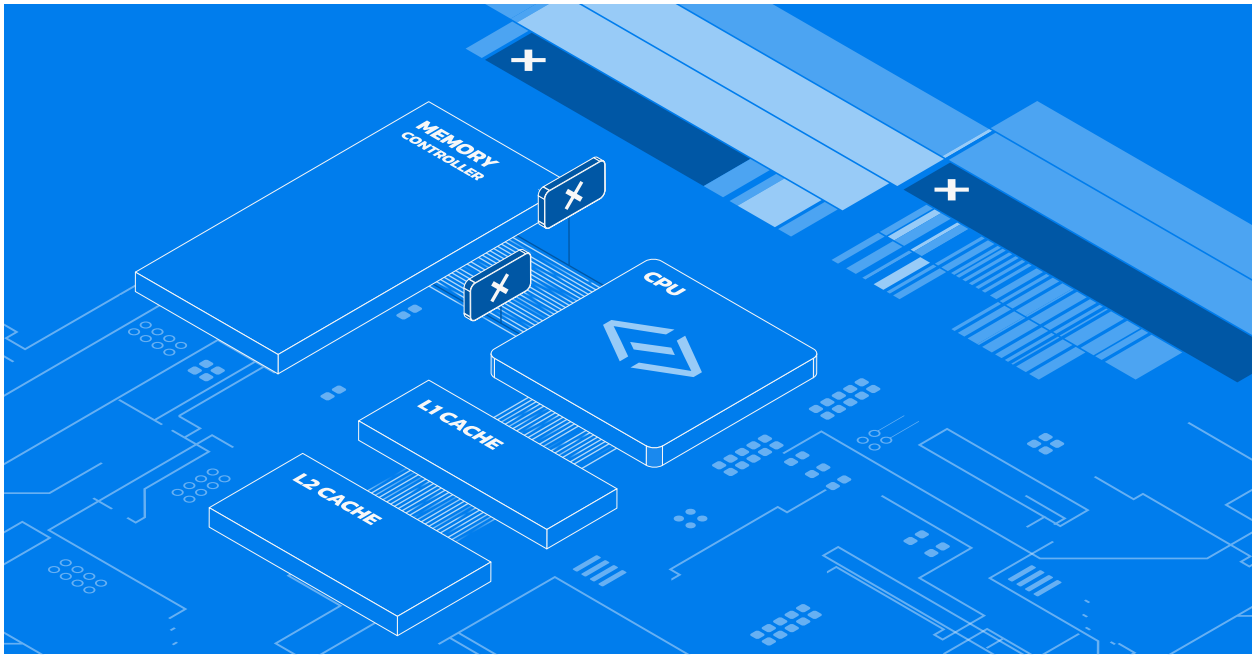
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Preserving Semantics of Multi-Threaded Programs During Cross-ISA Dynamic Binary Translation

Implementing the Hash Table-Based Store-Test in Renode

Master's thesis in Computer science and engineering

MARTIN JONSSON
VALDEMAR VÅLVIK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Preserving Semantics of Multi-Threaded Programs During Cross-ISA Dynamic Binary Translation

Implementing the Hash Table-Based Store-Test in Renode

MARTIN JONSSON
VALDEMAR VÅLVIK

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Preserving Semantics of Multi-Threaded Programs During Cross-ISA Dynamic Binary Translation
Implementing the Hash Table-Based Store-Test in Renode
MARTIN JONSSON
VALDEMAR VÅLVIK

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Piotr Zierhoffer, Antmicro
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A visualization of an emulated system in Renode, with a CPU core accessing memory, courtesy of Antmicro.

Typeset in LaTeX
Gothenburg, Sweden 2025

Preserving Semantics of Multi-Threaded Programs During Cross-ISA Dynamic Binary Translation
Implementing the Hash Table-Based Store-Test in Renode
MARTIN JONSSON
VALDEMAR VÅLVIK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Dynamic Binary Translation (DBT) is a method used to emulate programs on platforms on which they cannot execute natively. In the past, DBTs either did not emulate multi-core programs or did not parallelize their execution. This is no longer the case, as modern processors are often multi-core, necessitating better scaling in DBTs. Renode [1] is one such DBT that is able to emulate multi-core programs using parallel execution. However, Renode — like many other DBTs — fails to correctly emulate the semantics of certain atomic instructions. In particular, emulation of the RISC-V instructions `Load-Reserved` (LR) and `Store-Conditional` (SC) is currently incorrect. These semantics are paramount for program correctness.

In this thesis, we improve Renode's correctness by applying the Hash table-based Store-Test (HST) — a scheme proposed by Zhao et al. [2] — to correctly emulate LR/SC instructions. Using model checking, we find that implementing HST as described by Zhao et al. in Renode results in a race condition. We show how to remediate this race condition in Renode.

Furthermore, we compare the performance of two HST implementations: one written directly in an *intermediate representation* (IR) similar to assembly, the other written in C using *helper functions*. Previous work suggests that IR is faster due to less runtime overhead, which we show holds in this case. We find that the IR implementation is 34% faster than helpers in microbenchmarks and 6–18% faster in the PARSEC [3] benchmark suite.

Our IR implementation of HST in Renode improves both correctness and scalability. We show that our implementation can boot Linux on an embedded platform with multi-core emulation enabled, which Renode in its current state (current Renode) cannot do due to correctness issues. Moreover, our implementation scales well when current Renode does not: in an 8-thread microbenchmark of LR/SC, our implementation is 15.6x faster than current Renode. We find that this scalability can be achieved with as little as 8 KiB of extra memory usage.

Keywords: DBT, HST, Renode, Atomics, LR/SC, LL/SC, Cross-ISA, Qemu.

Acknowledgements

We would like to thank our academic supervisor Erik Sintorn for his consistent and insightful guidance throughout the process, and our company supervisor Piotr Zierhoffer for his helpful and specific suggestions.

Our examiner, Wolfgang Ahrendt, has always been quick to respond to our inquiries — something we very much appreciate.

The Renode team at Antmicro has been a great help. Lilly Jinstrand helped us initially grasp the inner workings of Renode, and Jakub Jatczak has been a reliable and helpful mentor throughout the entire project. Without their guidance, we could not have achieved as much as we did.

The Infrastructure team at Antmicro provided us with an entire cluster of CI-servers, consisting of 8 nodes, which vastly sped up our benchmarking process. It allowed us to iterate and measure more than we otherwise could. We had no expectations of this, and so we are grateful for the resources provided.

Finally, we would like to thank Simon Hultsborn for his detailed, nuanced and constructive feedback on the final draft of this thesis. His comments helped us make the thesis clearer and more accurate.

Martin Jonsson, Valdemar Vålvik, Gothenburg, 2025-06-20

Contents

List of Figures

List of Tables

1

Introduction

Suppose a rocket manufacturing company is preparing to launch its newest rocket model. In preparation for this, the firmware of the rocket's many control units has been rigorously tested. As it would not be economically viable or time-efficient to test it all on real hardware, they use a *digital twin* — i.e. an emulation of the rocket's flight computers — to emulate the rocket during development. Unbeknownst to the rocket engineers, there is an error in the emulator. The error is very subtle, only causing the emulation to misbehave in noticeable ways sporadically. This allows a bug in the propulsion control firmware to pass the automated testing pipeline. Thus, the engineers unknowingly proceed with the launch, resulting in the rocket failing catastrophically, exploding and wasting years of work and millions of dollars. This could have been prevented, had the *digital twin* been a more accurate emulation of the flight computers.

Another scenario could be that the rocket company has a large and thorough test suite, which can catch most bugs before they get deployed. The only issue is that the tests are not entirely deterministic, as they can sporadically fail due to a subtle bug in the emulator. This has sown distrust towards the tests in the development team, and a culture has developed where a failing test is not taken seriously, as it could have been a sporadic failure. Re-running failing tests rather than investigating their cause has become a common occurrence. This could result in either the team ignoring the tests and deploying faulty firmware, or the team wasting much time re-running tests, unsure about whether they fail because of real issues. This, too, could have been prevented, had the emulator been deterministic.

The above hypotheticals refer to nondeterministic execution, which is indicative of one of two things: either the program is implemented using some source of randomness, or the program is multi-threaded and exhibits race conditions. In the case of race conditions, possible causes include incorrect usage of atomic instructions or an incorrectly behaving atomic instruction. Assuming the program is written correctly, the fault may lie in the implementation of the atomic instruction. In an ordinary hardware context, this would be the CPU's circuitry, but in an emulated context this would be the software emulation of the atomic instruction. This is how incorrect emulation of atomic instructions can cause issues with nondeterminism in user programs executed inside the emulator.

An emulator called Renode [1] suffers from these non-determinism issues caused by faulty atomic instructions. Renode performs Dynamic Binary Translation (DBT)

to emulate e.g. a rocket's flight computers on a developer's personal computer. This thesis describes the implementation of one way to improve correctness, reliability and scaling in Renode by improving its atomics.

