different scientific applications. It was one of the first shared-address-space multiprocessing benchmarks that were publicly available. Soon, it became a reference for parallel application for two decades. Recently, an updated version (Splash-3 [39] has ported this benchmark suite to a modern C-language memory model and without data races. In Table 3.3 we can see all the applications present in this suite, a short description of them and the check list of synchronization directives used (barriers, mutexes or condition variables). By default Splash-3 uses Mutex, Semaphore and Condition Variables of the POSIX thread (pthread) library.

Application	Description	Barriers	Locks	Pauses
BARNES	N-body method	*	*	*
CHOLESKY	Sparse matrix factorization	*	*	*
FFT	1-D Fast Fourier Transform	*		
FMM	Hierarchical N-body method	*	*	*
LU-C	Dense matrix factorization with blocking	*		
LU-NC	Dense matrix factorization	*		
OCEAN-C	Optimized large-scale ocean movement partial differential equation	*		
OCEAN-NC	Large-scale ocean movement partial differential equation	*		
RADIOSITIY	Finite Element Method for scene rendering	*	*	
RAYTRACE	3D raytracing scene rendering	*	*	
VOLREND	3D raycasting scene rendering	*	*	
WATER-NQ	Water molecule force computation using $O(n^2)$ algorithm	*	*	
WATER-SP	Water molecule force computation using $O(n)$ algorithm	*	*	

Table 3.3: Application description and Synchronization directive list of the Splash-3 Benchmark Suite

Splash-3 features different predefined inputs to be executed in different scenarios. One of these default inputs consist on a big workload mean to be executed on multicores with big core counts named NATIVE. However, these inputs are not suitable for simulations since the CPU time that they consume can be enormous. Therefore, we have use one of the pre-defined inputs, which is several times smaller and it was mean to be used in simulators.

When performing our simulations we have observed that the use of Mutex locks from POSIX library can introduce variations in the experiments due to the variable sleep times. Two experiments with the same configuration, but some small changes in the kernel execution environment can lead to completely different execution times. This happens because we use a small computation time that is heavily affected by kernel interruptions. In order to reduce this variability we have decided to use POSIX spinlocks to implement locks in the simulations.

3.3 Tests Machines

To characterize the current synchronization instructions present in Arm based multicores, we have selected a group of machines that is representative of Server and HPC ecosystems: Huawei's Kunpeng 920 and AWS's Graviton 2.

3.3.1 Kunpeng 920

Huawei's Kunpeng 920-4826 is a 64-bit Armv8.2 server microprocessor developed by HiSilicon on 7nm. The multicore features 48 TaiShan v110 cores, a 4-way OoO core based on the Cortex-A72 that runs at 2.6 GHz. The chips uses a multi-die architecture to integrate such a big number of cores. Each die uses an internal ring NoC to interconnect cores [48] (see Figure 3.3 for the 64-core version diagram). Table 3.4 lists the cache hierarchy of the Kunpeng 920.

Kunpeng 920 supports Atomic Memory Operations included in the LSE extension of Armv8.2. Architectural details on how these instructions are supported are not disclosed.

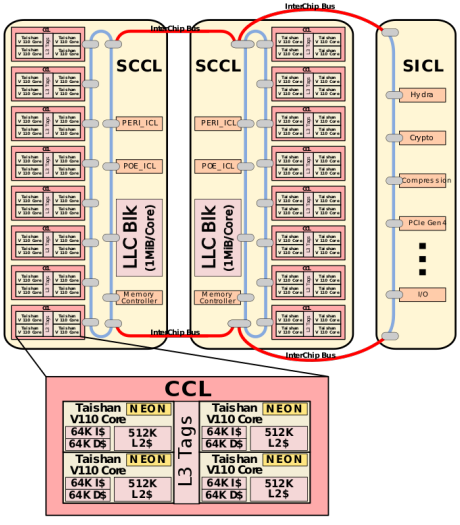

Figure 3.3: Kunpeng 920 64-core mesh and die diagram [44].

3.3.2 Graviton 2

Graviton 2 is a custom multicore developed by Annapurna Labs exclusively for Amazon. Currently these multicores are used to power EC2 instances for Amazon Web Services. Graviton 2 is a 64-bit Armv8.2 multicore SoC that features 64 custom Neoverse N1 cores running at 2.5GHz. Neoverse cores are identical to the A76 architecture, featuring Out-of-Order(OoO or O3) execution with 11 stages, 4-way decode and 8-way issue. The 7nm multicore integrates these 64 cores through a CMN-600 mesh interconnection, which implements AMBA 5 Coherent Hub Interconnection (CHI) architecture at 2.0GHz (see Figure 3.4). Table 3.5 lists the cache hierarchy parameters of the Graviton 2.

Neoverse N1 cores support Large System Extension (LSE) as part of the Armv8.2 specs, which introduces AMOs in Armv8 ISA. As we have seen in section 2.3, atomic instructions to cacheable memory can be performed as either near atomics or far atomics, depending on where the cache line containing the data resides. The Neoverse N1 manual states:

- When an instruction hits in the L1 data cache in a unique state, then

L1 Instruction Cache	Size	64KiB
	Block Size	64-bytes
L1 Data Cache	Size	64KiB
	Block size	64-bytes
	Type	Private
L2 Cache	Size	512KiB
	Associativity	8-way
	Type	Private
L3 Cache	Size	48x1MiB
	Type	Shared and Sliced

Table 3.4: Kunpeng 920 cache hierarchy [48].

L1 Instruction Cache	Size	64KiB
	Associativity	4-way
	Block Size	64-bytes
L1 Data Cache	Size	64KiB
	Associativity	4-way
	Block Size	64-bytes
	Type	Private
	Latency	4 cycles
L2 Cache	Size	1MiB
	Associativity	8-way
	Block Size	64-bytes
	Type	Private
	Policy	Inclusive (only L1D) MESI
	Latency	9/11 cycles
L3 Cache	Size	32MiB
	Associativity	16-way
	Type	Shared and Sliced

Table 3.5: Graviton 2 cache hierarchy [3].

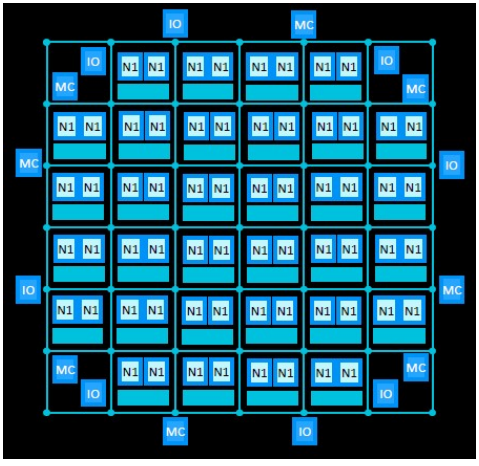

Figure 3.4: Graviton 2 64-core mesh diagram [5]. N1 boxes represent cores

it is performed as a near atomic in the L1 memory system. If the atomic operation misses in the L1 cache, or the line is shared with another core, then the atomic is sent as a far atomic on the core CHI interface.

- If the operation misses everywhere within the cluster, and the interconnect supports far atomics, then the atomic is passed on to the interconnect to perform the operation. When the operation hits anywhere inside the cluster, or when an interconnect does not support atomics, the L3 memory system performs the atomic operation. If the line it is not already there, it allocates the line into the L3 cache. This depends on whether the directory is configured with an L3 cache.

Alternatively, the Neoverse N1 manual specifies that the CPUECTLR system register can be programmed such that all atomic instructions execute as a near atomic. However, the programming of this register is restricted to firmware privilege level.

3.4 The gem5 Simulator

In order to model AMOs, we have selected gem5 [32] as our reference simulator. gem5 is one of the most popular cycle-accurate simulators used in computer architecture research. This simulation infrastructure allows researchers to model modern computer hardware systems in detail. One of the main features is its full-system mode that is capable of booting unmodified Linux-based Operating Systems (OS) and run full applications for multiple architectures including Armv8+.

However, the default gem5 package is not meant to simulate accurately cache hierarchies, cache coherence or NoCs. Therefore, gem5 features the Ruby cache model that specifically simulates these components. Ruby uses a domain-specific language that enables new definitions of coherence protocols. Recently Arm has published its CHI protocol for Ruby, partially based on the AMBA 5 CHI specification. This protocol supports LL/SC operations but does not support AMOs in any form. In Chapter 5 we will explain how we have modified gem5 in order to support AMOs in compliance with the AMBA 5 CHI protocol.

gem5 uses several configuration files to specify all the components that will be simulated. In this section, we will describe how we have configured all these parameters for our simulations.

3.4.1 CPU Microarchitecture

In gem5, the basic building block is the CPU. There are four CPU models with different levels of detail and performance. For this work we have used the detailed model of an out-of-order CPU (O3). This model has several configuration files that can be tweak to model different architectures. In our case we have configured the O3 model with the same parameters the Arm Cortex X1 [45], a high performance core with wide and complex datapath that enhances single thread performance. Since we target next generation cores, we have introduce some modifications influenced by the recent Apple M1 [4]. We have increased the amount of resources of internal structures such as Reorder Buffer, Load-Store Queue, etc. These parameters are listed in Table 3.6

Pipeline Widths	Fetch	8-way
	Decode	8-way
	Issue	14-way
	Writeback	14-way
	Commit	8-way
Structure size	Instruction Window	472 entries
	Reorder Buffer	630 entries
	Load-queue	154 entries
	Store-queue	106 entries
Branch Predictor	Type of predictor	Tournament
	Size	8192 entries
	BTB size	8192 entries
	RAS size	64 entries
Frequency	2.5 GHz	

Table 3.6: X1 gem5 configuration.

3.4.2 Cache Hierarchy and NoC Configuration

The architecture of Graviton 2 has inspired the way we have configured our multicore. The configuration file instantiates 64 cores with three levels of cache. The NoC layout is a square mesh of 8x8 crosspoints that runs at 2GHz. The latency of routing a packet and traversing a link is for both cases one cycle. Figure 3.5 depicts the actual shape of the mesh.

In each crosspoint there are two X1 cores, each connected to three private caches (Instruction L1, Data L1 and L2) and one slice of the shared distributed cache (L3) that works as a directory. The details of the caches are listed in Table 3.7. The L2 is fully inclusive with respect to the L1 and total amount memory is 64 MiB (1MiB per core). Meanwhile, the L3 is exclusive with a total of 64 MiB of memory (also 1 MiB per core). We have placed one prefetcher on each private cache based on some small tests done with the STREAM benchmark [37]. We have choosen High Bandwidth Memory 3 (HBM3) as the main memory technology. The chip has 8 channels of 64GB/s that sum up a total memory bandwidth of 512GB/s.

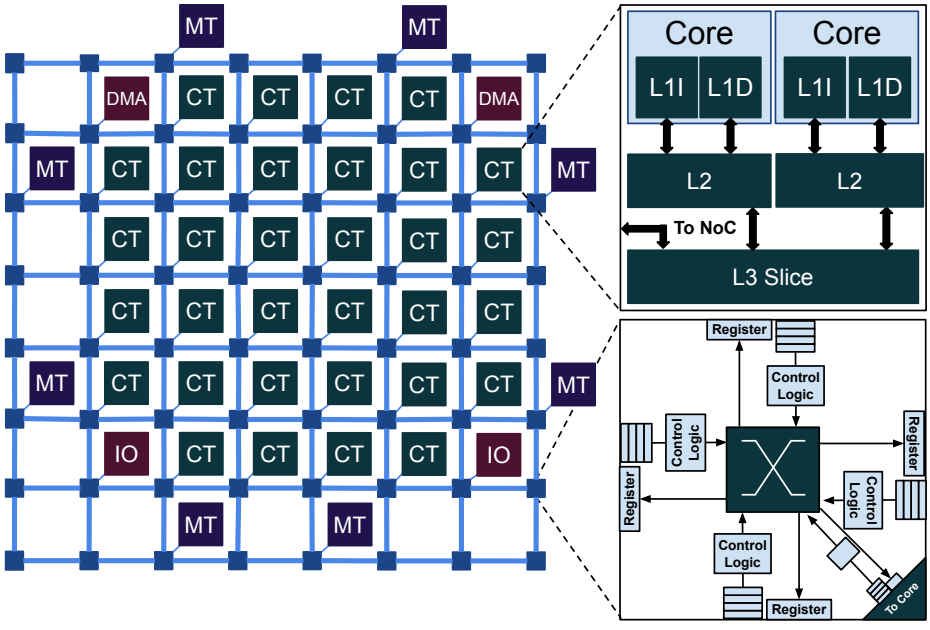

Figure 3.5: gem5 64-core mesh diagram. Boxes represent: Compute Tile (CT), Memory Tile (MT), Input Output device (IO), Direct Memory Access (DMA)

Thread Count	1 - 8 - 16 - 32 - 64 threads
Private Caches	L1I: 64KiB, 4-way, 2 cycles access L1I Prefetcher: Next Line L1D: 64KiB, 4-way, 3 cycles L1D Prefetcher: Strided L2: 1MiB, 8-way, 10 cycles L2 Prefetcher: Best Offset Prefetcher
LLC and HNF slice	2MiB, 8-way, 40-70 cycles Cache Coherency: MOESI Exclusive
NoC	8x8 mesh with 64 cores 2GHz
Main Memory	8 channels of 64GB/s each

Table 3.7: X1 Cache Hierarchy configuration.

Chapter 4

Characterization of Synchronization Primitives

This chapter presents and analyzes the results of the experiments performed in the tests machines. We employ the experimental infrastructure and the methodology explained in Section 3.1.1. First, we evaluate the LockHammer microbenchmark in which multiple locks and Read-Writer locks are tested. Then, we run the Splash-3 benchmark suite and analyze the scalability of the applications.

4.1 LockHammer Characterization

In this section we present and analyze the results obtained executing the LockHammer microbenchmark on the Kunpeng 920 and the Graviton 2. We present three case studies for each machine.

In the first one, we plot the single thread execution time of all the lock implementations. The goal behind this experiment is to find out what is the latency of LL/SC and AMO instructions. Since kernels are run with a single thread pinned to a core, the memory block that contains the lock will always be placed in the L1 of that core. Thus, the execution time will be determined by how fast LL/SC or AMO instructions are executed. We use the *Empty* kernel, which does not implement any synchronization directive, to measure the overhead of each directive.

In the second one, the benchmark is executed with *serialized* configuration. As explained in section 3.2.1, this configuration executes each kernel maximizing the amount of cores that fight for capturing the lock and removes the parallel execution. This way the shared lock is highly contended. To run this experiment we have used one thread per

core (48 threads in Kunpeng 920 and 64 threads in Graviton 2).

Finally, we execute a third scenario in which we use *unserialized* configuration (see section 3.2.1 for further details). In this configuration the size of the parallel region and the critical section (CS) is equal. In this configuration the contention of the locks is lower, but we avoid that a single thread could access several times in a row the critical section.

4.1.1 Kunpeng 920

First, we obtained the absolute execution time all the lock implementations using only one thread (see Figure 4.1). The lowest execution time corresponds to Empty microkernel, in which locks are removed to check what is the ideal execution time. We have placed an horizontal line that corresponds the execution time of *Empty* microbenchmark. Thus, we can see what is the overhead of each lock with respect to the ideal execution with no synchronization instructions. In most LL/SC experiments, we see that the overhead is minimal except for the OSQ and MySQL locks. However, in 8 out of 12 kernels the overhead of AMO instructions is non negligible. We can conclude from these results that the AMOs have a higher delay than the LL/SC in Kunpeng 920.