1.1 Dynamic Binary Translation For Systems Development

DBT is the process of taking a source binary, i.e., code built to execute on a specific Instruction Set Architecture (ISA), and live-translating it into a format that can be executed on a different machine. DBT is necessary due to how different ISAs implement the same functionality in different ways, leading to incompatible binaries. RISC-V and x86 machine code are different languages that can be used to express the same action in different ways, e.g. the instruction to perform a store to memory on RISC-V platforms is `sw` (store word) while on x86 it is `mov`.

A note on terminology: we use the term *guest* to refer to the platform and code that is being emulated, and *host* to refer to the platform and code that is native to the physical hardware. For example, if we emulate a RISC-V processor on an x86 machine, we would call that a RISC-V guest on an x86 host. *Foreign* code refers to code that is not compatible with the host, while *native* code is the opposite.

One common form of DBT is Just-In-Time (JIT) compilers, e.g. Java and its JVM (Java Virtual Machine) [4]. JITs take an Intermediate Representation (IR), bytecode in the case of Java, and translate it during runtime into host-executable code. That way, Java programs can be compiled into a common portable format which can be easily distributed to any platform. These languages define a Virtual Machine (VM) with an accompanying syntax and semantics, which is targeted by compilers to generate portable code and by JITs to convert it to host code. This means that when developing a JIT compiler, one has control over the VM design and can make it suitable for DBT. This is not the case for other forms of DBT, which try to adapt an existing machine language to another.

The core purpose of DBT is to make foreign guest code executable on new host platforms without needing to recompile or adapt the guest binary in any way. Systems development is one notable application of DBT, as it can help speed up the development process. This is achieved by allowing developers to both write and execute code on their own computers. Without DBT (or other forms of code execution, e.g., via interpreters), it is not always possible to write and execute code on the same machine due to the target platform being incompatible with the development machine. This incompatibility is especially relevant for embedded systems, which have low-power processors that utilize different architectures than development machines due to their differing needs for efficiency and speed.

Figure 1.1 shows how guest code gets converted into host code. The process involves intermediate translation steps, which can be implemented by hand using either C or IR. The frontend of the DBT parses the guest machine code, determines which instruction needs to be emulated, then emits a combination of IR code and helper

function calls. The backend of the DBT takes these as input and translates them into natively executable host machine code.

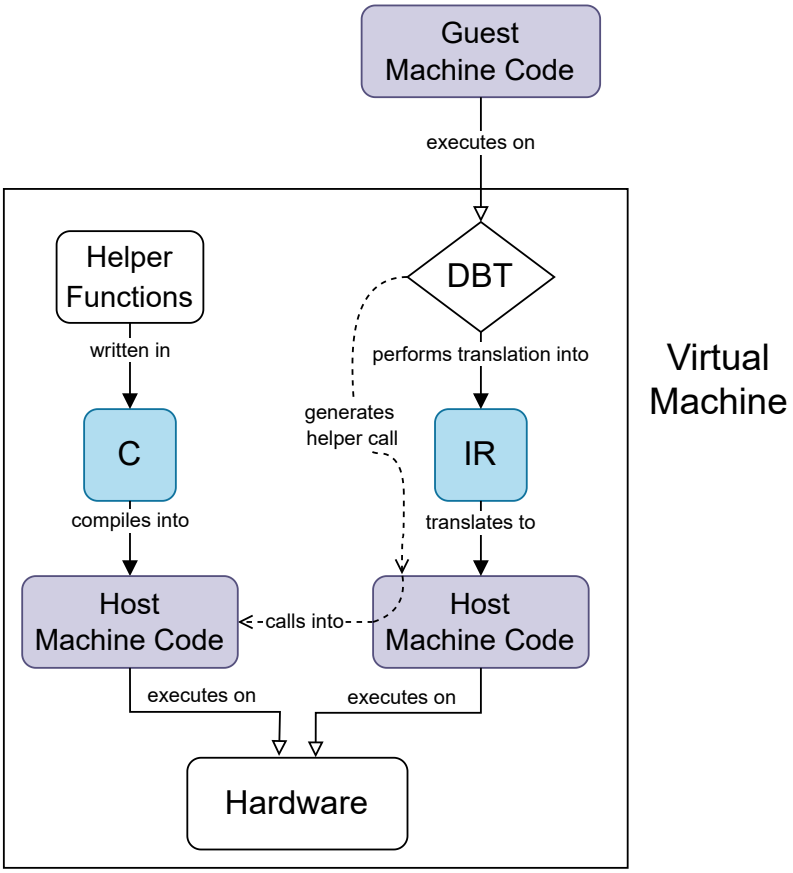


Figure 1.1: The flow of guest code into host code during dynamic binary translation. The blue boxes highlight the two languages used for implementing emulation of guest instructions, while the purple boxes highlight the languages interpretable by hardware.

Note that while there often is a one-to-one direct translation between instructions on different platforms, it is not always the case. For example, on x86 there is an instruction called `addsubps` which

> "Adds odd-numbered single precision floating-point values of the first source operand (second operand) with the corresponding single precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single precision floating-point values from the second source operand from the corresponding single precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand." [5]

and there is no corresponding instruction in neither Arm nor RISC-V. Some instructions, like the above `addsubps`, are highly specific and complex, making it unlikely to have a counterpart on other platforms. Differing platforms adhere to different principles and patterns, making it difficult to translate certain instructions.

Atomic instructions (hereafter referred to as *atomics*) are especially difficult to emulate correctly due to their complex semantics. In concurrent systems, such as multi-core and multi-processor systems, atomics are used for synchronization. If these systems are emulated with faulty atomics, then subtle and transitory bugs may appear due to data races caused by incorrect synchronization.

1.2 Correct and Efficient Emulation of Atomics

An important application of DBT is testing. Tests that are not reliable may even be detrimental to the development process, which is why it is paramount that tests execute deterministically. If the atomic instructions of the guest are not emulated in a way that results in atomic (i.e. uninterrupted) execution on the host, then that may cause the tests to behave nondeterministically.

Another important property of DBT is its speed. Emulation will more often than not be slower than native execution, which is why it becomes important to pay attention to the performance of translated guest instructions. Even in testing, where real-time interactivity is not a requirement, tests should run as fast as possible. This results in quicker feedback and more time left for other tests. Atomics could be emulated correctly by using locks that prevent all other cores from executing concurrently, but this would result in poor performance.

1.2.1 The Problem of Emulating Atomics

There are two main kinds of atomic primitives: Compare-And-Swap (CAS) and Load-Reserved/Store-Conditional (LR/SC). The former is mainly used by Complex Instruction Set Computer (CISC) platforms, such as x86, while the latter is mainly used by Reduced Instruction Set Computer (RISC) platforms, such as RISC-V or Arm. As these atomics have differing semantics, and varying expressive power, it becomes difficult to emulate one on a platform that only supports the other.

CAS consists of comparing a given value against a memory value, and if they are equal the new value is swapped for the existing memory value. A common use case for CAS is to read a value from memory (using an ordinary, non-atomic, load), perform computations based on this value and then check that the originally read value remains unchanged when storing the computed result. If the CAS fails, the whole sequence can be retried by performing the load again and recomputing the result.

LR/SC, as opposed to CAS, consists of a pair of instructions. The first, LR, loads a value from memory. It also, crucially, reserves the memory address to keep track of whether any stores are performed on it. The second, SC, conditionally writes a value to memory in case the reservation performed by LR is still valid. A reservation is invalidated if any other core writes to the reserved memory, or if another SC is performed by the core. These semantics allow for a kind of transaction to occur; it starts by loading a value, reserving its address, and ends with a successful store if the address remains untouched for the duration of the transaction. It is also possible

for invalidations to occur spuriously.

Whenever a reservation is invalidated without a cause, this is called a *spurious reservation invalidation* (hereafter also referred to as a *spurious invalidation*). These happen in real hardware (e.g. due to cache replacement), as well as in emulators. Due to these being a possibility, code that makes use of LR/SC instructions always needs to take into account that the SC may fail, e.g., by retrying the transaction until SC succeeds. Hence, it is safe to assume that guest code safely handles spurious invalidations.

LR/SC subsumes the functionality of CAS: it is more expressive, as it can achieve the same behavior and more. A situation that LR/SC can handle, which CAS cannot, is known as the ABA problem [6]. It is a problem caused by how CAS only relies on comparing the current value, and does not take into account how the value may have been changed and then restored to its original value. This can potentially lead to incorrect behavior in e.g. lock-free stacks implemented using CAS. LR/SC does not exhibit this problem, as it can detect whether the reservation was invalidated even though its original value may have been restored. The ABA problem is explained in detail later in Section 4.2.1.1.

To correctly emulate RISC guests on CISC hosts, one would need to provide some implementation of LR/SC that does not rely on the presence of native LR/SC instructions, as x86 only provides CAS. Translating an LR/SC pair to an ordinary load and a CAS would not be correct, due to the aforementioned ABA problem. Research into possible solutions has resulted in several proposed methods [2], [7]–[13]. In this thesis we implement one of them, known as the Hash table-based Store-Test (HST) scheme [2]. The goal is to achieve an implementation that is both correct and efficient.

1.3 Research Questions

We aim to explore whether implementing the HST scheme [2] in Renode...

(*RQ1*) ...emulates atomics correctly?

We investigate this by implementing the HST scheme [2]. HST is described in Section 4.2, and our implementation of it in Section 4.3. How we evaluate correctness is described in Section 3.1.

(*RQ2*) ...preserves the performance scaling of multi-core guest code when translating?

Multi-core guest code should ideally perform better when emulated using multiple host cores, thus preserving how well the guest program scales with the number of cores. We benchmark our implementations and evaluate how well they scale with more cores, as described in Section 3.2. The performance results are presented in Section 5.2.

(*RQ3*) ...using IR yields better performance than the same implementation in C?

It is a long-standing belief, both within Renode and in prior work [2], that implementations written in IR are faster than those written in C, primarily due to the overhead of calling into C code. However, it is possible that modern compilers can optimize C code to such an extent that this overhead is outweighed by the performance benefits gained through compiler optimizations. To investigate this, we implement the HST scheme using both helpers and IR. How we compare the two implementations is defined in Chapter 3 and their relative performance is shown in Section 5.2.

(*RQ4*) ...is feasible without significantly increasing memory usage?

The scheme proposed by Zhao et al. [2] exhibits spurious reservation invalidations, a phenomenon that is not harmful to correctness but may degrade performance. These can be reduced, but it requires increasing memory usage. This trade-off between memory usage and performance raises the question of how they are correlated and to what degree one is sacrificed for the other. Chapter 4 outlines the implementation details that cause spurious invalidations, with Section 4.3.7 elaborating how we measure their occurrences. The measurements are presented in Section 5.2.3.

1.4 Scope and Limitations

This thesis covers topics of correctness and dealing with different platforms, but it is not all-encompassing. Here, the scope and limitations are defined and outlined.

1.4.1 Correctness

We define correctness as when behavior is functionally equivalent to what is specified by the relevant ISA, not as a cycle-accurate recreation of hardware. That is, recreating non-observable hardware behaviors such as branch prediction and caching is not relevant. It is not a goal of this thesis to emulate guest hardware to that level of accuracy.

The main sources of truth for what is to be deemed as correct behavior are the ISA specifications [5], [14]. As a supplement to this, we make use of official tests such as the Sail tests used by RISC-V International [15].

No formal proof of correctness is provided. Instead, tests are used to gauge correctness. This means no guarantee of correctness is provided, therefore it is not known for certain whether the described implementation is completely correct. For a description of our testing and verification methodology, see Section 3.1.

1.4.2 Platforms

Even though Renode supports many different guest and host platforms, we narrow our focus to RISC-V 32/64-bit guests and x86-64 hosts. The same principles should apply to Arm guests, but other combinations of CISC and RISC platforms are not covered.

Despite the above scope constraint, it remains a goal to keep the solution as general and architecture-independent as possible. Not all behavior can be generalized, as platform semantics vary to some extent.

2

Renode

Renode [1] is an open-source [16] virtual development framework that allows users to emulate embedded hardware systems. Renode enables users to create, test, and debug platform specifications in a virtual environment. These platform specifications include everything from simple system-on-chip devices to multi-node systems. Renode can emulate heterogeneous systems, meaning that multiple processors of different architectures can execute simultaneously. This capability enables the emulation of more complex systems, such as networked Internet of Things (IoT) devices and multi-node environments. Supported guest CPU architectures include RISC-V, Armv7, Armv8, PowerPC and SPARC. Renode is therefore not only a binary translator but above all a platform emulator.

2.1 Renode as a Tool

Emulators like Renode create a digital twin — a virtual representation — of real hardware, recreating its functionality. There are several advantages to this approach.

2.1.1 Architectural Exploration

Hardware emulation allows for pre-silicon development, allowing software to be tested on hardware that does not exist yet. This facilitates early detection of architectural issues, which would be cumbersome and costly to resolve in the post-silicon stage. Antmicro emphasizes that joint development of software and hardware may be useful in areas such as machine learning, where hardware races to keep up with software breakthroughs [17].

2.1.2 Software Development

Emulating embedded systems and their peripherals during the development of firmware and software reduces the need to set up and interact with hardware. It mitigates the constraints posed by limited hardware accessibility and availability, which can be a significant barrier for larger development teams.

2.1.2.1 Debugging and Instrumentation

Debugging hardware directly is often challenging, as tracing code without proper logs or debugging tools is difficult. Even in the cases where such tools are available, simply attaching them to the system may interfere with it. Furthermore, code tracing itself causes the system to behave differently, potentially making it harder to identify root causes.

Renode is aware of the internal state of all system components, allowing it to accurately and transparently trace the execution of a binary and its interaction with peripherals. This allows for a better debugging experience as no modification of the source code or additional debugging hardware is needed, avoiding the aforementioned interference.

Beyond debugging, another use case for Renode's execution tracing is for analysis. One can gain insight into aspects such as cache usage, code coverage and instruction counts by analyzing the execution of a guest program.

2.1.2.2 Automated Testing

Another useful application of emulation is for testing, as part of a continuous integration/delivery (CI/CD) workflow. By executing automated tests in an emulator, the tests can be set up and scaled without additional hardware. Rather than having racks of purpose-built hardware set up specifically for tests, one can use general-purpose servers to execute the emulated tests.

This approach ensures test portability, enabling failed tests on a server to be easily reproduced on a developer's machine for debugging. This benefit is further strengthened by how execution can be made more deterministic in an emulator, as opposed to real hardware. Emulators allow for execution control, eliminating some causes of nondeterminism, such as thread scheduling.

2.2 Renode's Implementation

Renode originally reused the binary translator of Qemu [18] (the Quick EMUlator) called libqemu. The similarities between Renode and Qemu continue to diminish over time, as their distinct use cases and underlying philosophies increasingly diverge. While Qemu aims to be quick, it is less concerned with determinism and execution control. Renode, on the other hand, considers these to be paramount, potentially even sacrificing performance to achieve them. Additionally, Qemu's support for peripherals is rudimentary. In contrast, Renode simulates complete systems, enabling usage of:

- Input/Output (I/O) through peripherals and networking
- multi-node systems with each node having private memory and one or more cores

- multi-core heterogeneous processors with shared memory accessible to each core

Figure 2.1 illustrates some of the components and concepts of Renode, including the parts relevant to this thesis. The following sections will describe these components and their purpose.

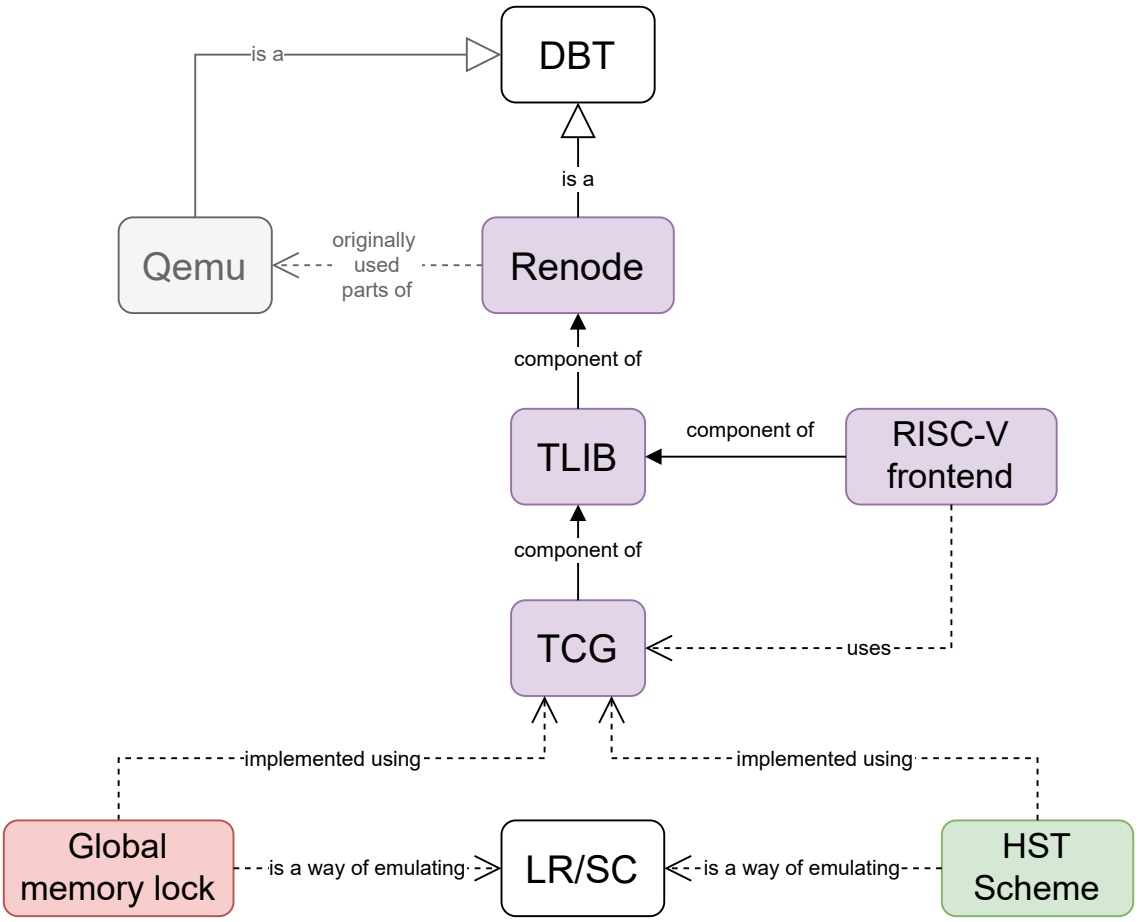


Figure 2.1: An overview of Renode, including its associated components and concepts. Purple marks the components of Renode, while red and green mark the different methods of implementing LR/SC emulation.