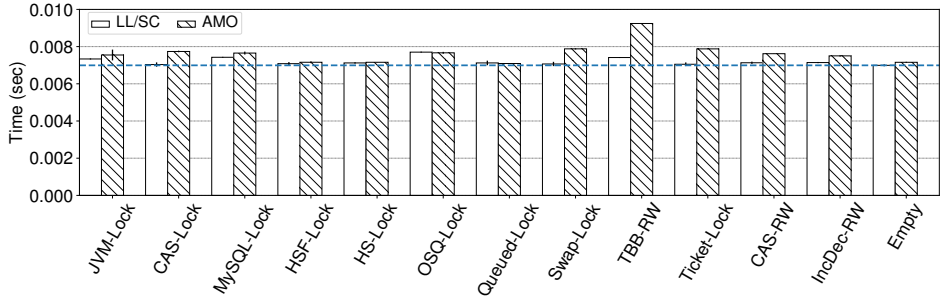

Figure 4.1: Absolute execution time of LockHammer locks on Kunpeng 920 with 1 thread

Next, we obtained the absolute execution time all the lock implementations using one thread per core (48 cores) and *serialized* configuration (see Figure 4.2). We execute each benchmark 10 times and represent the standard deviation using a vertical line on the top of each

bar. Locks are sorted by the best execution time among LL/SC and AMOs versions. Thus, we know what is the best implementation of a lock for this specific machine. For Read-Write locks we can conclude that CAS-RW is the best implementation, while for classic locks JVM-Lock is the fastest. We can see that in both locks using AMOs speeds-up the execution of the benchmark. While 6 locks obtain speed-up when using AMOs, there are 4 locks (Ticket-Lock, HS-Lock, HSF-Lock and Queued-Lock) that are faster using LL/SC rather than AMOs.

Regarding the variability of the experiments we can see that AMO have lower variability than LL/SC instructions. This observation is noticeable in most RW locks (TBB-RW, CAS-RW, IncDec-RW) and some regular locks (MySQL-Lock, JVM-Lock). This characteristic may be of interest in Real Time systems to implement predictable synchronization mechanisms.

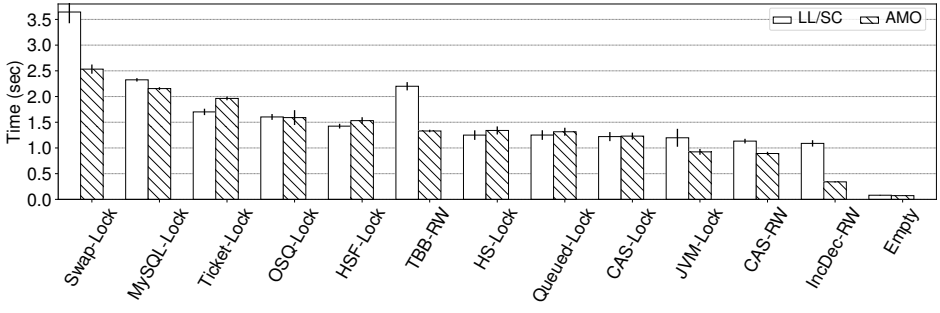

Figure 4.2: Absolute execution time of LockHammer locks on Kunpeng 920 with 48 threads using *serialized* configuration

The performance difference between LL/SC and AMOs is different in every lock implementation. In order to measure how big is the difference between versions we have plotted Figure 4.3. This figure shows the relative speed-up of AMO versions normalized to LL/SC version using again 48 threads. We have sorted each lock implementation from the lowest to the highest speed-up. The highest speed-up is achieved when we implement a shared counter using AMOs, reaching a 3.2×. Next, Intel TBB fair RW-lock implementation achieves a 1.67×; however, CAS-RW is faster in absolute execution time. Despite that both Swap-Lock and MySQL-Lock obtain more than 1.15×, these lock implementations are the slowest

compared to the rest. Finally, JVM-Lock has speed-ups of 1.3×, which make it the fastest lock implementation.

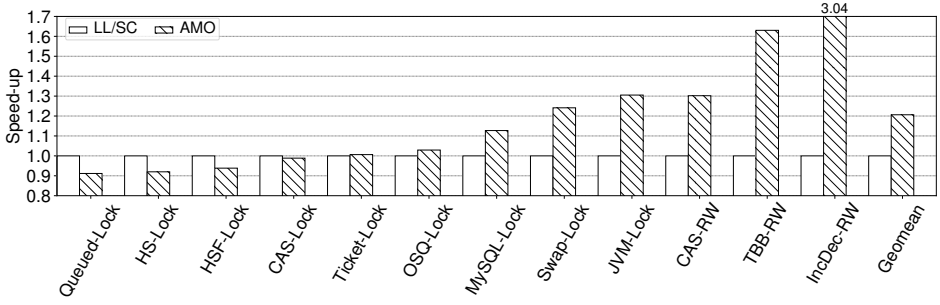

Figure 4.3: Speed-up of LockHammer locks (w.r.t LL/SC version) on Kunpeng 920 with 48 threads using *serialized* configuration

Regarding slowdowns, all the Linux locks except OSQ-lock obtain between 0.95× and 0.86× slowdowns. Despite these locks are not the fastest implementations, they play an important role in the software stack. Therefore, these slowdowns can affect many applications that use the POSIX library. All these Linux locks use sleeps when they fail to acquire a lock after a number of tries. This way they enable a context switch of other threads that may free the requested lock. This also avoids pooling accesses to the mutex address that produce noise in the cache hierarchy. Hence, if the catch function is enhanced to execute faster, the lock can put to sleep the thread faster because it consumes faster its attempts.

Finally, we execute our third set of experiments using one thread per core and the *unserialized* configuration (see Figure 4.4). In this case we have kept the axis size and the ordering of the benchmarks we have used in Figure 4.2. This way we can see how the execution time changes from *serialized* to *unserialized* configurations. Since the latter uses a CS that has half the workload, we observe a reduction in execution time for many benchmarks: Swap-Lock (only for LL/SC version), OSQ-Lock, HSF-Lock and TBB-RW (only for LL/SC version).

Nonetheless, three kernels increase their execution time because the thread releasing the lock cannot take advantage of having the memory block

that contains the lock in the L1 to acquire it again. MySQL-Lock (specially in LL/SC version) and CAS-Lock are two of these locks that experience the slowdown. Both kernels have a fast path that tries to catch the lock with a CAS, while those that failed in the fast path are waiting a signal from the thread that releases the lock. Other lock implementations are not affected by this because they use a data structure to hold the threads that want to access the CS (like OSQ, HSF, HS, Queued or JVM). So, while the releasing thread updates the structure receives an invalidation. The third kernel that increases its execution time is Ticket-Lock. This kernel enforces that fairness using a FIFO order ticket. The increase of the execution time in this kernels is caused by the fight between the thread that is releasing the lock and other previous threads that are modifying the lock to get a new ticket.

With *unserialized* configuration, HSF-Lock is the fastest lock implementation because AMO version is faster than in *serialized*. But, the overall speed-up of AMOs over LL/SC is still 1.2×.

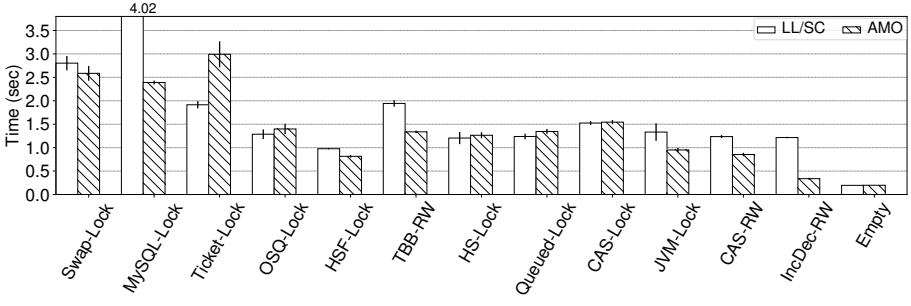

Figure 4.4: Absolute execution time of LockHammer locks on Kunpeng 920 with 48 threads using *unserialized* configuration

Summarizing, AMOs obtain on average a 1.2× speed-up with respect to LL/SC in both *serialized* and *unserialized* experiments with 48 threads. Whilst, AMO seems to outperform LL/SC for most locks, AMOs have a higher latency.

4.1.2 Graviton 2

We perform the same experiments on the Graviton 2. We start with the absolute execution time of the different lock implementations with only one thread (see Figure 4.5). This time, we see how the trends reverse and LL/SC are more expensive than AMOs. However, the differences between these two versions are smaller. Moreover, Graviton 2 executes the experiments almost 1.2× faster with respect to Kunpeng 920.

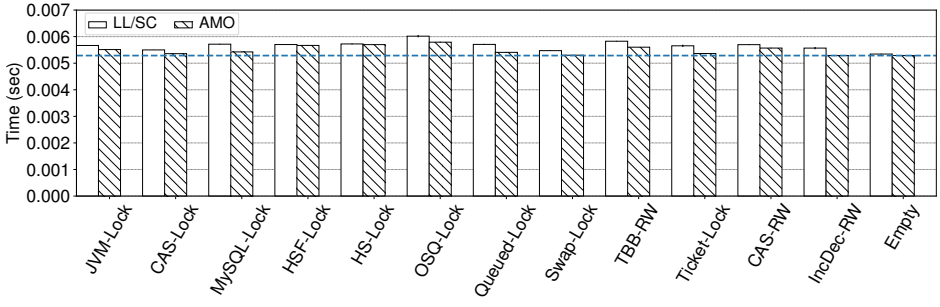

Figure 4.5: Absolute execution time of LockHammer locks on Graviton 2 with 1 thread

Figure 4.6 shows the absolute execution time of LockHammer benchmark using one thread per core with configuration one. Note that this time the number of threads used is 64. Again, locks are sorted using the order used in Figure 4.2. We can see that the results change dramatically from those observed in the Kunpeng 920. However, the fastest implementation for locks and RW-lock are the same (JVM-Lock and CAS-RW).

MySQL-Lock stands out due to the big execution time obtained with LL/SC version. This big slowdown is caused by transient live-locks caused by the higher amount of threads plus the overhead of traversing the NoC. This is common in systems in which the access time to the directory is big enough to receive an invalidation between the LL and the SC. Furthermore, we see that OSQ-Lock, TBB-RW, CAS-RW and IncDec-RW suffer the live locks for LL/SC versions. An AMO is faster than LL/SC in 7 out of 12 kernels (1 more than in Kunpeng 920), while in 3 benchmarks is slower than LL/SC (1 less than in Kunpeng 920). Another interesting insight is

that, despite the Graviton 2 executes these experiments with 64 threads, there are 6 kernels that using AMOs finish in less than one second. In Kunpeng 920 with 48 threads, only three kernels were below one second. Therefore, even that LL/SC may suffer from live-locks, Graviton AMOs are faster than any of the versions in Kunpeng 920.

Regarding variability, we see again that LL/SC in Graviton 2 have higher variability than in Kunpeng 920, and much higher than that of AMOs. The case of OSQ-Lock is notably bad, reaching 30% of the total execution time.

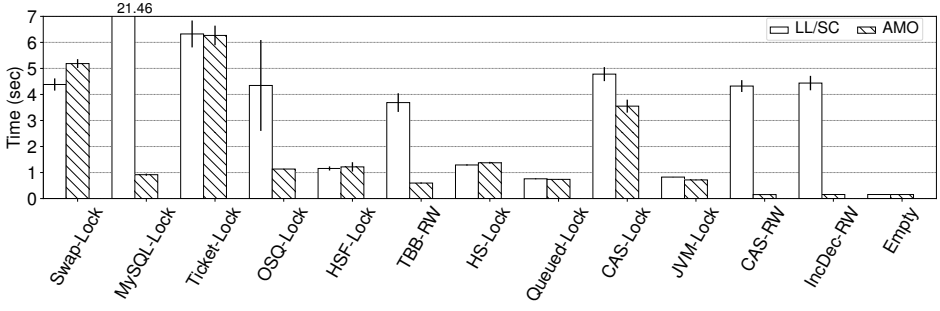

Figure 4.6: Absolute execution time of LockHammer locks on Graviton 2 with 64 threads using *serialized* configuration

Again, we have computed the speed-up of the AMOs with respect to the LL/SC for each kernel (See Figure 4.7). In these case, the order followed to sort the lock implementations is the same as in Figure 4.3. The AMO version of MySQL-Lock, CAS-RW, TBB-RW and IncDec-RW outperforms the LL/SC version because of the live-locks. As a consequence of these speed-ups, the AMOs are 2.1× faster than LL/SC on average. Not everything is caused by the drop of performance of LL/SC version. In Kunpeng 920 some locks experimented some slowdowns when replacing LL/SC by AMOs (Queued-Lock, HS-Lock and HSF-Lock), but in Graviton 2 the slowdowns are less than a 2%. Therefore, we can say that in Graviton 2, AMOs perform always better or equal to LL/SC, except for Swap-Lock.

Finally, we reproduce the previous experiments with *unserialized* configuration (see Figure 4.8). The first thing we notice is that some of

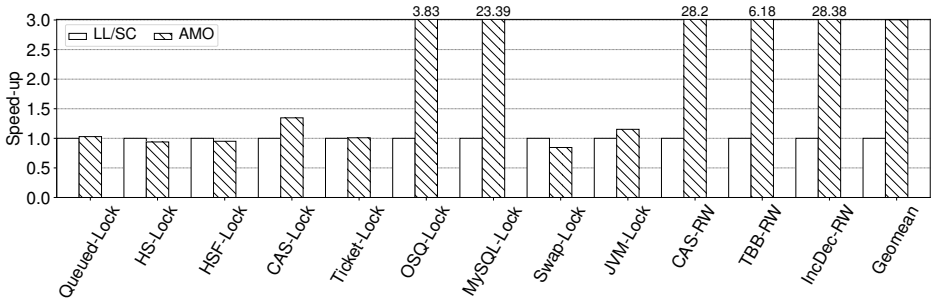

Figure 4.7: Speed-up of LockHammer locks (w.r.t LL/SC version) on Graviton 2 with 64 threads using *serialized* configuration

the live-locks that affected OSQ-Lock, TBB-RW, CAS-RW and IncDec-RW are reduced drastically. In the case of RW locks, we know that the contention of the lock is higher than in regular locks because more than one thread can access the CS in Read mode. So, the lock is updated more frequently. LL/SC is very sensible to multiple Store Conditional(SC) happening at the same time because only one SC can succeed. Since the release and new acquire of the lock are done back to back in *serialized*, all the RMW operations are executed consecutively increasing the amount of SC being executed in parallel. Meanwhile, in *unserialized* the RMW are distributed evenly in time, avoiding transient live-locks caused by a burst of updates. In the case of OSQ-Lock, we see the same behavior as in a RW lock because threads that are waiting are constantly updating a shared linked list queue that is managed as a RW lock.

4.1.3 Summary

The main take-aways we can derive from these experiments is that AMOs are more efficient in both machines, even if the overhead of AMOs in Kunpeng 920 is higher than LL/SC. We have observed that on high contention scenarios LL/SC can suffer from live-locks while AMOs scale without problems. We have seen that JVM-Lock and HSF-Lock are both the most efficient implementations for a Lock, while CAS-RW is the best implementation for RW-locks. Programmers should avoid using

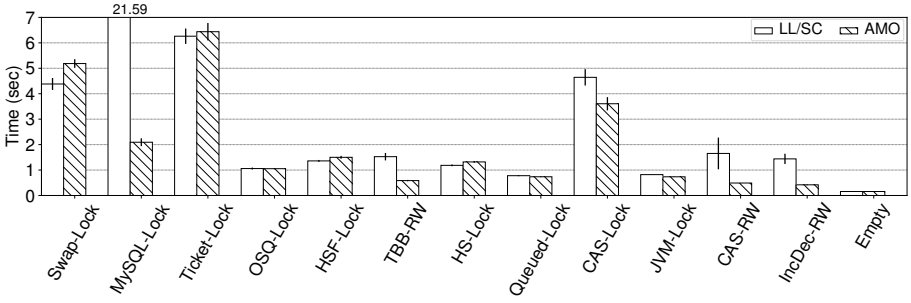

Figure 4.8: Absolute execution time of LockHammer locks on Graviton 2 with 64 threads using *unserialized* configuration two

Swap-Lock, Ticket-Lock and MySQL-Locks to avoid performance degradation in high contention scenarios.