2.2.1 Tiny Code Generator

The Tiny Code Generator (TCG) lies at the core of Renode — in a submodule called the translation libraries (tlib) — and is responsible for translation into host code. It originates from Qemu and began as a generic C compiler backend for x86 hosts. TCG exposes an API through its Intermediate Representation (IR).

Renode uses TCG by first translating guest instructions into TCG's IR, which it then hands off to TCG for translation into host instructions. The IR can be viewed as a platform-agnostic assembly code. TCG has its own variables, types and operations that abstract away the low-level details of host register allocation

and host instruction selection. It even performs some basic optimizations, such as liveness analysis and dead store elimination.

Figure 2.2 is an updated version of Figure 1.1 with highlighting showing which components of RENODE fulfill which role in the DBT process. TLIB is where translations of guest instructions are implemented, either using TCG IR or helper functions. TCG implements the IR and translates it into host-executable machine code. The helper functions' C code is compiled into host-executable machine code using the C-compiler GCC.

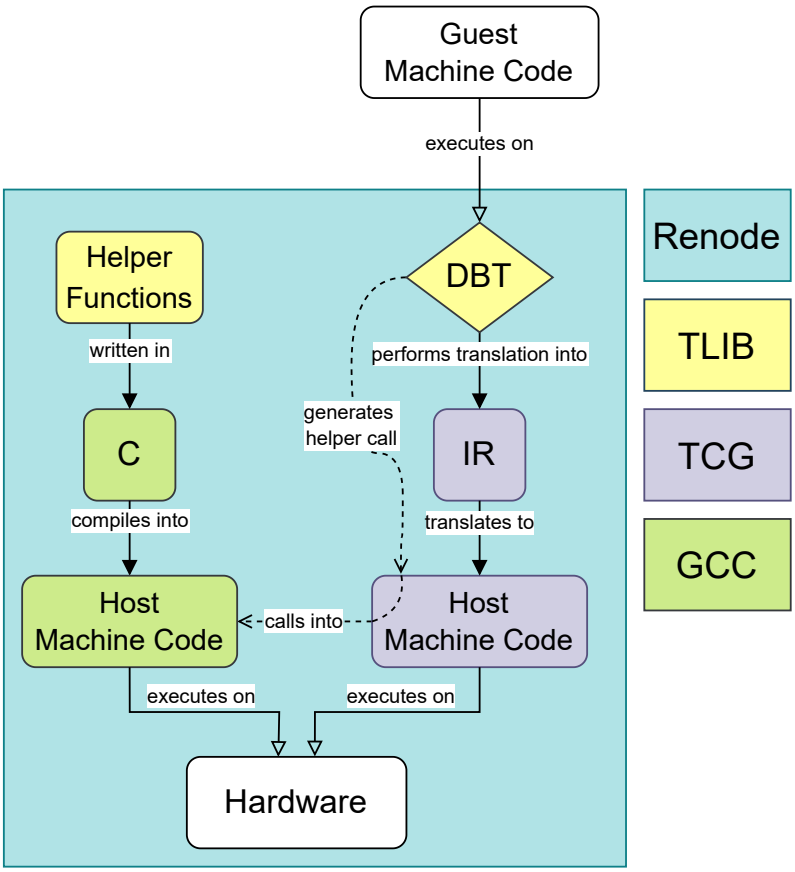


Figure 2.2: A version of Figure 1.1 which highlights the roles different components of RENODE fulfill. It describes the flow of execution through RENODE.

A guest instruction may be translated into several TCG instructions which, in turn, may translate into several host instructions. Using an IR provides the benefit of reducing the number of translation combinations, because rather than implementing a translation between every possible combination of guests G and hosts H, resulting in $G \cdot H$ combinations, it is enough to implement translation from each guest to IR and from the IR to each host, resulting in $G + H$ combinations.

To implement the emulation of a guest instruction within RENODE, one must modify the TLIB frontend corresponding to the guest. This is exemplified by the RISC-V frontend in Figure 2.1. In the frontend, TCG IR instructions are emitted to recreate

the functionality of the original guest instruction, as shown in Figure 2.2 where the DBT performs translation of guest code into IR.

Deciding which sequence of TCG IR instructions to emit for a given guest instruction is difficult, akin to programming in assembly. This is further exacerbated by the fact that TCG is not commonly known, and thus lacks good tooling support. Debugging incorrectly implemented guest instructions is both tedious and time-consuming. Hence, an appealing alternative to implementing guest instructions in IR is writing them in C using helper functions.

2.2.2 Helper Functions

TCG supports a feature called *helper functions* (hereafter also referred to as *helpers*), which consist of plain C code that can be jumped to and executed from a block of translated TCG IR. These allow Renode to support more complex use cases, e.g., cases where the internal state within Renode needs to be modified or cases that are too complex to implement directly in IR.

For example, helpers are used in Renode to acquire and release a global memory lock whenever mutual exclusion is necessary. As helper functions, they can use a standard C mutex instead of reimplementing it in IR. Using this global memory lock is how Renode currently implements LR/SC instructions, as illustrated by Figure 2.1.

Figure 2.2 illustrates the relation between TCG IR and helper functions. The main control flow of the guest program is controlled by the host machine code emitted from IR, but this code may also contain jumps into precompiled helper function code. Once the helper function finishes, control flow is returned to the IR-generated code.

Avoiding helpers has been the norm as the common perception [2] is that calling helpers is slower than executing translated TCG IR. This is because calling a helper involves a procedure call, necessitating the overhead of argument-passing and stack management. Only generating and executing code from TCG IR avoids this and streamlines execution.

However, this perception of helpers being slower than IR lacks experimental support, giving rise to the hypothesis that compiler improvements may have offset the additional overhead imposed by helpers. Consequently, it may be more efficient to generate a procedure call and execute compiler-optimized helper function code rather than handcrafting correct and efficient TCG IR. The overhead of the procedure call may be outweighed by the performance gain of modern compilers being able to produce better code than a developer handcrafting IR.

2.2.3 Time Framework

Time inside a Renode emulation is referred to as virtual time, due to it not always flowing at the same speed as real time. This is organized by Renode's Time Framework [19], which defines time domains that synchronize the execution of their members against a master time source. Different emulated machines make up distinct

time domains, which may contain virtual CPU cores or peripherals. The master time source generates time quantums (quants), which are distributed to time domains to inform them of how much virtual time has passed.

Renode provides the possibility to enforce global serial execution using the Time Framework. In this mode, Renode's master time source only allows one virtual core to execute at a time, in serial. It does this by only distributing quants one at a time, waiting for it to complete before distributing the next one. While the real emulation runtime is longer, virtual time does not pass differently, as each virtual core's time quantum is executed consistently according to the emulation's deterministic schedule. In practice, this is used to ensure deterministic execution of guest code. Global serial execution can mitigate synchronization issues in Renode itself, as each virtual core executes without interleaving.

3

Evaluation Methodology

Concurrent systems are notoriously complex and therefore present challenges in evaluation. The two relevant aspects of the implementation to evaluate are (1) its correctness and (2) its performance. Zhao et al. [2] do not provide a correctness proof of their scheme, and thus it would be a major assumption that it is correct in all situations. Therefore, we need to evaluate the correctness of both their scheme and our implementation of it. Compounding this issue is the fact that this thesis involves making changes to as complex a system as Renode, which is especially difficult due to concurrent semantics correctness. Hence, the need for rigorous correctness and performance evaluation arises.

3.1 Measuring Correctness

Before performance becomes a relevant factor, it is paramount to ensure that the HST implementations behave correctly. Our main avenues of measuring correctness, answering research question *RQ1*, are: (1) running a wide range of test cases that exercise the functionality of the implementation; (2) model checking using the Simple Promela Interpreter (Spin) [20].

3.1.1 Test Cases

A test case passing does not guarantee that the implementation is correct, it only guarantees the absence of specific kinds of incorrectness. Still, tests are useful during development and to prevent future regressions when running in CI. The more repeated test runs pass, the more interleavings are tested, increasing confidence in implementation correctness. This section describes the used test cases, and why they were chosen.

3.1.1.1 LR/SC Contention

LR/SC Contention is a test consisting of two guest cores simultaneously trying to atomically increment a shared-memory variable repeatedly, resulting in high contention and many SC failures. This acts as a stress-test, by increasing the likelihood of interleavings that expose implementation incorrectness.

Both cores execute the assembly loop shown in Listing 1 until they have both successfully incremented it a specified number of times. Once both cores have halted,

the shared-memory counter must have been incremented to $2 * count$. If the counter is lower than the expected result, this indicates a race condition in the implementation.

```
1 repeat: lr.d a4, (a0);      # Begin transaction, load counter
2         add a4, a4, a2;     # Increment shared counter
3         sc.d a4, a4, (a0);  # End transaction, store counter
4         bnez a4, repeat;    # Retry if transaction fails
5         addi a1, a1, -1;    # Decrement loop counter
6         bnez a1, repeat;    # Repeat if loop not done
```

Listing 1: RISC-V assembly implementing an LR/SC loop that increments a shared-memory counter. Registers are initialized as: `a0` is the address of a shared counter, `a1` is how many times to increment and `a2` is the increment (1 in this case). The shared counter is initialized to 0. Register `a4` is a scratch register, used for temporary values.

Figure 3.1 illustrates an error the LR/SC race would detect. At point (a), core 0 executes line 1 of Listing 1, reading the current value of the shared counter using `LR`, then at (b) core 1 interleaves and also executes line 1. The first of them to reach line 3, the `SC`, should cause the other's reservation to be invalidated. If the reservation logic is incorrectly implemented, causing an invalidation to not occur at (c), then it may be the case that they both succeed their `SC` and core 1 does not retry at point (d). This data race becomes visible in the counter, as at least one incrementation is lost due to the overwrite.

The LR/SC Contention test, as described above, tests how LR/SC instructions interact with other LR/SC instructions. It can also be used to test how LR/SC interacts with other atomic instructions, such as atomic fetch-add. This is done by having one core execute the code from Listing 1 while another executes a similar loop which uses atomic fetch-add instead of LR/SC. It is also possible to test interactions with ordinary store-instructions in a similar manner.

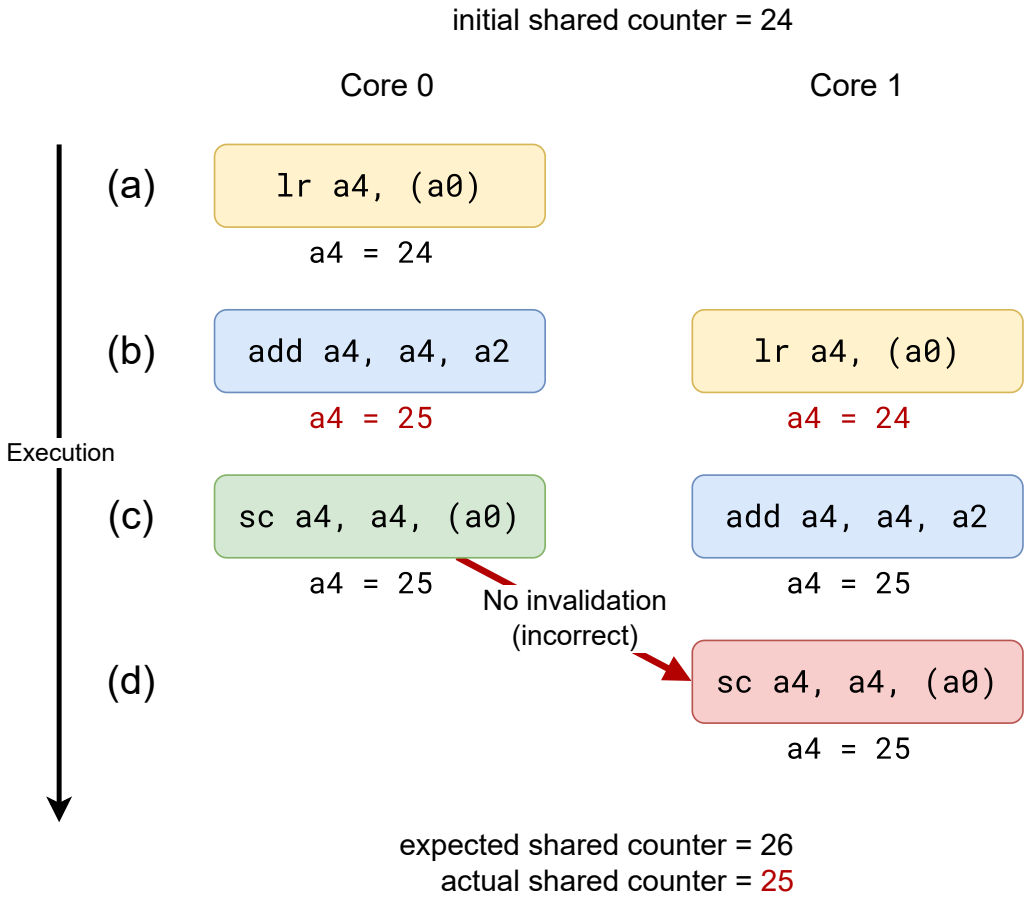


Figure 3.1: An illustration of how incorrectly implemented reservation invalidation logic can result in incorrect behavior. The address of the shared-memory variable is stored in register a0, and the syntax (a0) is used to refer to the value located at that address.

3.1.1.2 RISC-V Sail Tests

The RISC-V Sail Model [15] provides a model for the ISA, defining RISC-V instruction semantics using the Sail specification language [21]. This specification allows for automated instruction-sequence test generation, resulting in test cases that exercise the semantics of specific instructions. These are useful to verify that the semantics of our LR/SC implementation matches the RISC-V semantics.

3.1.1.3 Renode's RISC-V Integration Tests

Renode features an extensive suite of integration tests, encompassing many — though not all — guest behaviors. RISC-V is one such platform, with integration tests for its atomic instructions. These tests are handwritten to verify that certain specified behaviors are emulated correctly by Renode, such as which values the output registers of an atomic instruction should contain after it has executed.

Specifically, the Renode test suite contains tests for LR/SC invalidation behavior. The tests recreate scenarios where another core interleaves between `LR` and `SC`,

performing a store to `LR`'s reserved address and then asserting that the `SC` must fail.

These tests are useful due to their specificity. Whenever incorrect behavior is observed by a test assertion, it quickly and precisely reports the expected and actual value observed.

3.1.2 Model Checking

In model checking, one creates a simplified model of a system. This model can then be rigorously verified against correctness criteria expressed using formalisms such as Linear Temporal Logic (LTL). Note that what is being verified is the model, not the system it represents. As such, model checking can only assert whether the algorithm itself is sound, not whether any implementation of the algorithm is. We make use of model checking to ensure the soundness of the theory behind our implementation, thereby focusing development effort on ensuring that the implementation matches the algorithm.

We create models using the Process Meta Language (Promela) and then verify them with the Simple Promela Interpreter (Spin) [20]. The tool supports verifying two kinds of correctness properties, providing execution traces as counterexamples if verification fails. These properties are:

- Liveness properties, which verify that an assertion must eventually hold. Therefore, a counterexample consists of an infinite execution trace showing a loop in which the property never holds. An example of a liveness property is starvation freedom, i.e., that all non-blocked threads eventually get to execute.
- Safety properties, which assert invariants that must always hold. Thus, a counterexample consists of a finite execution trace showing the steps leading up to, and including, the invariant being broken. A canonical example of a safety property is deadlock freedom, i.e. that the program can always make forward progress.

We create models of five scenarios involving LR/SC instructions and verify them with variants of two main correctness properties. The model source code can be found in Appendix A. The following sections describe the five models and their correctness properties.

3.1.2.1 LR/SC Invalidation by Store

Here, we model the exact implementation of LR and SC as described by Zhao et al. [2]. The scenario is that two cores execute concurrently, where one performs `LR; add 1; SC` on a shared-memory address (retrying if `SC` fails) and the other performs a `store` to the same address. If LR/SC works correctly and the `store` interleaves between the `LR` and the `SC`, then the `SC` must fail and be retried.

The liveness property of this model is that the `SC` must eventually succeed. The safety property of this model is that if both cores have finished executing then the final value of the shared variable must be one of two valid values. The two values were determined through manual verification of possible interleavings.