4.2 Splash-3 Characterization

When evaluating Splash-3 we have used the execution time and scalability to measure how LL/SC and AMO instructions affect the performance of the parallel applications. First we evaluate Kunpeng 920, next the Graviton 2 and finally we compare both machines. When analyzing the Splash-3 benchmarks on a specific machine, we first plot the scalability of the LL/SC and AMO versions of all the applications. In order to obtain the scalability, we execute the same workload with different thread counts and compute the relative speed-up with respect to the single threaded version using LL/SC version. Since some applications only scale in powers of 2, we have used 1,2,4,8,16 and 32 threads in our experiments. Next, to measure the real benefit of using AMOs over LL/SC, we compute the speed-up of AMO version with respect to the LL/SC version. Instead of using the execution times obtained using 32 threads, we have taken the experiment with the number of threads that minimizes the execution time in each version. Hence, we can for example compare a execution of AMOs with 32 threads again a version of LL/SC with 16 threads. We do this because in some applications using a bigger number of threads can lead to higher execution times.

4.2.1 Kunpeng 920

Figures 4.9 and 4.10 show the relative speed-up of the LL/SC and AMO binaries on Kunpeng 920 with 1, 2, 4, 8, 16 and 32 threads. The speed-up is normalized to the execution time using one thread and the LL/SC version. Benchmarks are sorted by the highest speed-up from left (the lowest speed-up) to right. We split the results in two figures to ease readability.

In the first set of applications (see Figure 4.9) there are 4 applications (Volrend, Raytrace, Barnes and Cholesky) that have a Parallel efficiency below 25%, while the rest are below 50%. The second set of benchmarks (see Figure 4.10) include the applications that have a parallel efficiency of 50% or more. The overall scalability of applications is bad, only Water Spatial achieves ideal scalability.

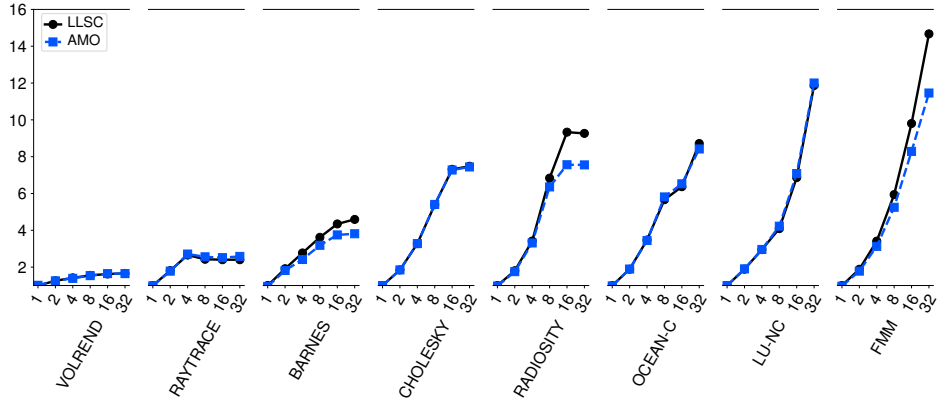

Figure 4.9: Splash-3 scalability on Kunpeng 920 (1/2)

Raytrace and Radiosity are the only applications that achieve lower speed-ups when increasing thread count, and both are raytracing applications. Raytrace uses a job stealing approach, in which each thread after finishing its own tasks steals tasks from other cores. Job stealing in scenarios with low amount of tasks can lead to load imbalance scenarios, as we have seen in the traces obtain with *extrae*. In contrast, Radiosity uses a shared queue approach to distribute the tasks between threads, so when the number of threads used is high the contention does not allow threads to get tasks from the queue.

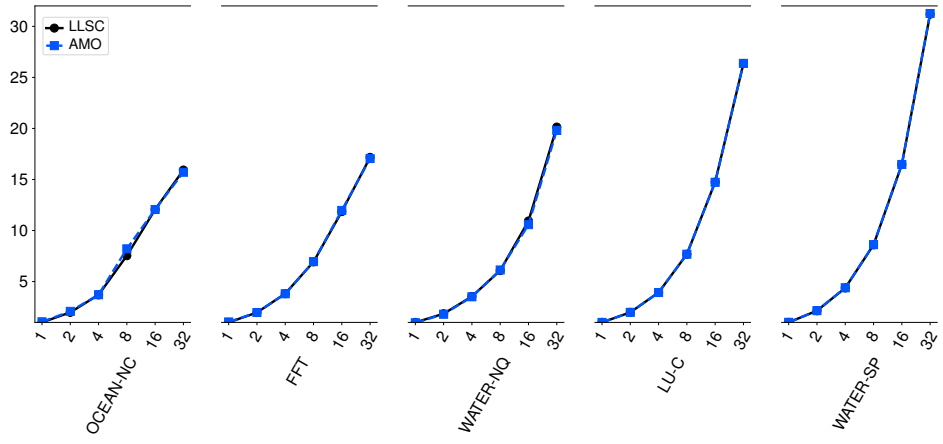

Figure 4.10: Splash-3 scalability on Kunpeng 920 (2/2)

Regarding behavioural differences between LL/SC and AMOs, we can see that Barnes, Radiosity and FMM have a significant better scaling when using LL/SC over AMOs. These three benchmarks are the ones with a higher ratio of synchronization instructions from the suite. We have executed our applications measuring the amount of synchronization instructions executed and the total instructions executed. With these two numbers, one can obtain the number of synchronization instructions per kilo instruction (SPKI). We can measure this value for different number of threads, but as we increase the thread count the amount of sync instructions can increase due to contention collision (i.e SC or CAS operations, can fail). First, we focus on single thread executions, the benchmark with the high ratio is Barnes with 1.98 SPKI, followed by Radiosity 0.78, FMM 0.44 and Raytrace 0.44. The rest have ratios below 0.01. Raytrace seems to be an interesting case, although it has a big ratio of synchronization instructions we cannot see such a big difference between LL/SC and AMO executions. This could mean that many of their synchronization operations are not in the critical path of the application, and can be delayed, or that both AMOs and LL/SC behave similarly.

However, the synchronization ratios obtained with single thread can

be misleading. For example, Volrend divides the workload in blocks depending on the amount of threads, and then synchronization is used to communicate results between blocks. Therefore, with a single thread, the problem is divided in one unic block, and no synchronization is needed. Thus, we have measured the SPKI using 32 threads. Some benchmarks like Barnes, FMM and Raytrace execute a similar number of sync instructions. Radiosity increases the number of synchronization instructions by 3× arriving to 2.95 SPKI. Cholesky goes from 0 to 0.84 SPKI, Ocean from 0.01 to 0.66, Water-Nsquared from 0.01 to 0.23 and Volrend from 0.0 to 0.19. Despite increasing the weight of synchronization primitives, we cannot see any difference between versions in the overall execution time.

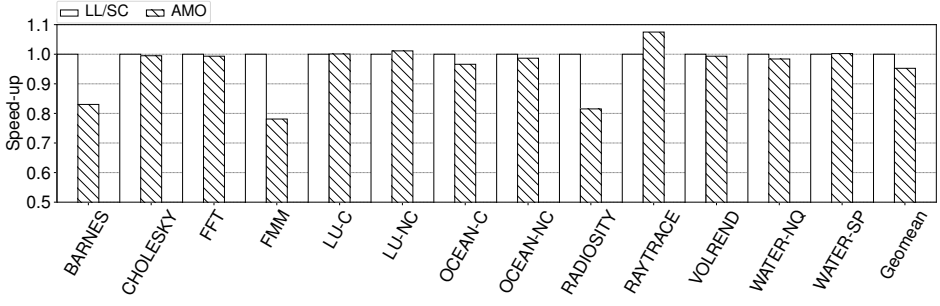

Figure 4.11: Speed-up of best AMO execution (normalized w.r.t best execution of LL/SC) running Splash-3 on Kunpeng 920

Due to the large y-axis scale of scalability plots, it is hard to correctly appreciate the exact magnitude of speed-ups in Figures 4.9 and 4.10. Thus, we have computed the speed-up of LL/SC and AMO versions running with 32 threads normalizing the execution time with respect to LL/SC version running with 32 threads (see Figure 4.11). In the figure, the AMO version of Barnes, FMM and Radiosity have slow-downs of 20% compared to LL/SC. AMOs only have a significant speed-up in Raytrace, outperforming LL/SC by 1.1×. The remaining benchmarks have similar execution times when using LL/SC and AMOs. When computing the geometric average we find that AMOs are a 5% slower than LL/SC.

4.2.2 Graviton 2

In the previous section we saw how AMO instructions do not offer any kind of benefit with respect to LL/SC instructions in real-world applications. To verify this premise we have re-executed the same experiments in Graviton 2. Again, we have plotted the scalability of the application in two Figures 4.12 and 4.13. Applications are sorted in the x-axis using the maximum speed-up achieved by each application. The maximum speed-up achieved by some applications changes from the Kunpeng 920 to the Graviton 2, so these applications appear in a different position than in Figure 4.9. The benchmarks that scale better in Graviton 2 than in the Kunpeng 920 are Volrend, Barnes and Radiosity.

The most remarkable difference between these experiments and the ones done in the previous section, is that AMO versions scale better than LL/SC. There are five benchmarks (Volrend, Raytrace, Barnes, Radiosity and FMM) in which the scalability of AMOs stands out over LL/SC. If we compare the speed-up of AMOs over LL/SC version we see big differences between Kunpeng 920 and Graviton 2. Whilst, Volrend achieved a 2× speed-up in Kunpeng 920, in Graviton 2 it achieves a 14× for AMOs and 8× for LL/SC. Barnes had 5× in Kunpeng 920, and in Graviton 2 obtains 8× for AMOs and 6× for LL/SC. Radiosity has same speed-up for LL/SC in both Kunpeng 920 and Graviton, but the AMO version goes from 8× to 11×.

In contrast, Cholesky drops from 8× to 5×. The rest of the benchmarks do not change or have some small speed-up that increment the scalability of both versions.

Raytrace and Radiosity scale poorly when using LL/SC instruction with more than 16 threads. The performance of 32 threads in both cases is worse than with 4 threads. As we explained in the previous section, these two raytracers depend on fine-grain synchronization to correctly execute theirs tasks. Therefore, a possibly weak implementation of LL/SC has a big impact on the performance, similar to what we see in Figure 4.12. It is also remarkable that four benchmarks reach 80% of parallel efficiency, whilst in Kunpeng 920 only two achieved that.

To conclude, we have computed the speed-up of AMO version with respect to LL/SC. We have used the experiments with the tread count

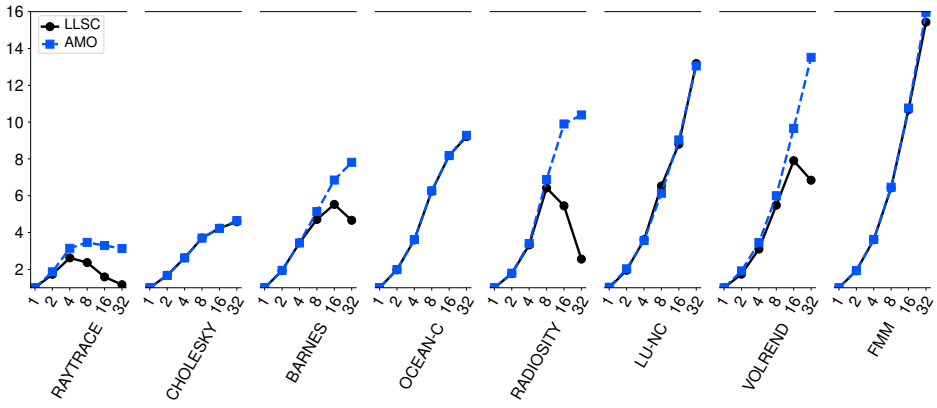

Figure 4.12: Splash-3 scalability on Graviton 2 (1/2)

that minimize the overall execution time in each version. We can see in Figure 4.14 that AMOs are faster or equal to LL/SC in all cases, and on average a 1.05× faster. Only Barnes, Raytrace and Volrend achieve significant speed-ups over 1.2×.

4.2.3 Kunpeng 920 vs Graviton 2

Because we saw big differences between the experiments of Kunpeng 920 and Graviton 2, we cannot conclude what are the benefits of using AMOs. One could thing that either AMOs are poorly implemented Kunpeng 920, or that Graviton 2 uses a LL/SC implementation that is slower than Kunpeng 920. To fairly compare both results we have plot the scalability with respect to the same experiments in Figures 4.15 and 4.16. We have normalized the speed-ups with respect to the Kunpeng 920 with LL/SC. Note that the figures have the y-axis is in logarithmic scale. This kind of plots are fair because both multicores execute the exact same binaries, both have a similar frequency, and we execute with the same number of threads.

As we can see, in most benchmarks Graviton 2 outperforms Kunpeng 920. There are several factors that make these performance differences, but the factor that is present in all benchmarks is the difference between the architecture of both machines. Graviton 2 uses an advanced

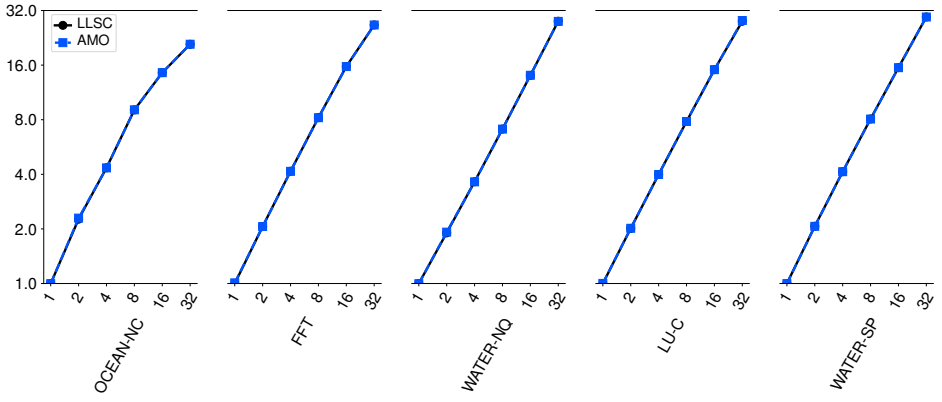

Figure 4.13: Splash-3 scalability on Graviton 2 (2/2)

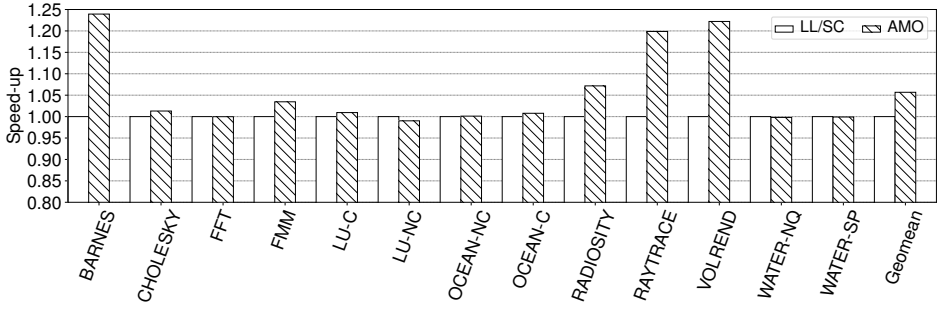

Figure 4.14: Speed-up of Splash-3 (normalized w.r.t. LL/SC) in Graviton 2

out-of-order core that devotes more resources to the different microarchitectural structures than Kunpeng 920. For example, Ocean-Contig is a memory bounded benchmark (0.33 Flops/Byte) that benefits from the extra memory ports available in the Graviton 2. However, in some of the experiments there are factors that have a bigger impact that the microarchitecture of the cores.