3.1.2.2 Fine-grained LR/SC Locking

This model is exactly like **LR/SC Invalidation by Store**, except that it does not make use of PROMELA's `atomic {...}` blocks. Those blocks enforce atomicity by definition, acting similarly to a global memory lock. While this implementation may be correct, it would be more efficient to use finer-grained per-address locking. That is what this model aims to verify.

It uses the same correctness properties as the model it is based on.

3.1.2.3 LR/SC Livelock

This model attempts to elicit a livelock by having two cores execute the same LR/SC-loop on the same shared variable. A livelock is much like a deadlock in that no forward progress is made, but it differs in that the cores actually do execute instructions. The cores are not stuck, they are repeatedly retrying the LR/SC instructions ad infinitum.

Livelock is a common issue in the LR/SC instruction family, and therefore ISAs such as RISC-V provide special guarantees to avoid it. RISC-V defines constrained LR/SC loops [14, Section 14.3], to which they bestow a guarantee of forward progress. These constrained loops are defined according to a strict set of conditions, e.g., they must consist of at most 16 instructions and may only contain instructions from the base ISA. It is up to the implementation how to achieve this guarantee. The loops used in these test cases are all constrained LR/SC loops.

The HST scheme does not provide any forward progress guarantee, making our implementation of it susceptible to livelock. The PROMELA model confirms that it is an unsolved issue.

This model uses the same correctness properties as the previous two, with only the two valid values being different.

3.1.2.4 LR/SC Racing

This model aims to recreate the **LR/SC Contention** test scenario from Section 3.1.1.1. It consists of two cores, each performing a sequence of `LR; add 1; SC` on the same shared variable. Note that, as opposed to the test it is based on, this model does not loop. The looping is unnecessary, due to how SPIN exhaustively verifies every possible interleaving from the very start.

The liveness property of this model differs slightly from that of previous models, as it not only necessitates the completion of the `SC`s but also requires that at least one SC completes successfully. It should not be the case that both fail. It may be the case that both succeed, when the two cores do not interleave.

The safety property of this model is more nuanced as well, as it asserts the shared variable must contain differing values depending on if one or two `SC`s have succeeded, corresponding to if it got incremented once or twice.

3.1.2.5 LR/SC + CAS

A noteworthy implementation detail in the method of Zhao et al. [2] — one not mentioned in their paper — is that they make use of a Compare-And-Swap (CAS) instruction in their implementation of `SC`. They did not place it there, rather it is a remnant of Qemu which they did not remove. The purpose of this model is to verify its necessity. We model this implementation separately, to evaluate the effect CAS has on correctness. This scenario is set up similarly to the **LR/SC Invalidation by Store** scenario (Section 3.1.2.1), but with a different LR/SC implementation.

3.1.3 Real-World Applications

Booting Linux is used as an indicator of whether instructions behave as expected, as it is likely that an incorrectly implemented instruction would cause a code base as big as Linux to fail somewhere. It is by no means a certainty that instructions are correct as long as Linux can boot, though, as Linux may have fallbacks in place that are designed to handle such errors gracefully.

3.2 Measuring Performance

Second only to correctness is the performance of the atomics. While it will often be the case that an emulator is slower than actual hardware, the emulation must remain reasonably efficient. Otherwise emulator-based workflows risk becoming slower than the inherently cumbersome hardware-based workflows. To answer research question *RQ3*, we compare the performance of our LR/SC implementation using helpers against the one using TCG IR. This shows whether there is any non-negligible overhead in using helpers, or if compiler-optimized code outperforms handcrafted TCG IR.

The source code of the benchmarks and the associated benchmark harnesses are available in Appendix B.

3.2.1 Microbenchmarking

Microbenchmarks are benchmarks that test the performance of something very restricted and specific, rather than an entire system. This isolates the unit under test from most confounding factors and means the results indicate how much or little it contributes to overall system performance.

To isolate the evaluation of individual instructions, we execute only the submodule of Renode that performs DBT: tlib. It only executes the translation logic, without Renode's UI, platform and peripheral support.

These kinds of tests are unrealistic in the sense that they do not represent real-world performance, but are nonetheless useful indicators of the overhead imposed by our implementations. We microbenchmark LR/SC in two ways.

3.2.1.1 Contended Synchronization Variable

This microbenchmark executes the **LR/SC Contention** test described in Section 3.1.1.1 to evaluate how quickly it increments the counter. By benchmarking this scenario, we gain insight into how well LR/SC performs in the context of high contention. It is a case where many atomic operations will end in failure, due to the other guest core having interleaved and invalidated some state. The failure case is often the slowest one, which is why it is relevant to benchmark.

The results of this benchmark are presented in Section 5.2.1.2.

3.2.1.2 Uncontended Synchronization Variable

This microbenchmark is nearly identical to the above contended microbenchmark, except for which shared variable each guest core uses. To avoid contention, each core is assigned a unique address to repeatedly increment. This, in contrast to the contended microbenchmark, tests the performance of the nominal path, which is expected to be the most efficient path. The purpose of this microbenchmark is to highlight the benefits of fine-grained locking over current Renode's global memory lock.

The results of this benchmark are presented in Section 5.2.1.1.

3.2.2 PARSEC 3.0 Benchmark Suite

The PARSEC 3.0 benchmark suite [3] is a collection of symmetric multiprocessing (SMP) programs that represent different domains such as networking, computational economics and physical simulations. This indicates how our implementation of LR/SC affects real-world performance.

The PARSEC benchmarks need to execute within an OS, as they utilize both threading and standard libraries. To execute them within Renode, we first emulate Linux using the BeagleV-Fire platform [22]. This board provides four 64-bit RISC-V SMP cores. This setup allows us to test Renode implementations executing a Linux-dependent RISC-V guest binary on an x86 host machine. Ideally, the PARSEC benchmarks would have been executed on bare-metal, i.e., without an operating system. However, this would require rewriting their source code.

The benchmark consists of a Robot Framework [23] test, which loads a Renode script. This Renode script, in turn, boots Linux on the BeagleV-Fire platform specified in a platform definition. Once the system has booted into user space, the Robot script executes and measures the runtime of the PARSEC benchmarks included in the guest Linux file system.

The results of this benchmark are presented in Section 5.2.2.

4

Improving Renode's LR/SC Implementation

The main method of LR/SC atomics emulation explored in this thesis is the Hash table-based Store-Test (HST) scheme proposed by Zhao et al. [2]. This chapter describes Renode's current LR/SC implementation and how we replace it with the HST scheme.

All code artifacts are publicly available and can be found in Appendix B.

4.1 Current Renode Implementation

Renode's current LR/SC emulation (hereafter referred to as *current Renode*) is incorrect, due to reservations not being invalidated in cases when they should. It is not an implementation of HST; rather, it is a different ad hoc method similar to the software memory management unit (softMMU)-based approach of [11].

4.1.1 Current Solution Using Global Memory Locks

To emulate guest atomics correctly, current Renode relies on a brute-force solution where threads are synchronized using a global memory lock. If a thread needs to execute an atomic instruction and the lock is occupied, it will sleep until the lock is available. In the case of LR/SC, there is a global shared state that stores information about reserved addresses.

To trigger invalidation of reservations when executing instructions that write to guest memory, Renode's softMMU is utilized. The softMMU performs translation of guest addresses to host addresses. Then, when any type of guest `store` is executed, a helper function for invalidating addresses is called from within the softMMU (synchronized by the global memory lock).

This global lock mechanism is implemented through various helper functions (see Section 2.2.2), necessitating several helper calls in the translated machine code. The overhead introduced by these calls may be negligible; however, the main performance limitation of this solution is the global lock. It causes unnecessary contention between guest stores accessing distinct memory addresses. Moreover, this solution's reliance on the softMMU leads to incorrect behavior, as described in the next section.

4.1.2 The Failed Attempt at LR/SC Emulation

The core idea behind Renode's current LR/SC scheme, using the global memory lock, is sound. Its implementation, however, suffers from correctness issues due to complex interactions with the softMMU. It is reasonable to keep track of reservations in the softMMU, as that is the one place where all guest memory accesses pass through. This is because every guest address must be looked up in the Translation Lookaside Buffer (TLB) of the softMMU. This simplifies the implementation by centralizing all logic related to reservation invalidation.

The problem with the softMMU approach stems from optimization. All softMMU interactions involving a store require the global memory lock, as the global shared state that needs to be updated is guarded by it. To prevent contention on this lock from slowing down Renode, an optimization is made by TCG which allows code to skip the softMMU and index straight into the TLB. This enables guest memory accesses to bypass the global memory lock, once the address translation has been performed previously and is now cached in the TLB.

If the softMMU is bypassed non-deterministically, depending on whether the address is cached and when the TLB is flushed (e.g. due to I/O), this results in store instructions only correctly causing reservation invalidation nondeterministically. Spurious reservation invalidations are permitted by the RISC-V and Armv8-A specifications [14], [24]; however, omitting one (i.e. when bypassing the softMMU) is strictly incorrect.

One way to address the softMMU bypass is to trigger TLB invalidation before instructions that must go through the softMMU (such as stores). However, managing TLB coherence is challenging, as is often the case with cache invalidation logic. Furthermore, even if correctly implemented, this approach incurs performance losses due to contention on the softMMU's global memory locks.

4.2 Hash Table-Based Store-Test as a Solution

This thesis investigates the Hash table-based Store-Test (HST) scheme of Zhao et al. to emulate LR/SC on hosts without native support. They implemented their solution in Qemu and demonstrated that it surpasses previous solutions in both correctness and performance.

4.2.1 The Hash Table, the Store and the Store Test

The HST method relies on, as its name implies, a hash table that is shared between all emulated cores. The table's purpose is to record when, and by whom, stores to memory are performed. Then, by checking the entries in the table, `SC` instructions can be determined to either succeed or fail.

The table is laid out in memory as a contiguous sequence of entries, as shown in Figure 4.1. Unlike general-purpose hash tables, though, there is no field for the key. As there is no need to support different kinds of keys, the table can be specialized to

treat memory addresses as keys. Instead of key and value fields, each table entry contains two values:

- a 32-bit[1] core ID representing which core last accessed the memory address associated with the entry
- a 32-bit[1] fine-grained lock that guards write-access to the first field

The key of the entry is the address at which the start of the entry is located, i.e., an address within the confines of the hash table.

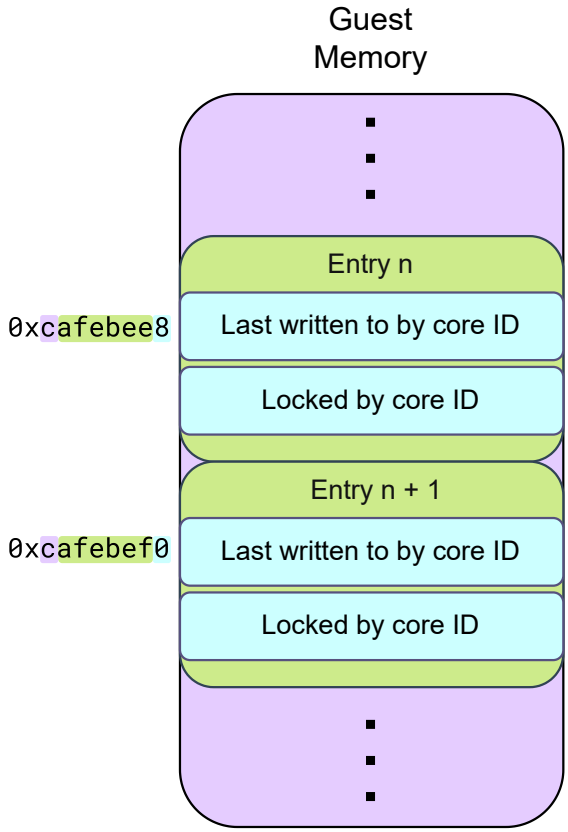


Figure 4.1: The layout of the hash table entries. Each entry is keyed by the address of its first field.

The table needs to be initialized in a specific way, depending on implementation details, as will be described in Section 4.3.4.

Whenever a guest memory write occurs, i.e., a guest store instruction, the table needs to be updated. The guest address of the store is hashed using a specialized hash function (explained in detail in Section 4.2.3), to determine which table entry to update. Then, the core ID field is updated with the ID of the core performing the store. Note how the fine-grained table lock is not utilized, nor respected, by this operation. That is because no read-modify-write (RMW) operation is happening in this case, it is only a single store.

[1]The fields are 32-bit due to alignment and atomicity constraints.

The table entry fields can be updated using ordinary store instructions, rather than using slower atomic instructions. This is because stores to aligned memory locations are atomic in x86's Total Store Ordering (TSO) memory model [5], the RISC-V Weak Memory Ordering (RVWMO) model [14] and the Armv8-A memory model [25]. It is known as a single-copy atomic operation, meaning that if a `load` and a `store` occur simultaneously at the same location, the `load` observes either the entire new value or the entire old value. No in-between values are ever seen.

Which core last updated a memory location is kept track of using the hash table, and the information is used to implement correctly behaving `LR/SC`. When a core executes an `LR` instruction, that address is marked as reserved for that core. In RISC-V and Armv8-A a single core may only hold one active reservation at a time. Furthermore, the `LR` updates the hash table with the current core ID to mark the address as reserved. Later, the `SC` will use this fact.

The `SC` instruction can be implemented by first acquiring the fine-grained table entry lock, using a spinlock, followed by a check of the core ID field. If the ID stored there matches the ID of the core executing the `SC` instruction, the store can succeed. Otherwise, it must fail. In any case, the reservation is invalidated and the core ID field of the table entry is updated with the current core ID. Finally, the table entry lock is released.

The fine-grained table entry lock helps prevent data races between multiple cores executing `SC` concurrently, as it handles the case when two cores have reserved the same address and try to perform `SC` simultaneously. Only one core should successfully perform its store, but without the lock both may succeed. The mutual exclusion provided by the fine-grained locking prevents this incorrect behavior.

4.2.1.1 How HST Solves The ABA problem

A scenario that LR/SC can handle — which CAS cannot — is the ABA problem [6]. The problem arises because CAS only checks whether a value has changed, without detecting if the value was changed and then reverted to its original state. This section outlines a scenario where CAS usage causes correctness issues in this manner.

Consider the lock-free stack in Figure 4.2, implemented as a linked list. The top points to node A; A points to B; and B points to C. Thread 1 loads the top pointer, which points to A. Then, thread 2 interleaves and pops nodes A and B from the stack, updating the top pointer to point to C. Thread 2 then pushes node A back onto the stack, leaving the top pointing to A and A pointing to C. Finally, thread 1 performs a CAS, verifying that the top is still A (which it is), and swaps it with another value. The CAS does not detect the underlying change, even though it has affected the stack structure in unexpected ways (e.g. losing track of a node). An LR/SC, however, would not have succeeded in its check, as thread 1's reservation would have been invalidated by thread 2 mutating the top pointer.

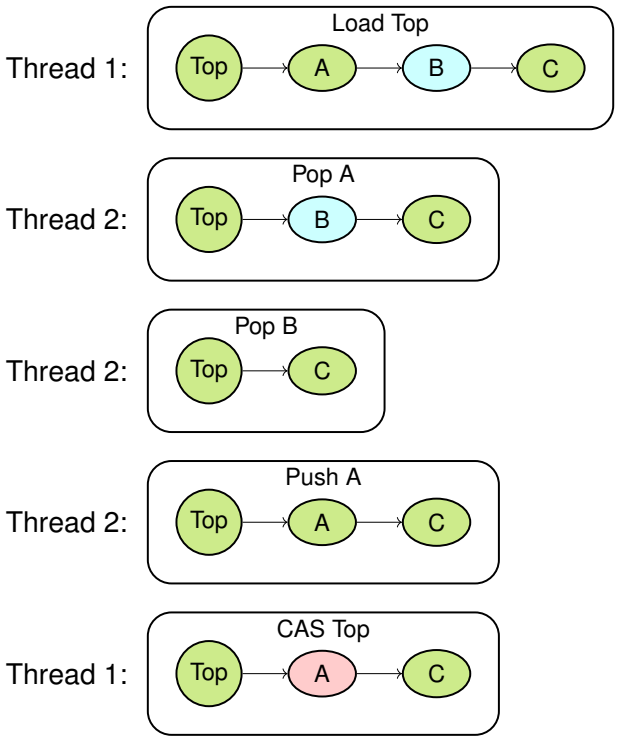


Figure 4.2: An illustration of how the ABA problem can be encountered in a lock-free stack.

HST solves the ABA problem. With HST, thread 1 performs a hash table lookup and detects that the top pointer has been mutated by thread 2, causing the operation to fail. This mimics the semantics of LR/SC without using the intrinsic itself.

4.2.2 Weak vs. Strong Atomicity

Blundell et al. [26] present two notions of atomicity: weak and strong. Their work is specifically referring to transactions, but the LR/SC construct can be viewed as a transaction and thus the same reasoning applies.

4.2.2.1 Weak Atomicity

An atomicity model is weak if it only guarantees atomicity between transactions. In the context of LR/SC, this corresponds to a guarantee of atomicity between LR/SC pairs and other LR/SC pairs.

In a weak atomicity model, reservations made by `LR` instructions can only be invalidated by other `LR` or `SC` instructions. Hence, an ordinary store to an address reserved by `LR` would not be considered invalidated under this model. This clashes with the definition of LR/SC, as it requires reservations to be invalidated by any store [14], [24]. Thus, implementing a weakly atomic version of LR/SC would not be correct according to the ISAs.