Volrend is an application that carries on a projection of a three-dimensional volume onto a two-dimensional image plane. One of the tasks performed by the application is to read from a 3D model stored in a file and then write the resulting image to another file. Kunpeng 920

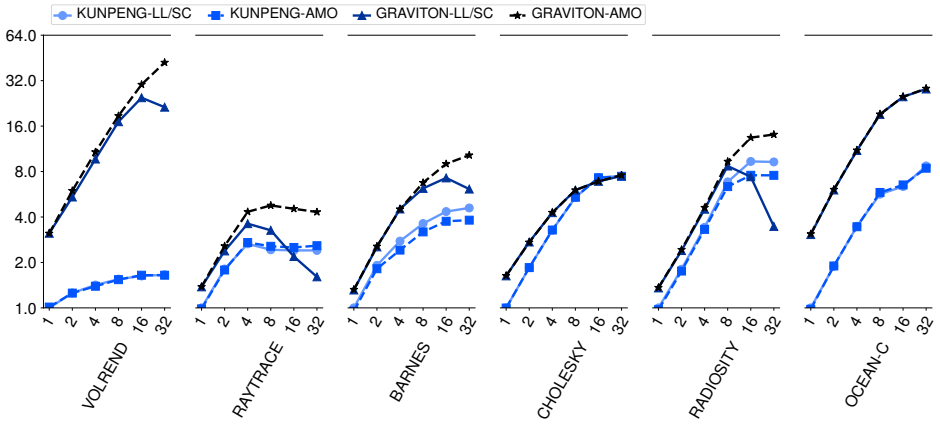

Figure 4.15: Splash-3 scalability on Kunpeng 920 and Graviton 2 (1/2)

uses a traditional hard-disc drive, while Graviton 2 uses a faster SSD for permanent storage. Therefore, Graviton 2 achieves more than 3× just on single thread performance. Despite this big difference on the storage technology, Graviton 2 still achieves a better scalability than Kunpeng 920.

In Raytrace, although LL/SC version is faster on Graviton 2 than Kunpeng 920 from 1 to 8 threads, from 16 to 32 threads we see that the performance of LL/SC drops below both versions on Kunpeng 920. This drop of performance is caused by transient live-locks in some global shared locks. Meanwhile, the AMO version running on Graviton 2 scales up to 5× with 8 threads but keeps this speed-up from 16 to 32 threads. We find another example of this behavior in Radiosity, in which we see that LL/SC version scales up to 8× with 8 threads. Then, the performance of LL/SC on Graviton 2 drops below the performance of both LL/SC and AMOs on the Kunpeng 920.

Cholesky is only application in which all four experiments have the exact same execution time. In this application the scalability is limited by the synchronization directives rather than the computation or the memory bound. However, the effect of replacing LL/SC by the AMOs does not affect the final performance.

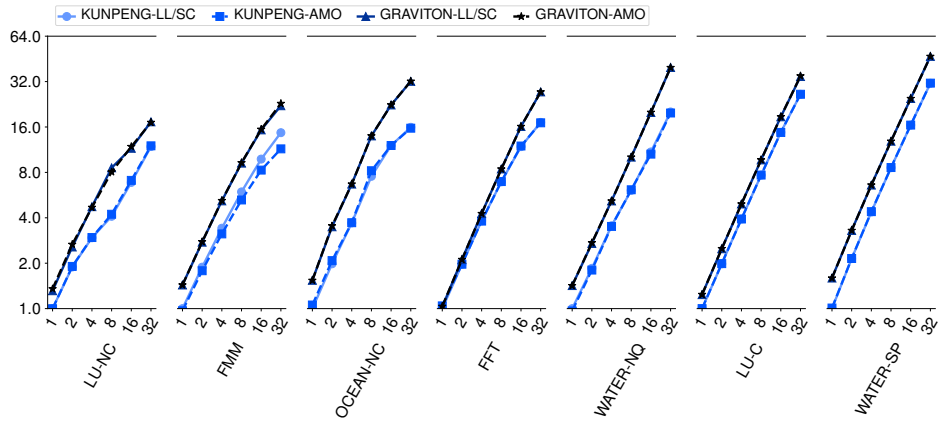

Figure 4.16: Splash-3 scalability on Kunpeng 920 and Graviton 2 (2/2)

4.2.4 Concluding Remarks

We have seen that the LL/SC instructions outperform AMO instructions in the Kunpeng 920. However, on the Graviton 2, the opposite is true. From the comparison between both machines we have learnt that Graviton 2 is much faster than Kunpeng 920. Moreover, we know that LL/SC do not scale well in Graviton 2 because they suffer from transient live-locks. On the top of that, we have observed how AMOs in Graviton 2 outperform all the other experiments.

Chapter 5

Modeling Atomic Memory Operations

In this chapter we introduce the specification of AMO transactions in AMBA 5 Coherent Hub Interconnect (CHI). Then, we explain how we have implemented AMOs in gem5. Since the specs enable multiple implementations of AMOs, we have performed a design space exploration study. Finally, we evaluate our implementations using the same benchmarks we used to characterize the test machines.

5.1 Advanced Microcontroller Bus Architecture 5 Coherent Hub Interface

Arm's Advanced Microcontroller Bus Architecture (AMBA) is an open standard for connection of components in a System-on-Chip (SoC). The fifth generation of the standard introduces the Coherent Hub Interface (CHI) specification, which was designed to be the standard of future high performance NoCs. The protocol describes the channels, type of messages and transaction flows to implement a directory based cache coherent system that supports MESI and MOESI protocols.

In a directory based system, all transactions are handled by a Home Node (HN or Directory) that co-ordinates snoops, cache, and memory accesses. Optionally, every HN can have attached a slice of the Shared Last Level Cache (LLC), which is usually the Third Level (L3). The core and its private caches (usually L1 and L2) are treated as the Request Node (RN). Main memory and other off-chip storage are the Slave Nodes (SN).

Since AMBA 5 CHI is a highly complex protocol, in the following subsections we are going to describe some fundamental concepts to understand how the protocol works and how AMOs are implemented.

5.1.1 Channels

Communication between components is channel based, which means that each type of message is associated with a specific channel. This design decision targets possible resource deadlocks in the NoC (i.e. chains of request blocked by a pending response). Channels and its functionality are described below:

- **REQ** Channel to send request from a RN to a HN
- **RESP** Channel to receive responses from a HN or RN
- **DATA** Channel to receive data from a HN or RN
- **SNOOP** Channel to receive snoops from a HN

Every channel has a list of fields that are used to send codified information. Since this description is only used by the real implementation and is not necessary to model the protocol, we have omitted this information.

5.1.2 Cache States

To determine which action is required on a cache line accesses or snoop, the protocol defines the cache states that encode important information about the cache line. Each possible cache state is derived from three different characteristics:

- **Valid / Invalid** When Valid, the cache line is present in the cache. When Invalid, the cache line is not present in the cache.
- **Unique / Shared** When Unique, the cache line exists only in this cache. When Shared, the cache line might exist in more than one cache, but this is not guaranteed.

- **Clean / Dirty** When Clean, the cache does not have responsibility for updating main memory. When Dirty, the cache line has been modified with respect to main memory, and this cache must ensure that main memory is eventually updated.

All the possible combinations of these characteristics result in five different states, summarized in Figure 5.1.

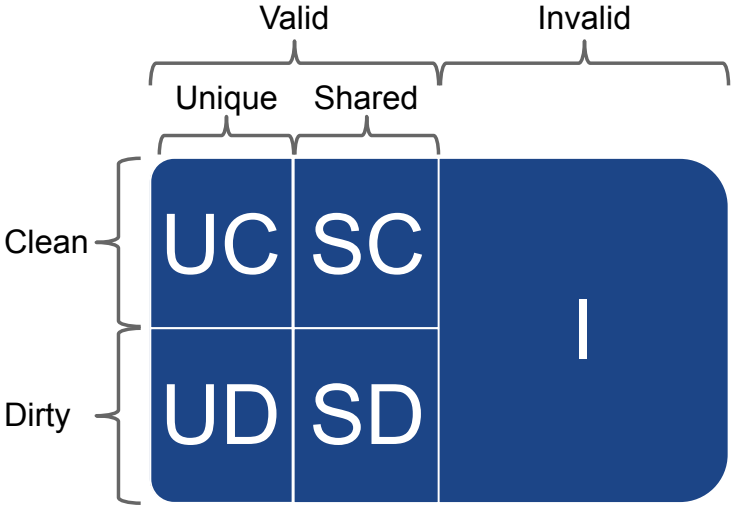

Figure 5.1: AMBA 5 CHI Cache States

5.1.3 Supported AMO

One of the key features of the protocol is the support for far AMOs that can be executed in the HN. When a RN wants the corresponding HN to execute the AMO remotely it issues an Atomic Transaction to the interconnection. These Atomic transactions can be classified in two categories based on their transaction structure:

- Transactions that return only a completion response
  - AtomicStore

- Transactions that return Data with a completion response
  - AtomicLoad
  - AtomicSwap
  - AtomicCompare

AtomicSwap exchanges a data value stored in a register by the value stored in a memory position. AtomicCompare uses two data values, the first one is a value of reference to be compared, while the second value is the data value to be swapped. When the AMO is executed, the original value stored in the target address is compared against the reference value. If they match, the swap value is stored at the address location. The transaction returns the original value that was previously stored at the memory position.

Then AtomicStores and AtomicLoads perform different kind of operations on a memory position using the original data stored (InitialData) and an operand (OperandData). AtomicLoads return always the InitialData into a register. The operations supported are:

- ADD: unsigned integer increment operation. Update location with $(InitialData + OperandData)$
- CLR: bitwise clear operation. Update location with $(InitialData\ AND\ (NOT\ OperandData))$
- EOR: bitwise exclusive operation. Update location with $(InitialData\ XOR\ OperandData)$
- SET: bitwise or operation. Update location with $(InitialData\ OR\ OperandData)$
- SMAX: signed integer maximum accumulator. Update location if $(OperandData–InitialData) > 0$ with OperandData
- SMIN: signed integer minimum accumulator. Update location if $(OperandData–InitialData) < 0$ with OperandData
- MAX: unsigned integer maximum accumulator. Update location if $(OperandData–InitialData) > 0$ with OperandData

- MIN: unsigned integer minimum accumulator. Update location if $(OperandData–InitialData) < 0$ with OperandData

5.1.4 AMO Transactions in AMBA 5 CHI

AMBA 5 CHI specifies a basic subset of transactions that all NoCs that are complaint must support. These transactions describe how the NoC must send the different messages through the virtual channels to ensure deadlock free communication. There are two types of atomic transactions used by AMOs, those that expect to receive the data and those that not. The former type uses the sequence of messages described below:

1. The Requester sends a request with the Atomic transaction opcode on the REQ channel.

2. The HN sends a DBIDResp response on the RESP channel. And returns the Read data using a CompData message on the DATA channel.

3. The Requester sends the Operands with the NonCopyBackWrData message on the DATA channel.

Those requests that do not expect any data coming from the HN (AtomicStores) have the next structure:

1. The Requester sends a request with the Atomic transaction opcode on the REQ channel.

2. The HN sends a CompDBIDResp response on the RESP channel.

3. The Requester sends the Operands with the NonCopyBackWrData message on the DATA channel.

Both kinds of transactions are summarized in Figure 5.2.

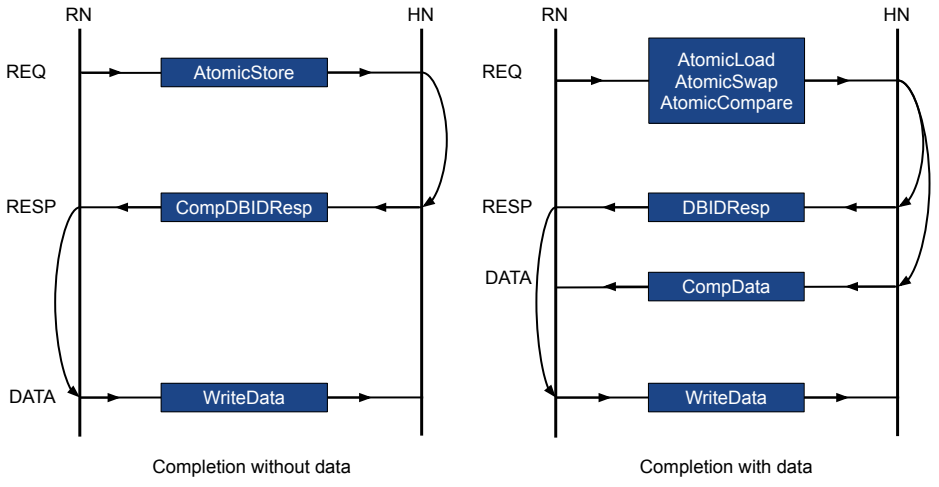

Figure 5.2: Atomic Transaction diagram

5.1.5 Snoops

Invalidation-based protocols enforce the *single-writer, multiple-reader invariant*: at a given point in time, a cache line may either have at most one sharer with read-and-write permission, or multiple sharers with read-only permission. In order to follow the invariant, when an AMO occurs it needs read-and-write permission. Therefore, the Point-of-Coherence (PoC) must send snoop messages to invalidate the private copies of the cache line. The AMBA 5 CHI protocol specifies many different types of snoops that not only invalidate cache lines in private caches, but can also make the snoopee forward data, force writebacks, etc. The AMBA 5 CHI specifies a unique type of snoop for AMO, the SnpUnique. This snoop request is sent to the snoopee to obtain a copy of the cache line in Unique state while invalidating any cached copies. This snoop is also used by stores to obtain Unique state cache lines.

5.1.6 Example Flowcharts

Summing up all the previous elements we can understand in detail how the protocol works. We have taken some examples drawn in a flowchart format from the AMBA 5 CHI manual to illustrate how the protocol would act in some cases.

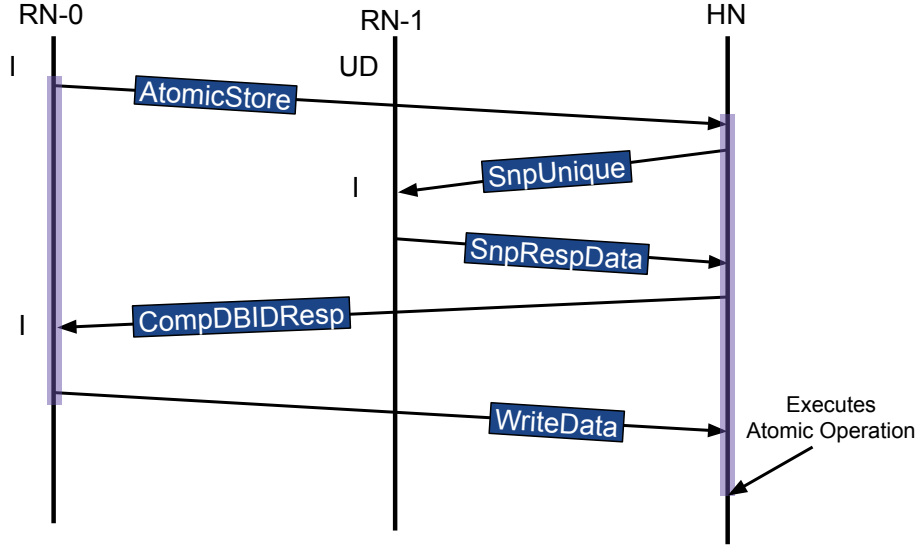

Figure 5.3: Example 1 flowchart

For the first example (see Figure 5.3), we assume that there is a cache line that is mapped to an specific HN. At same time, there are two different RN, one has already fetched the target cache line, so it is present in the L1 with UC state (RN-1). Meanwhile, the RN-0 does not have the cache line in any of its private caches. The latter issues an AtomicStore (i.e STADD) to the HN, in order to perform a far AMO. To perform the AMO, HN-0 needs the most recent value of the cache line and no other copy of the block can exist to keep consistency and coherency. Therefore, the HN issues an invalidation message (SnpUnique) to the owner of the cache line, so RN-1 returns the latest value and invalidates its local copy. Once the HN receives the most updated data (SnpRespData), a completion message (CompDBIDResp) is sent to the original requester to notify that its request

has been processed. Then RN sends the operand data to complete the operation (WriteData).

Note that once the WriteDate message is sent, the core attached to the RN does not need to keep waiting until the operation is finished. The head of the Reorder Buffer that was waiting until the completion of the operation is freed. New instructions can be committed, even store operations. This is possible because any new read or write operation targeting the cache lined modified by the AMO will be ordered after the AMO.

The second example (Figure 5.4) illustrates the other kind of operation, the AMO with return data. This time the requested cache line is not present either in any RN or the HN. Thus, the block is only present in the SN (main memory). Once the HN receives the AtomicSwap request it issues two messages, one to the SN requesting the memory block and other to the RN acknowledging the request. Once the HN receives the corresponding cache line, it sends the value that corresponds to the memory address sent in the original request. Note, that the HN does not need to wait until the write data has arrived because the request has already been serialized. Once this messages arrives to the RN, the RN can mark the transaction as finished and issue new requests to the same address.

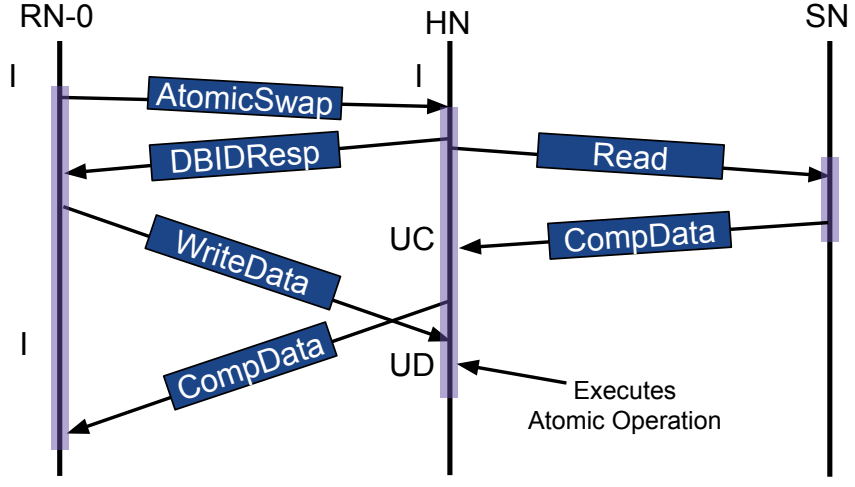

Figure 5.4: Example 2 flowchart

Meanwhile, the RN sends the missing operand needed to perform the AMO to the HN. The HN uses this data to perform the AMO. Once the result is stored in the cache, the HN can process the next request.

5.1.7 AMBA 5 CHI on Gem5

Arm Research has recently integrated its own implementation of the AMBA 5 CHI protocol in gem5-21.0 (named as CHI [16]). The CHI ruby protocol provides a single cache controller that can be reused at multiple levels of the cache hierarchy, which means that the same state machine is able to model L1, L2 or the HN. The protocol can be configured to model multiple instances of MESI and MOESI cache coherency protocols. Although the CHI protocol is based in the AMBA 5 CHI protocol, it does not follow all the features and technical details (i.e. partial writes, QoS and Error handling are not implemented).

The CHI gem5 protocol does not support any AMOs by default. Therefore, we have modified it to enable the execution of near and far AMOs. We have followed the AMBA 5 CHI manual in order to keep the original specification. We have limited the execution of the AMOs to the L1 and the L3, and we have excluded the L2. We haven taken this decision due to the recursive design of the protocol. One of the possible implementations of the AMBA 5 CHI protocol is performing the AMO near whenever the block is present with write permissions (UC or UD states), and far if the block is not present or with only Read permissions (I, SC or SD states). We name this policy Unique Near. In a real machine, the RN comprehends the L1 and the L2 (as it is explained in the specs). When an AMO is executed near the block could be located any of the two levels. In case that the block was present in the L2, it is relocated to the L1 to perform the AMO.

However, in CHI protocol L1 and L2 are two separate instances connected through the interconnect. The problem appears if an AMO misses in the L1, so a far AMO is issued to the L2 and then the AMO hits in the L2 with UC or UD states. In this case we are not able to relocate the block in the L1 (We would need to rewrite the whole protocol). The solution we have taken is to forward the operation to the L3, and invalidate the copy present in the L2.

Another solution, that we would have wanted to explore is placing an Arithmetic Logic Unit in the L2. Hence, in case of hitting the L2 the AMO is directly executed there without relocation of the block. We have discarded this option because it implies consuming some area of the L2 that could be devoted to the prefetchers, second level TLB or extra entries. But we do not discard the idea for future works.

5.2 Design Space Exploration

The specs do not enforce how atomics must be sent. However, the specification gives some hints about what options are available when implementing this decision.

If the cache line is Shared but not Dirty, it can either:

- Generate a ReadUnique or CleanUnique to gain ownership of the cache line and perform the atomic operation locally.
- Invalidate the local copy and send the Atomic transaction to the interconnect.

If the cache line is Shared Dirty, it can either:

- Generate a CleanUnique or ReadUnique, gain ownership of the cache line, and perform the operation locally.
- WriteBack and Invalidate the local copy and then send the Atomic transaction to the interconnect.

Thus, there are two possible decisions for Shared Clean and Shared Dirty: perform the operation in the directory or gain write permissions to perform the operation in the L1. But, instead of applying this decision only to two states, we go further and assume that this decision can be done for every cache line state. From this scheme one can derive up to 32 different static policy implementations. However, many of the possible implementations do not have any rationale behind and seem counter-intuitive. From the 32 possible policies we have selected 6 implementations listed in Table 5.1.

Policy Name	**UC**	**UD**	**SC**	**SD**	**I**
All Near	N	N	N	N	N
Present Near	N	N	N	N	F
Dirty Near	N	N	F	N	F
Shared Far	N	N	F	F	N
Unique Near	N	N	F	F	F
All Far	F	F	F	F	F

Table 5.1: Comparison table of policy Design Choices. Near (N), Far (F)

The first policy is All Near, it is the most simple policy implementation because it always executes AMOs in the L1. With All Near, AMOs behave as regular Writes, the L1 requests write and read permission for the accessed cache line using a Read Unique Transaction. Once the block is in L1 in UC or UD states it performs the AMO. Similarly, Present Near performs the AMO near if the cache line is already present in the L1. If the cache line is invalid Present Near performs a far AMO request to the L3. The heuristic behind the protocol is that if the block is already on the cache is better to gain write access to it, than invalidate the block.

On the other extreme of the design space, All Far policy assumes that the best way to execute an AMO is sending it to the L3. This way we avoid bringing it to the L1 and prevent future invalidations. However, when the block is in UC or UD state, the L1 needs to perform a writeback because there is no other copy in the other caches (L3 is exclusive from L1 and L2). Unique Near policy is more conservative and assumes that if the block is already present in the L1 with Write permissions, it is not necessary to pay extra cycles to writeback the cache line. It also assumes that if the block was already present there may be locality of reference that can be exploited.

Between these two extremes we have Dirty Near and Shared Far policies. The former policy assumes the same premise as Unique Near, but also that if the block is in Shared Dirty state it means that the L1 was the last writer of the cache line and it may be the next writer in the future. In contrast,

Shared Far policy assumes that if the block is in Shared state there are other cores that generate invalidations to read the data, and is better to perform atomics at the L3. Furthermore, if the block is invalid it may be because it was evicted from L1 to L2, and the best choice is to bring it back to the L1.

For now on, we will refer to these policies as static policies because they perform always the same choices based only on the cache line state.

5.3 Evaluation

In this section, we evaluate the performance of the different AMBA5 5 CHI implementations in gem5 resulting from the design space exploration done above. The evaluation follows the same structure as chapter 4.1. First, we present the characterization of the static AMO policies with LockHammer microkernels. We evaluate two scenarios using *serialized* and *unserialized* configurations. Then, we execute the Splash-3 benchmarks to analyze how applications scale with the different static policies.

5.3.1 Static AMO Results with LockHammer Serialized

First, we obtained the absolute execution time of LockHammer benchmarks using *serialized* configuration and 64 threads in gem5 (see Figure 5.5). For each kernel there are 6 bars, each one represents the execution time of a different static AMO policy. Bars are sorted from right to left, being the bar on the right the policy executes all AMOs in the L1 (All-Near) and the left bar the policy that executes all AMOs in the L3 (All Far).

As we have seen in Section 4.1 with the commercial machines, the best RW Lock implementation is CAS-RW, in this case with All Near policy. The fastest lock implementation is MySQL-Lock, while on the physical machines the fastest were JVM-Lock or HSF-Lock. If we look at the general picture we observe some resemblances with bars that correspond with AMO experiments in Figure 4.6. For example, the three slowest lock implementations in both figures are the Ticket-Lock, Swap-Lock and CAS-Lock. While the three fastest in both figures are JVM-Lock, Queued-Lock and MySQL-Lock.

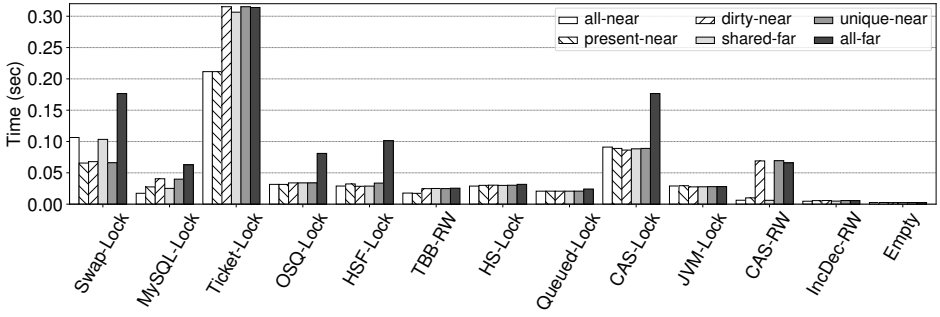

Figure 5.5: Absolute execution time of LockHammer locks with 64 threads on *serialized* configuration in gem5

By looking at Figure 5.5 we cannot extract many insights due to the big amount of information and the scale of the bars. Overall, we can see that All Far policy is the slowest policy in most kernels. The reason of this low efficiency is cause by the design of the static policy itself. The main fault of All Far policy is that when the target cache line of the AMO is present in the L1, the protocol first needs to evict the block from the L1 to issue then the remote AMO. This way the latency of an AMO is increased without necessity.

Among all static AMO policies we have chosen All Near as the baseline. Now, we can compute the speed-up of each version with respect to our baseline in order compare all policies in a simple way. Since we already know that All Far policy perform always worse than the rest we have omitted the results from the plots. In Figure 5.6 we can see the speed-ups of all the static AMO policies.

On one hand, there are several kernels that are not sensible to the AMO policy used, such as OSQ-Lock, HSF-Lock, HS-Lock, Queued-Lock and CAS-Lock. Because the variation with respect to the baseline is 10% at most, which is small for a synchronization microkernel. On the other hand, the AMO policy can have a big impact in Swap-Lock, MySQL-Lock, Ticket-Lock, TBB-Lock, CAS-RW and IncDec-RW.

On average, All Near policy outperforms all other policies. Present Near and Shared Far are almost as fast as All Near (8% slowdown). In the case of Dirty Near and Unique Near, we see a slowdown of 30%, caused by

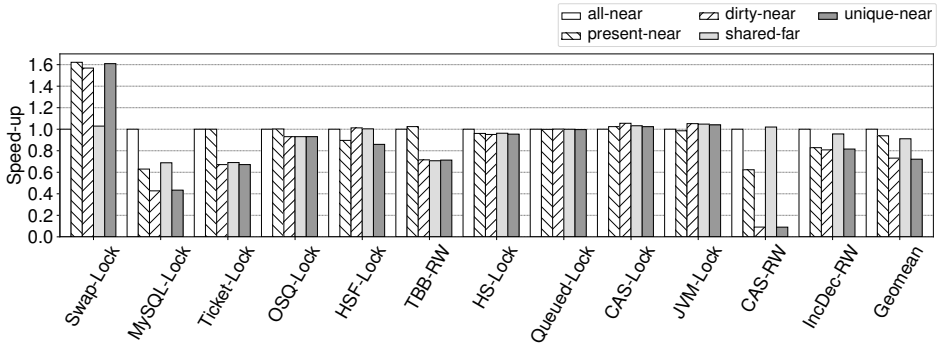

Figure 5.6: Speed-up of LockHammer locks (w.r.t All Near policy) with 64 threads on *serialized* configuration

a poor performance on CAS-RW and MySQL-Lock.

We have used our tracing mechanism to understand how policies work. So, in the following subsections we will explain the reasons why some policies are more efficient than others for those kernels that are sensible to the AMO policy used.

Swap-Lock

Swap-Lock is a lock that benefits from far AMOs. When threads try to acquire the lock, they constantly issue SWP instructions to exchange the be lock value to locked, even if the lock is already locked (see Listing 5.1). This creates a lot of pressure in the HN that holds the lock cache line. Issuing a far AMO instead of fetching the cache line to perform a Near AMO, can help to reduce the congestion in the HN controller. The total latency of invalidating the previous owner of the cache line plus fetching the cache block to the new owner is higher than just issuing the AMO to the HN and performing the operation there.

Listing 5.1: Swap-Lock implementation

```
uint64_t lock_acquire (uint64_t *lock) {
    uint64_t val = 1;
    while (val) {
        val = swap64 (lock, 1);
```

```
    }
    return 0;
}

void lock_release (uint64_t *lock) {
    lock = 0;
}
```

Present-Near, Dirty-Near and Unique Near outperform both All Near and Shared Far. The key to understand this behavior is to analyze what are the cache states the lock goes through. As we have said there are no load instructions but only writes and AMOs. Therefore, the states that the lock cache line visits are always UC, UD and I. We see that the fastest policies apply near AMO for the UC and UD and far AMO for the I. Meanwhile, All Near and Shared Far fetch the cache line block when is Invalid. Therefore, issuing far AMO when the block is invalid state reduces the pressure at the HN.

MySQL-Lock

MySQL-Lock is an optimistically Swap-based lock implementation. First, the acquire function tries to capture the lock with a SWP instruction. Second, a second SWP instruction is issued to release the lock. If the thread fails to acquire the lock, it is penalized to execute a random amount of NOPs that range from 50 to 30000 as a back-off mechanism. Therefore, the thread that is able to acquire the lock will easily execute several CS, while the other threads execute NOPs. When the threads that were executing the back-off function finish their penalty, they try again to capture the lock with a SWP instruction. These threads fail in most cases because the owner of the lock only releases it for a small fraction of time. Therefore, threads in this kernel perform all their workload sequentially, so fetching the cache line that contains the lock is always beneficial.

To illustrate this behavior we have plotted Figure 5.7, which shows a trace obtained with gem5 simulator using All Near policy. X-axis represents time, and Y-axis represents the thread number. Colors represent the actions made by each thread. While purple represents the time lapse a thread is blocked performing a near AMO (fetch +

operation), white represents the amount of time that a thread is performing any other kind of task (i.e. computation, regular memory access, branching, etc.). In this trace, we can see in that thread 27 is performing several AMOs, some of them take thousands of cycles. Since we know that AMOs are used to capture and release the lock we can see that threads 27 is iterating in the CS while the other threads wait. We see that threads 5, 16, 19 and 22 issue some SWP AMOs. We know that these threads do not acquire the lock because the execute a single AMO (try to capture) instead of two consecutive AMOs (capture and release). Meanwhile, thread 27 is able to perform 24 pairs of AMOs that mean 24 CS.

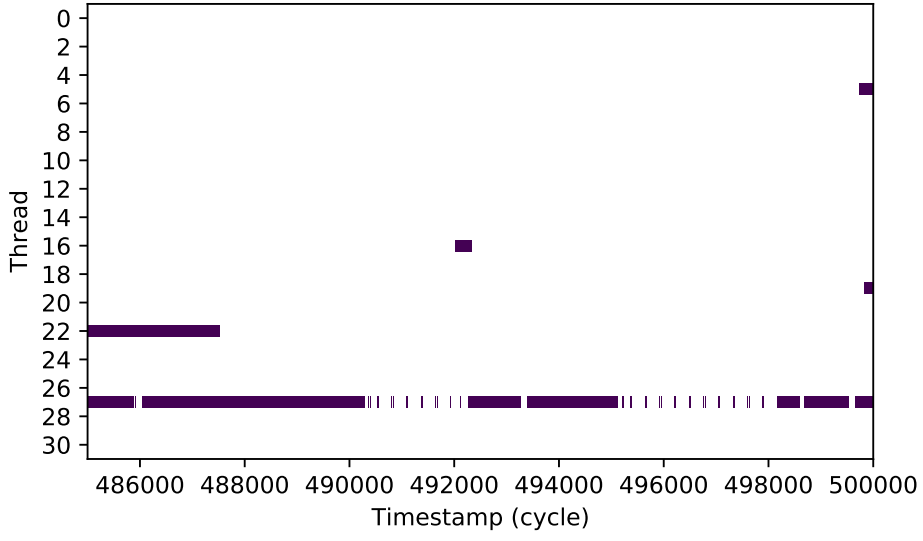

Figure 5.7: Trace of 32 threads executing MySQL-Lock kernel with All Near policy. Each rectangle represents the amount of time spent executing an AMO

Ticket-Lock

Two global shared 16-bit counters are used in Ticket-Lock to guarantee that only one thread executes the CS. One counter is stored in the highest 16-bits of a 32-bit word in memory and the other counter is stored in the lowest

16-bits. We refer to this 32-bit memory word as the Lock. Placing counters in the same memory word guarantees that both counters are always fetched at the same cache line, and the higher and lower parts cannot be updated at the same time by two threads. The first counter is used by threads to obtain a ticket through a fetch-and-add (LDADD) instruction. This AMO increments the upper 16-bits of the Lock and returns the previous value which is the thread ticket. Then, threads compare their ticket against the lowest 16-bits of the Lock that encode the second shared counter. When the ticket matches this counter, the thread can access the CS. To release the lock, threads atomically update the lowest 16-bit of the Lock with a STADD instruction. Notice, that instead of a LDADD, the directive uses a STADD instruction is used because threads releasing the lock do not care about the value of this counter.

When threads check if their ticket is equal to the lowest 16-bit of the Lock, they fetch the cache block in SC. Therefore, when a thread enters and finishes the CS, the block is still in SC state. In Dirty Near, Shared Far and Unique Near policies the thread executing the STADD instruction will issue a far AMO, while in All Far and Present Near they will fetch again the block. The problem of issuing a Far AMO is that soon after releasing the lock, the thread updates again the highest 16-bit of the Lock to get a new ticket. In this case, having already the block in the L1 reduces the latency of the second AMO.

TBB-RW, CAS-RW and IncDec-RW

These three kernels implement a RW lock through a thread shared counter. In TBB-lock we have multiple functions that enable the acquire and release of the lock. In the CAS-RW and IncDec-RW we have a much simpler mechanism, the shared counter is incremented at the beginning of the CS and decremented at the end. In the case of IncDec-RW, two LDADD instructions are used, one with positive value and the others with negative value. In CAS-RW, since there is a maximum amount of threads that can enter the CS in parallel, the function that acquires the lock uses a CAS instruction to avoid a possible underflow of the counter, while the release function uses a LDADD instruction.

The reason why fetching always the block is beneficial in RW locks is

that the parallel section of the kernels is zero and the CS is just 500 NOPS. Thus, when a thread fetches the cache line that contains the lock, it is able to complete several CS before being invalidated.

5.3.2 Static AMO Results with LockHammer Unserialized

The next step in our evaluation is to execute again LockHammer kernels using *unserialized* configuration. Figure 5.8 shows the absolute execution time of the kernels running with 32 threads. We can see that the global picture is very similar to *serialized* configuration. However, there are three experiments that change from one configuration to another. All Near policy increases the execution time in MySQL-Lock. Present Near policy now outperforms all other policies in Ticket-Lock. Shared Far policy experiments a big performance degradation in CAS-RW kernel.

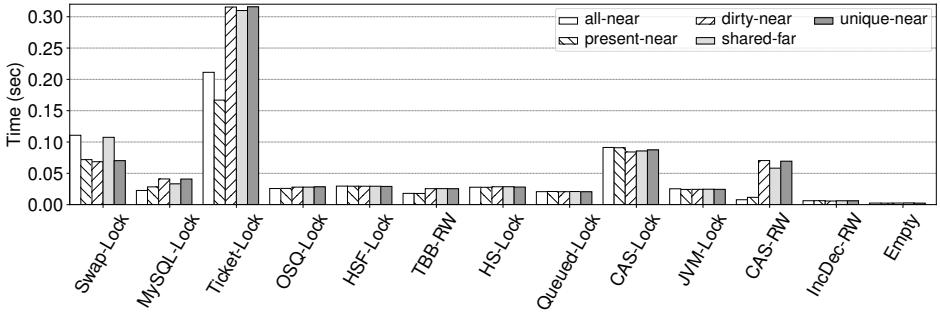

Figure 5.8: Absolute execution time of LockHammer locks on with 64 threads *unserialized* configuration in gem5

Figure 5.9 shows the speed-up of all policies with respect to All Near policy. We can see that Present Near and Unique Near mitigate the slowdowns from 60% to 42%. Moreover, Present Near improves its 40% slowdown to 20%. As we mentioned in the previous section, All Near policy is more suitable in MySQL-Lock with *Serialized* configuration. A single thread can constantly enter and exit the CS while the other threads perform back-off functions because it always have the lock cache block in the L1. However, using *unserialized* configuration reduces this advantage because after releasing the lock is more probable that other threads

acquire the lock.

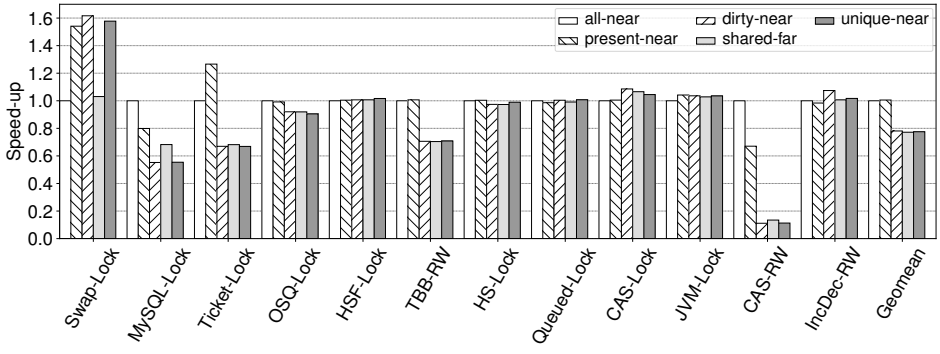

Figure 5.9: Speed-up of LockHammer locks (w.r.t All Near policy) on with 64 threads using *serialized* configuration

In Ticket-Lock we see a similar behavior. Now, the thread that releases the lock needs to wait to get a new ticket. During this period is probable that another thread enters the CS. If the thread in the CS releases the lock before the previous owner gets a new ticket, the former will invalidate the local copy of the latter. After invalidated, with All Near policy this thread will fetch again the block despite it will not modify again the block in the near future. This action invalidates the local copy hold by the thread releasing the CS, which is going to modify the lock again to get a new ticket. This action increases the overall execution time of both threads. With Present Near policy the thread that has been invalidated issues a far AMO that does not fetch the cache line. Thus, avoiding future invalidations.

As we have mentioned before, CAS-RW has an acquire function that increments a shared counter through a CAS instruction. This kind of approach has low efficiency in high contention scenarios because CAS instructions fail several times. In this case, we have found in our experiments that fetching the cache line to perform the AMO is better than issuing a far AMO. The reason is that the if the first AMO fails, the subsequent AMOs can have higher probability to succeed if the block is in the L1.

Thanks to computing the speed-up, we can notice some differences in IncDec-RW kernel. Present Near and Unique Near reduce their

slowdowns from 20% to zero slowdown. Meanwhile, Dirty Near policy goes from a 5% slowdown to a 8% speed-up. The explanation to these speed-ups is a big slowdown on All Near policy, rather than an improvement in the execution time of these policies. As happened with MySQL-Lock, All Near policy benefits from having the lock cache line in the L1 with *serialized* configuration. But, when we change to *unserialized* configuration the chances to receive an invalidation are higher.

5.3.3 Static AMO Results with Splash-3

In this section we present the evaluation of the static AMO policies with the Splash-3 benchmark suite. We have executed the whole benchmark suite, but we will only show the results of those applications that present some sensitivity to changing the AMO policy, namely: Barnes, Radiosity, Volrend, FMM and Water-Nsquared. Despite Raytrace has a big interaction with AMOs we have discarded the application because it suffers heavily from load imbalance. This load imbalance leads to misleading results because depending in how tasks are randomly scheduled we obtain different execution times. Thus, we cannot differentiate from real speed-ups and variations in task scheduling decisions.

Figure 5.10 shows the relative speed-up of each Static AMO policy with respect to All Near policy. We have selected the amount of threads that minimize the execution time for each benchmark. Note that the speed-up scale starts at 0.5 for better readability. We are going to analyze the performance of the Static AMO policies from right to left. Starting with Barnes, we can see that the best policy is Present Near, followed by Shared Far. In this benchmark we have a big array of locks that controls the access to a shared data structure. Since locks are allocated contiguously in memory we observe a false sharing pattern as stated in other papers [38]. The use of Present Near and Shared Far seems to partially alleviate this effect.

Dirty Near and Unique Near policies perform better than the rest in Radiosity. In this benchmark we have used the traces to understand where these speed-ups come from. We have observed that there is a global lock that is shared among all threads and it is used very frequently.

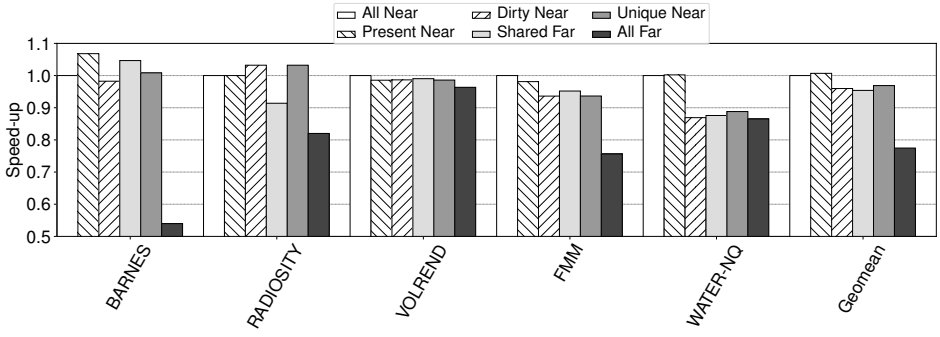

Figure 5.10: Speed-up of Static AMO policies (w.r.t. All Near) simulating Splash-3 benchmarks on gem5.

Meanwhile, there are other addresses that are accessed by a single thread frequently. Therefore, we observe different lock sharing patterns. Dirty Near and Unique Near policies adapt better to both patterns. In the case of the highly shared lock the AMOs are performed far, while those addresses that are accessed by a single thread are executed always near.

For Volrend, FMM and Water-NQ, we see that All Near policy outperforms all other policies. As more AMOs are executed remotely the performance degrades. There are many different locks being used during the execution of these applications, so the probability of two threads fighting for the ownership of the same lock is low. Since, these locks have big reuse patterns, bringing a lock block to the L1 is beneficial due to subsequent accesses.

On average, we can see that the policies are performing similarly, and the performance difference is below 4%. The only exception is All Far policy. As we have seen in LockHammer this policy has a poor performance caused by its higher latency executing AMOs.

5.4 Concluding Remarks

In this chapter, we have described how AMBA 5 CHI implements near and far AMOs. Moreover, we have implemented six different policies following the specs in our cycle accurate simulator (gem5). We have evaluated these

policies and we have identified several different behaviours.

In the case of LockHammer, there are many kernels in which the AMO policy used does not have a large impact. In those cases in which we see some differences between policies, All Near policy outperforms the rest. The advantage of this policy is based on the capability of the thread to reuse the block that has been fetched into the first level of cache.

We have seen a different picture in Splash-3 benchmarks. In this suite, Static AMO policies have a bigger impact on the final execution time. From the six static policies we have tested we have not found one that outperforms the rest. Therefore, different sharing and access patterns benefit from different AMO polices.

Chapter 6

Dynamic AMO Policy Predictor

Based on the insights we have obtained in previous chapters, we introduce the DynAMO, a microarchitectural predictor that dynamically selects the most suitable AMO policy. Through this chapter, we will introduce the architecture of the predictor, a sensitivity study of the parameters of the predictor, and a performance evaluation of our solution.

6.1 Choosing the Best Static Policy

As we have seen in Section 5.3, no single static policy outperforms the rest for all benchmarks. Therefore, having a single static policy does not guarantee always the best performance and opens a challenge to find better policies. The naive approach to solve this problem is building a predictor to dynamically predict which static policy must be used, in the same way Tournament Branch Predictor is able to select the best branch predictor for each branch.

Before starting to develop a policy predictor, we have computed the maximum speed-up achievable choosing for each workload the best policy. Figure 6.1 shows the speed-up achieved selecting the best policy with respect to All Near policy executing LockHammer suite with *serialized* configuration. We can see that on average we can only gain a 1.05× with respect to All Near policy. As we have already seen in Section 5.3, many locks are not sensible to the AMO policy used. And those kernels in which there are some significant variation, All Near policy tends to dominate over the other policies. Since the differences in behavior between *serialized* and *unserialized* configurations are small we have

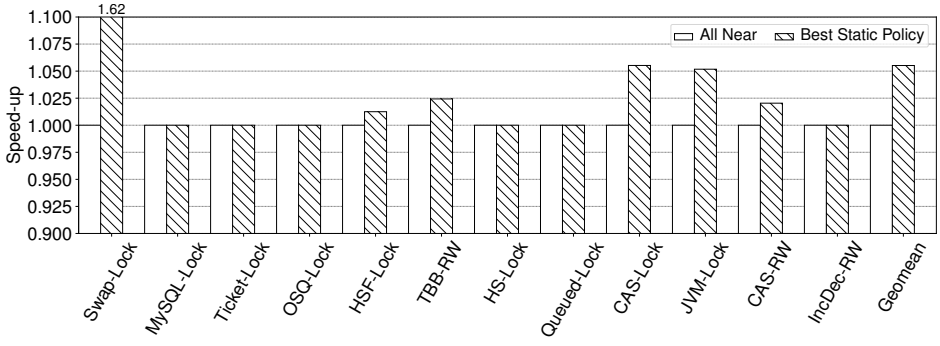

Figure 6.1: Speed-up of best AMO policies (w.r.t All Near) running LockHammer with 64 threads using *serialized* configuration.

omitted the results with *unserialized* configuration.

Next, we have repeated the process with the benchmarks from the Splash-3 suite. Figure 6.2 shows the maximum available speed-up we can obtain selecting one policy for each benchmark with respect to All Near policy. Note that we have removed All Far policy due to its low performance and the y-axis scale starts at 0.85 for clarity. The achievable speed-up selecting the best policy compared to All Near policy seems to be small, we can only achieve a 1.07× in Barnes and 1.03× in Radiosity. Therefore, the achievable speed-up gained on average is only 2%

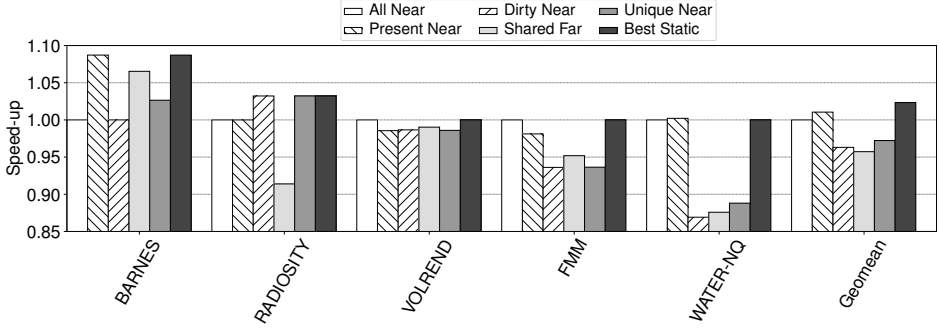

Figure 6.2: Speed-up of best AMO policies (w.r.t All Near) running Splash-3 benchmarks.

However, we have to note that with this approach we can obtain only a limited speed-up. Since we are applying a single policy to all addresses, we may be using the wrong policy in some of the addresses. Thus, we could take advantage of a finer granularity in the selection of policies to speed-up applications. However, before shorting to implement a predictor we want to understand what is the upper-bound of the speed-up we can achieve. This way, we can avoid investing any effort into a mechanism that obtains a negligible speed-ups. Thus, we have developed a new Ideal policy that simulates an unrealistic scenario in which all AMOs are executed in three cycles. With this new model we have obtained the results shown in Figure 6.3.

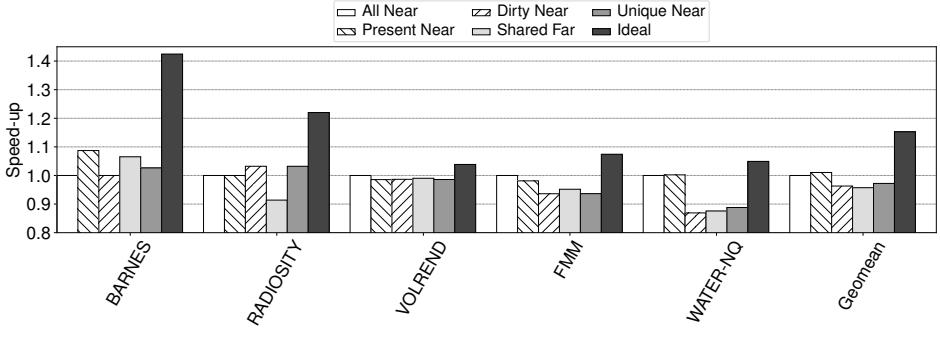

Figure 6.3: Speed-up of Ideal (always 3 cycles AMO) AMO policy (w.r.t All Near) running Splash-3 benchmarks.

Thanks to this Ideal model, we know that we cannot obtain speed-ups over 1.1× in Volrend, FMM and Water-Nsquared benchmarks. However, in Barnes and Radiosity we can achieve speed-ups of up to 1.42× and 1.22× respectively. On average, the maximum speed-up we can achieve optimizing AMOs is 1.15×. We have also tested this ideal model with the LockHammer but the results obtained are not significantly better than those obtained with the Best Static policy. All in all, we have considered that Splash-3 speed-ups are good enough to design and develop a dynamic AMO policy predictor.

6.2 Design Philosophy

One of the most studied fields in Computer Architecture is Branch Prediction (BP). We can see that there are several similarities between selecting near or far AMOs and Branch Prediction. In both cases we have to apply a binary decision. In both cases we have an instruction that has a target address (in the case of BP is the address to jump). We can be attracted to the idea of applying the same mechanisms that have been applied to branches for AMOs. One could replace Taken and Not Taken decision by Near and Far. However, in the case of branches, after some cycles we end up knowing the correct answer to update the predictor. With this information predictors learn and improve their future predictions. However, in AMOs we do not know what is the right prediction. We could consider right prediction if a block brought to the L1 with an AMO is reused afterwards, and a bad prediction if the block is evicted without being used.

Although this idea is attractive, it does not cover the problem of labeling the prediction of far AMOs. Once we have issued a far AMO, we do not know if the prediction was right or not. If we tracked subsequent accesses to detect missed opportunity to fetch the cache line, we would need to track the AMO meta data indefinitely, which can be expensive. And even with the required hardware, we cannot know if the block would have been invalidated by other cores.

In addition, we only know local information with respect to a single core (i.e. if the block is present in L1, subsequent AMOs executed, etc.), but the actions of a single core can affect third parties. A core that instead of issuing a far AMO, fetches the cache line can increase the access time of another cores. Once the block is in a L1 in UC/UD, when the HN receives a request needs to first send an invalidation to the current owner. Which increases the overall latency of the transaction.

Another possible perspective to consider is from a data prefetching point of view. The prefetchers place data blocks in the caches so the thread that needs to access that data only needs to wait few cycles. Timeliness prefetching is a desired property to get the maximum performance of a prefetch, as bringing the data too early can increase the miss ratio of caches and requesting the data too late can lead to sub-optimal performance. In

the same way, Timeliness prefetching can be applied to AMOs, so threads can perform near AMOs without invalidating other threads copies too early. Nevertheless, we left the design of a AMO prefetching mechanism for a future work and we focus on selecting the ideal policy.

6.3 Dynamic AMO Predictor Heuristic

Inspired by the profuse use of heuristics in the field of Prefetching (i.e. Next Line, Stream, etc.) and Branch Prediction (i.e. Local Branch History), we have created our own heuristic to select the best policy. We have considered several insights we have obtained from the evaluation of the static policies.

In LockHammer, we have seen the behavior of the static policies when a thread wants to perform several AMOs to the same address in a short period of time. In these cases, threads that fetch the cache line and perform the AMOs in the L1 cache are more efficient than those that issue far AMO transactions. The cost of a single far AMO is similar to the cost of a cache miss, so assuming the cost of one cache miss to perform several AMO is lower that issuing several far AMOs. Therefore, when a thread fetches a cache line we want to count subsequent near AMO accesses that hit in the L1 cache. We can use an online algorithm to compute the average number of uses of a cache line on the fly. But for the final heuristic we will simply count the total amount of near AMOs.

In Splash-3, we observe different access patterns inside a single application. Some highly shared locks are updated only once every several cycles. In those cases, bringing the cache line does not pay off because the thread does not reuse the cache line. If the target cache line of an AMO is present in the HN, issuing an AMO is slightly cheaper than fetching the block. Because the data sent is smaller (quad word instead of a cache line). Moreover, if all threads issue far AMOs to a block instead of fetching it, we can save those invalidation messages required to move a block between caches. Therefore, in those cases in which a thread has the same amount of near AMOs to a cache line than invalidation messages, we know that the thread is not reusing the fetched block.

Next, we decide the static AMO policies to be considered in our predictor.We have discarded All Far policy because we have seen previously that it has the lowest performance always. All Near policy

performs well in most scenarios, so we have decided to select this policy for those addresses that have some reuse patterns. We also know that Present Near, Shared Far or Unique Near can work in those cases in which we want to perform far AMOs. However, Present Near and Shared Far are very similar to All Near policy. Thus, we have selected Unique Near policy because it has more states in which a far AMO is issued instead of a fetch request to perform the near AMO.

Consequently, we already have two policies that are represented as two states in a Finite State Machine (FSM). We have decided that the initial state is All Near, because we know that this policy outperforms Unique Near policy in single thread experiments. Next, we have to create the transitions between these two states. We know that counting for the amount of near AMOs and invalidations of a cache line can help us decide between near and far AMOs. We have opted for a simple design in which a we use a single metric computed as:

$$Ratio = \frac{NumNearAMO}{NumInvalidations}$$

If the rate is below one we know we have executed less near AMOs than received invalidations. Thus, we are not taking advantage of the cache line being in the L1 cache. Note, that we can have higher number of invalidations than near AMOs because a store that fetches the block in UC/UD state can receive latter an invalidation. When the ratio is higher than one, the thread executes more near AMOs than received invalidations. So, in some cases after fetching a cache line the thread executes a near AMO.

Therefore, we will use this ratio to transit between All Near and Unique Near states. Figure 6.4 illustrates the FSM used in the predictor. We have two states and two transitions. We transit from All Near state if the ratio is below a given threshold. We will transit from Unique Near to All Near if the ratio is higher than the same threshold.

6.4 DynAMO Architecture

Once we have defined our heuristic to select the best policy, we have to develop the hardware to apply the heuristic to addresses accessed by AMOs.

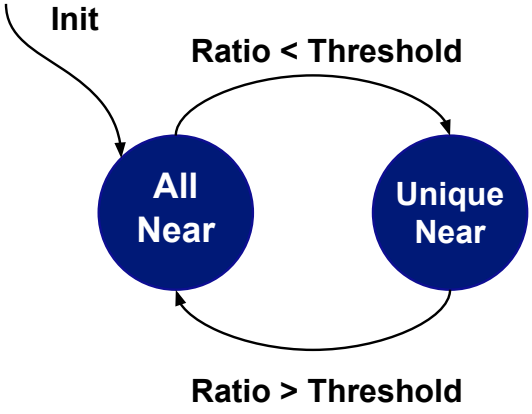

Figure 6.4: Dynamic AMO predictor Finite State Machine.

Inspired again by Branch Predictors we have opted for a look-up table to store the FSM metadata because we only want to apply the heuristic to a small set of addresses. The size of the look-up table is limited to a small amount of entries to limit its hardware cost. We have measured the amount of different cache lines that are accessed by AMO instructions in Splash-3 benchmarks and we have found that 90% of AMO accesses target less than 256 different addresses. Therefore, we have sized our look-up table with 1024 entries.

Since the amount of entries is smaller than the whole address space, we can address the table with the lower bits of the address. To ensure that the accessed entry matches the whole address we have to store a TAG field that contains the remaining bits of the address. We have also implemented a parametric associativity in the look-up table to avoid possible conflicts between addresses that are indexed in the same entry/set. The final design is shown in Figure 6.5.

To figure out which associativity achieves the highest performance we have executed our model with different associativity levels: 1, 2, 4, 8 and 16, keeping constant the total amount of entries. We have evaluated these

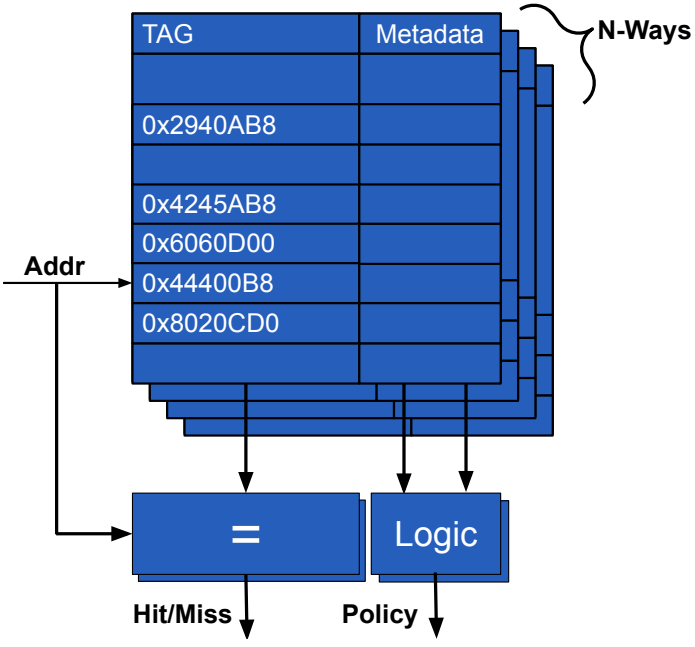

Figure 6.5: Block diagram of the Dynamic AMO Policy Predictor based on a look-up table

models in the next section.

6.5 Evaluation

In this section we evaluate the Dynamic AMO Predictor (DynAMO) we have designed. We have evaluated the predictor in the same environment we tested the static AMO policies and with the same benchmarks (LockHammer and Splash-3).

6.5.1 DynAMO Results with LockHammer

Figure 6.6 compares the performance of the Dynamic AMO Policy Predictor compared against the other static AMO policies. We have computed the speed-ups with respect the execution times with of All

Near policy. All versions execute the benchmark with 64 threads and *serialized* configuration. We have not tested the different associativities because in this benchmark there is a single address that is accessed by AMOs. We can see that our predictor in most cases has the same performance of the Unique Near policy. Only in HSF-Lock we can see that the predictor behaves like the All Near policy. On average the predictor is a 25% slower than All Near Policy.

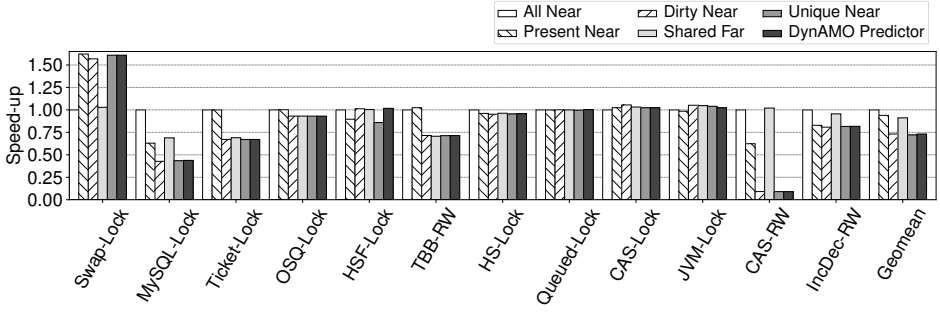

Figure 6.6: Speed-up of LockHammer locks (w.r.t LL/SC version) on with 64 threads using *serialized* configuration

We have investigated why the predictor in most cases behaves like Unique Near, instead of adapting to the kernels that benefit more from All Near policy. We have concluded that in the first stages of the experiments there is high contention that makes the predictor select the Unique Near policy. This contention happens because threads use the fastpath of the lock implementations that just try to capture the lock before checking its value. Once the predictor has selected Unique Near state, the predictor is unable to transition back to All Near state. This may happen because three out of five states choose to execute far AMOs (I, SC and SD). So, the amount of near AMOs observed by the predictor is low. Thus, we still have some room for improvement in future designs.

6.5.2 DynAMO Results with Splash-3

Next, we have tested our predictor with the different levels of associativity on Splash-3. Figure 6.7 shows the relative performance of our predictor

with five different associativities: 1, 2, 4, 8 and 16. In the figure, we have computed the speed-up with respect to All Near policy.

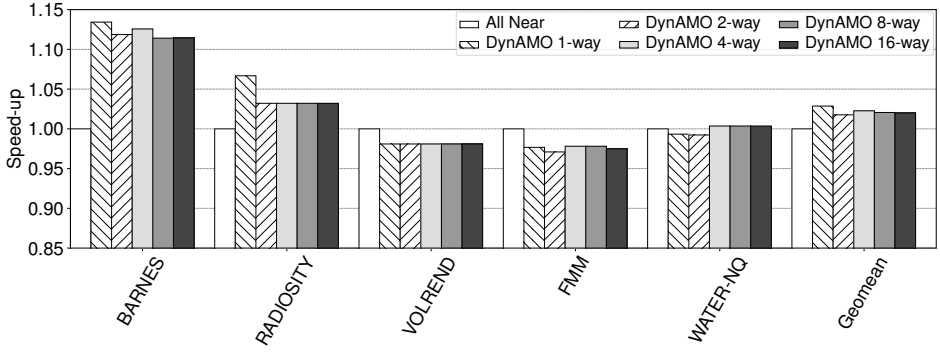

Figure 6.7: Speed-up of Dynamic AMO Policy Predictor running Splash benchmarks with assocciativities 1, 2, 4, 8 and 16 (w.r.t All Near)

We can see that our predictor beats All Near policy in Barnes and Radiosity benchmarks with a 1.11× and 1.3× speed-ups respectively. However, in Volrend and FMM we have a slowdown of 3%. In the case of Water-Nsquared, we have a minimal slowdown in the direct map and 2-way associativity predictors and a speed-up of 1% for higher levels of associativity. The best configuration for our predictor is surprisingly the direct mapped. We obtain with this configuration a 3% speed-up on average and up to 14% speed-up in Barnes. The case in which direct mapped configuration obtains the highest difference against the other predictor configurations is Radiosity, in which this configuration achieves a 7% speed-up while the other configurations achieve a 3.5%. The only case in which higher levels of associativty achieve higher speed-ups is Water-Nsquared.

This kind of behavior is rare because traditionally higher levels of associativity in caches and other structures translate into speed-ups. The main reason to use higher levels of associativity is the reduction of collisions that allow a better track of unique addresses. But in this case, a higher amount of collisions means a higher performance. After studying the traces of the predictor we have been able to determine that this improvement in performance is caused by a better adaptation to the

different phases of the applications.

We have observed that most applications have different parallel regions that have different data sharing pattern for the locks. In these cases, we can experience a degradation in performance when changing from a phase to another. The reason why the predictor cannot adapt to the new phases fast is that the counters we use to compute the heuristic ratio have such big values that it takes a big amount of cycles to transition to a new state.

In direct mapped configuration, it is easier that a new address evicts an already existing entry. This way, by allocating a new entry for the evicted block we have effectively reset the counters used to compute the heuristic ratio. Therefore, the collisions help us to achieve a higher speed-up by resetting the stats of the old entries. We can take advantage of this insight implementing some mechanism that performs this reset in a controlled way. However, we leave this new task for future work.

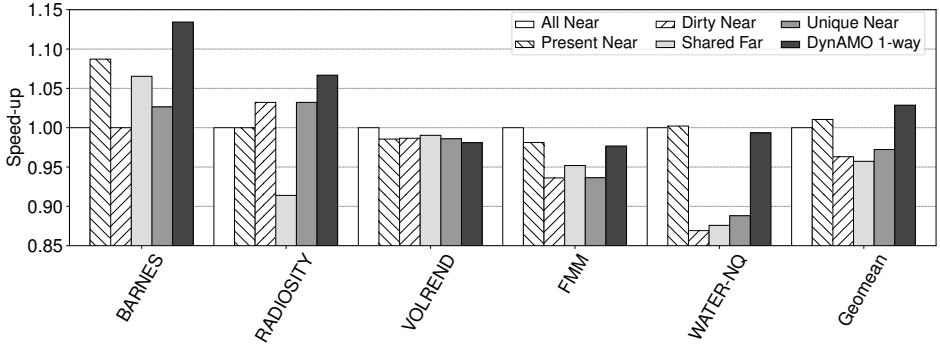

Figure 6.8: Speed-up of Dynamic AMO Policy Predictor (Direct Mapped) running Splash-3 benchmarks (w.r.t All Near)

Finally, we compare the best predictor configuration we have found (Direct Mapped) against all other static AMO policies (see Figure 6.8). Again we have computed the speed-up with respect to All Near policy. We can see that our predictor outperforms all other policies on Barnes and Radiosity. Moreover, the predictor is on average the fastest among all models.

6.6 Concluding Remarks

We have designed, developed and evaluated a policy predictor to select dynamically the best AMO policy. With our simple design we have obtained a speed-up of 3% on Splash-3 applications, with up to 1.15× in one benchmark. These results encourage us to develop a better AMO policy predictor that can also take advantage of prefetching and stashing mechanisms to speed-up AMOs.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In the first part of this work we have presented an exhaustive analysis of two State-of-the-Art synchronization directives: Load Link / Store Conditional and Atomic Memory Operations. This analysis has covered two benchmarks suites (LockHammer and Splash-3) in two commercial machines (Kunpeng 920 and Graviton 2).

We have observed that most kernels in LockHammer benchmark reduce their execution time when replacing LL/SC by AMOs. We observe this behavior in both machines, AMOs outperform LL/SC by 1.2× in Kunpeng 920 and a 3× in Graviton 2. AMOs work better than LL/SC in high contention scenarios like this microbenchmark suite. During these experiments we have detected that AMOs have a higher cost than LL/SC in Kunpeng 920 architecture.

We have observed with Splash-3 applications that each machine behaves differently in terms of performance scaling for these synchronization instructions. Kunpeng 920 usually scales better using LL/SC instructions. Binaries using AMOs slowdown applications by up to a 20% and a 5% in average compared to binaries compiled with LL/SC. However, AMOs outperform LL/SC in Graviton 2 achieving a 25% speed-up in some applications and an average speed-up of 5%.

In the second part of this thesis we have followed AMBA 5 CHI specifications to develop six different multicore models in gem5. Each model follows a different static AMO policy that executes AMOs in the L1 cache or in the L3 cache for different cache states. We evaluated these

six policies with LockHammer finding that the All Near policy (policy that always executes AMOs in the L1 cache) is the most efficient policy. However, when executing Splash-3 benchmarks we have observed that there are other policies like Present Near or Unique Near that can execute faster some applications. These policies in some cases can issue a far AMO to execute the operation in the L3 without fetching the catch block into the L1 cache of the requester. All in all, we cannot find a static policy that outperforms the rest for all Splash-3 applications

In the third and final part we design, develop and evaluate a simple Dynamic AMO Policy predictor. This predictor selects between All Near and Unique Near policies using a simple heuristic based on the amount of near AMOs performed in the past and the number of invalidation messages received. With this predictor, we are able to achieve speed-ups of 15% in Barnes and 6% in Radiosity. Although average speed-up are modest (3% on average), we believe there is room for improvement.

7.2 Future Work

In a near future we plan to expand this work following the next steps:

- Extending our experiments to some domain specific applications that use AMOs extensively. Graphs analytic applications have an important role in many industrial and research areas. Many of the graph analytic frameworks use AMOs to update sparse data structures. Therefore, studying the pattern accesses of graph applications and devloping a specific AMO policy can speed-up these important applications.

- Improve our predictor design to capture better the different phases of the programs. As we have seen in our experiments the direct mapped predictor outperforms all other models that use associativity. We know that collisions in the AMO policy predictor trigger the eviction of some entries. These evictions act as a reset mechanism and help the predictor to adapt to the different phases of the application. Therefore, we plan to implement a reset mechanism to improve the performance of our predictor.

- Explore the use of prefetchers and other kind of prediction mechanism to improve AMOs execution. As we have discused in Chapter 6, selecting wheter the AMOs execute in the L1 cache or in the L3 cache has an effect similar to prefetching. Each cache block has a different reuse pattern and fetching the block to the L1 caches can have a big impact in performance. Therefore, we want to develop a AMO friendly prefetcher in the future.

7.3 Publications

Our goal is to publish the results obtained in this Master Thesis in a relevant computer architecture conference. We plan to enhance the DynAMO policy and submit a paper in the next few months.

In addition, during the life span of this project I have been involved in other publications:

- J. Abella, C. Bulla, G. Cabo, F. J. Cazorla, A. Cristal, M. Doblas, R. Figueras, A. González, C. Hernández, C. Hernández, V. Jiménez, L. Kosmidis, V. Kostalabros, R. Langarita, N. Leyva, G. López-Paradís, J. Marimon, R. Martínez, J. Mendoza, F. Moll, M. Moretó, J. Pavón, C. Ramírez, M. A. Ramírez, C. Rojas, A. Rubio, A. Ruiz, N. Sonmez, **V. Soria**, L. Terés. O. Unsal, M. Valero, I. Vargas, L. Villa, "An Academic RISC-V Silicon Implementation Based on Open-Source Components." Conference on Design of Circuits and Integrated Systems (DCIS), 1-6 2020. Paper accepted [1].

- **V. Soria Pardos**, A. Armejach, D. Suárez Gracia, M. Moretó, "On the use of many-core Marvell ThunderX2 processor for HPC workloads. Journal of Supercomputing, 77(2):3315–3338, 2021. Paper accepted [41].

Bibliography

[1] J. Abella, C. Bulla, G. Cabo, F. J. Cazorla, A. Cristal, M. Doblas, R. Figueras, A. González, C. Hernández, C. Hernández, V. Jiménez, L. Kosmidis, V. Kostalabros, R. Langarita, N. Leyva, G. López-Paradís, J. Marimon, R.o Martínez, J. Mendoza, F. Moll, M. Moretó, J. Pavón, C. Ramírez, M. A. Ramírez, C. Rojas, A. Rubio, A. Ruiz, N. Sonmez, V. Soria, L. Terés, O. Unsal, M. Valero, I. Vargas, L. Villa, and C. Ramííez. An academic risc-v silicon implementation based on open-source components. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2020.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[3] Anandtech. Graviton 2 architecture details. `https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd`, 2019. [Online; accessed 2-December-2021].

[4] Anandtech. Apple m1 architecture. `https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2`, 2021. [Online; accessed 2-December-2021].

[5] The Next Platform Arm Holding. Graviton 2 module diagram. `https://www.nextplatform.com/2021/03/17/can-graviton-win-a-three-way-compute-race-at-aws/`, 2019. [Online; accessed 2-December-2021].

[6] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal. Techniques to improve performance in requester-wins hardware transactional memory. 10(4), dec 2013.

[7] H.B. Bakoglu and J.D. Meindl. Optimal interconnection circuits for vlsi. *IEEE Transactions on Electron Devices*, 32(5):903–909, 1985.

[8] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[9] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[10] J. M. Broughton, P. M. Farmwald, and T. M. McWilliams. S-1 Multiprocessor System. In Joel Trimble, editor, *Real-Time Signal Processing V*, volume 0341, pages 327 – 332. International Society for Optics and Photonics, SPIE, 1982.

[11] BSC. Extrae: Paraver trace-files generator. `https://tools.bsc.es/extrae`, 2019. [Online; accessed 2-December-2021].

[12] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[13] V. Dimić, M. Moretó, M. Casas, J. Ciesko, and M. Valero. RICH: Implementing reductions in the cache hierarchy. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. ICS '07, page 232–241, New York, NY, USA, 2007. Association for Computing Machinery.

[15] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang. The sunway taihulight supercomputer: system and applications. 59(7), 2016.

[16] gem5 and Arm Holdings. gem5 chi protocol. `https://www.gem5.org/documentation/general_docs/ruby/CHI/`, 2021. [Online; accessed 30-December-2021].

[17] E. J. Gómez-Hernández, J. M. Cebrian, R. Titos-Gil, S. Kaxiras, and A. Ros. Efficient, distributed, and non-speculative multi-address atomic operations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 337–349, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Gottlieb, Grishman, Kruskal, McAuliffe, Rudolph, and Snir. The nyu ultracomputer—designing an mimd shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, 1983.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery.

[20] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[21] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote store programming. In *High Performance Embedded Architectures and Compilers*, pages 3–17, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] Arm Holdings. Exlusive monitors. `https://developer.arm.com/documentation/100934/0100/Exclusive-monitors?lang=en`, 2021. [Online; accessed 2-December-2021].

[23] Arm Holdings. Lock hammer. `https://github.com/ARM-software/synchronization-benchmarks/tree/master/benchmarks/lockhammer`, 2021. [Online; accessed 2-December-2021].

[24] R.E. Kessler and J.L. Schwarzmeier. Cray t3d: a new dimension for cray research. In *Digest of Papers. Compcon Spring*, pages 176–182, 1993.

[25] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 162–173, New York, NY, USA, 2007. Association for Computing Machinery.

[26] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, pages 665–674, 1996.

[27] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 241–251, New York, NY, USA, 1997. Association for Computing Machinery.

[28] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 48–59, New York, NY, USA, 1995. Association for Computing Machinery.

[29] C. Liang and M. Prvulovic. Misar: Minimalistic synchronization accelerator with resource overflow management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 414–426, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Linux. Taskset command. `https://man7.org/linux/man-pages/man1/taskset.1.html`, 2021. [Online; accessed 2-December-2021].

[31] Linux. Time command. `https://man7.org/linux/man-pages/man1/time.1.html`, 2021. [Online; accessed 2-December-2021].

[32] J. Lowe-Power, A. Mutaal Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. Rodrigues Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani,

P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. The gem5 simulator: Version 20.0+, 2020.

[33] lowRISC. lowrisc untethered project. `http://www.lowrisc.org/docs/untether-v0.2/`, 2015. [Online; accessed 2-December-2021].

[34] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[35] M. Musleh and V. S. Pai. Automatic sharing classification and timely push for cache-coherent systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[36] D. Nagle. Mpi – the complete reference, vol. 1, the mpi core, 2nd ed., scientific and engineering computation series, by marc snir, steve otto, steven huss-lederman, david walker and jack dongarra. *Sci. Program.*, 13(1):57–63, jan 2005.

[37] University of Virginia. Stream2 home page. `http://www.cs.virginia.edu/stream/stream2/`, 2020. [Online; accessed 30-December-2021].

[38] B. Sahelices, P. Ibáñez, V. Viñals, and J. M. Llabería. A methodology to characterize critical section bottlenecks in dsm multiprocessors. In *Euro-Par 2009 Parallel Processing*, pages 149–161, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[39] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016.

[40] S. L. Scott. Synchronization and communication in the t3e multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 26–36, New York, NY, USA, 1996. Association for Computing Machinery.

[41] V. Soria-Pardos, A. Armejach, D. Suárez, and Moretó. M. On the use of many-core marvell thunderx2 processor for hpc workloads. *The Journal of Supercomputing*, 77(2):3315–3338, 2021.

[42] X. Tang, J. Zhai, X. Qian, and W. Chen. Plock: A fast lock for architectures with explicit inter-core message passing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 765–778, New York, NY, USA, 2019. Association for Computing Machinery.

[43] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[44] Wikichip. Kunpeng 920 module diagram. `https://en.wikichip.org/wiki/hisilicon/microarchitectures/taishan_v110`, 2019. [Online; accessed 2-December-2021].

[45] Wikichip. Arm cortex x1 architecture. `https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-x1`, 2020. [Online; accessed 2-December-2021].

[46] C. M. Wittenbrink, Emmett K., and A. Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011.

[47] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[48] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. *IEEE Micro*, 41(5):67–75, 2021.

[49] G. Zhang, W. Horn, and D. Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 13–25, New York, NY, USA, 2015. Association for Computing Machinery.

[50] L. Zhang, Z. Fang, and J.B. Carter. Highly efficient synchronization based on active memory operations. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 58–, 2004.