4.2.2.2 Strong Atomicity

Strong atomicity subsumes weak atomicity, augmenting it with the guarantee that atomicity is upheld with respect to non-transactional code. In the context of LR/SC, this means that any kind of store instruction can cause the reservation to be invalidated. This includes atomic stores, such as fetch-and-add or compare-and-swap, and non-atomic stores.

To correctly implement strong atomicity for HST, every guest instruction that performs a memory store needs to be instrumented to update the table. This is necessary because strongly atomic (i.e. correct) LR/SC requires that `SC` must fail if the `LR` is followed by a store from another core before the `SC` executes. For the `SC` implementation to be aware of the store having occurred, it requires the HST table to be updated by the store.

4.2.2.3 Weak vs. Strong HST

The reason one would want weak atomicity in the context of LR/SC is that it allows for a more efficient implementation. Zhao et al. describe a variant of their HST solution, called HST-weak, which does this. They analyzed the binaries of PARSEC [27] and the Linux kernel, from which they conclude that certain usage patterns could be taken advantage of. The patterns show that shared-memory variables are only ever modified using atomic instructions, or non-atomically by a thread that owns a lock.

HST-weak can therefore safely avoid instrumenting ordinary store instructions, based on the aforementioned conclusions. The benchmarks performed by Zhao et al. show that HST-weak greatly outperforms HST-strong in benchmark programs that encounter many store instructions during execution. The same assumption cannot be made in Renode, as it aims for correctness in any scenario — including ones where weak atomicity is insufficient.

Zhao et al. only use the fine-grained table entry lock in their HST-weak implementation, instead opting for a global memory lock in HST-strong. This is necessary in Qemu, because other atomics and store instructions only respect the global memory lock and not the fine-grained table entry lock.

In our HST-strong implementation, we make use of the fine-grained lock that Zhao et al. use in their HST-weak implementation, rather than use a global memory lock. This lets us scale better, as contention is reduced by the fine-grained locking.

4.2.3 The Hash Function

The HST-strong scheme tracks guest memory stores. This means that every store needs to be instrumented at runtime to update the table, which may impact the performance of guest code that writes to memory frequently. To alleviate this potential performance issue, Zhao et al. propose a simple and efficient hash function, which can be computed using only a few machine instructions.

The idea behind Zhao et al.'s hash function is to use the guest address to construct the address of the table entry. The table is located at an arbitrary address in guest memory, with a pointer to the start of the table being shared between all guest cores. Thus, to address individual table entries, it becomes necessary to compute where the table entry of that guest address lies in memory (i.e. hash the guest address). Note that one drawback of placing the hash table in guest memory is that its contents are visible to the guest, even though it is an implementation detail of the emulator. In our implementation, we opt for allocating the table outside of guest memory, thereby hiding its presence from the guest. How this can be implemented is explained in Section 4.3.

Zhao et al. fix the size of their hash table to 256 mebibytes and detail the hashing specifically for 32-bit address spaces. We show how to generalize this, to support an arbitrarily sized table, later in Section 4.3.3. Furthermore, the table size affects the number of hash collisions, and thereby spurious reservation invalidations, that occur. This, and its effect on performance, is presented later in Section 5.2.3.2 and then discussed in Section 6.1.1.

4.3 Our Implementation of HST

We have implemented a variant of the proposed HST scheme from Section 4.2, replacing Renode's previous incorrect LR/SC implementation as described in Section 4.1.2. This section describes how it is implemented, and how it differs from the original HST scheme.

4.3.1 Hash Table Memory Layout

The hash table — which keeps track of what core last wrote to a given guest memory address — is the backbone upon which the HST scheme is built. A core tenet of its design is its efficiency, and thus its memory layout is highly relevant.

Figure 4.3 illustrates the memory layout of the hash table. Unlike Zhao et al. we place it in host memory to avoid unnecessary address translations. As it lies in host memory, it is not visible to, nor accessible by, the guest. Note how the addresses are aligned. Both the hash function and the atomicity of the table operations rely on it being correctly aligned. The hash function assumes each table entry field is aligned, allowing it to mask out certain bits to access specific fields. The ISA of the host processor (i.e. x86) only guarantees atomicity of ordinary reads and stores if the accesses are naturally aligned [5].

The lower portion of Figure 4.3 exemplifies the translation of guest addresses to host addresses using the hash function described in Section 4.2.3. First, the guest address must be hashed and then that hashed address can be used as a host pointer to the correct table entry. This mapping is highlighted using colors in the figure, with the purple indicating bits associated with the hash table itself, green indicating guest address bits and cyan indicating table entry fields.

Unlike a general-purpose hash table, collisions are handled differently. A hash

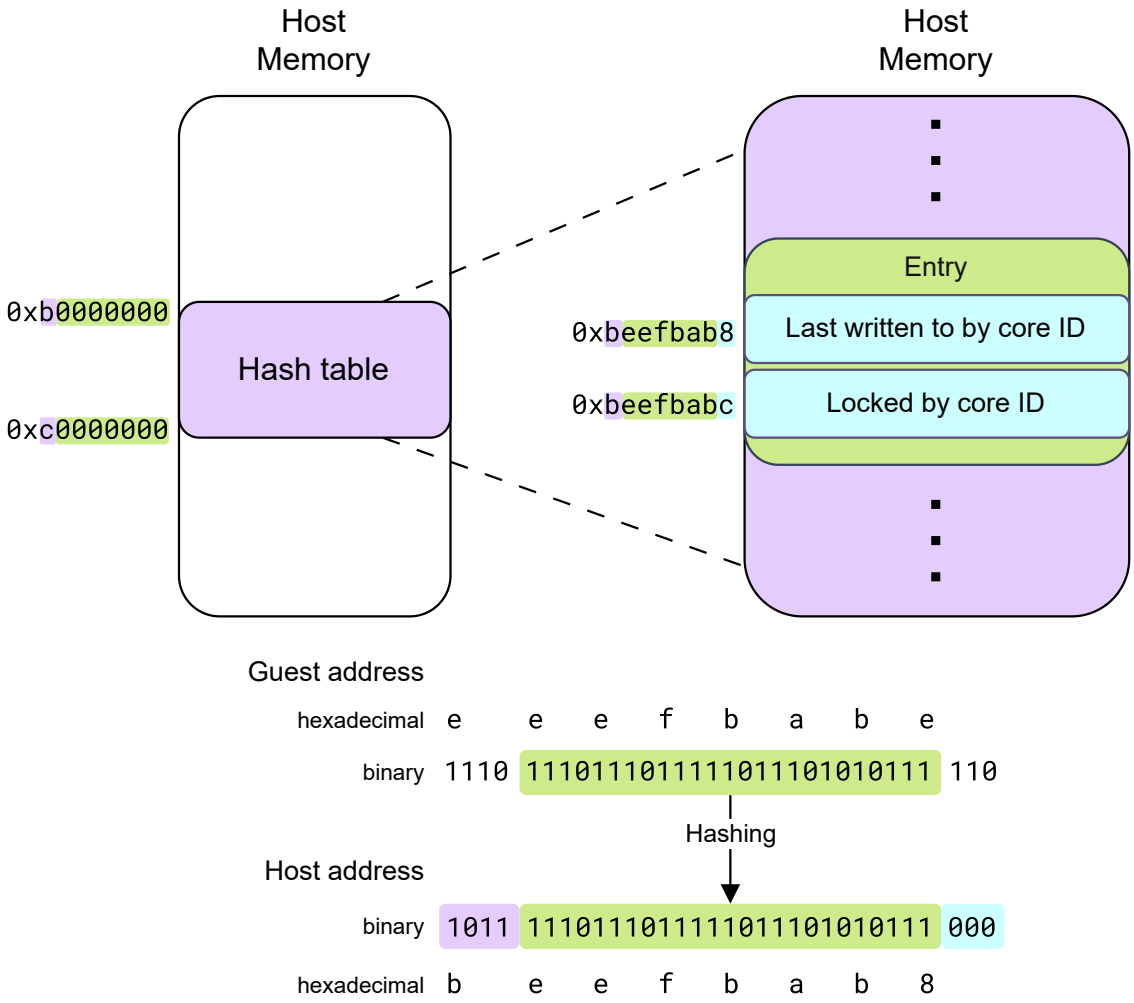


Figure 4.3: The layout of the hash table, illustrating how a guest address is hashed to map to a table entry located in host address space. The address space is 32-bit only for the sake of simpler illustration. In practice, the address space is 64-bit. The color highlighting in the addresses represents which bits address which area of memory.

collision in this context means that two guest addresses get mapped to the same table entry. While ordinary hash tables handle these collisions by either separate chaining (i.e. a linked list) or open addressing (i.e. placing colliding items next to each other), this table does not handle collisions at all. If two guest addresses get mapped to the same table entry, they are allowed to overwrite each other's data. This is because overwriting a table entry means the reservation of the original address is invalidated, requiring the `SC` to fail and the guest code to retry with a new `LR`. This is acceptable, as these spurious invalidations are expected and, crucially, the LR/SC sequence is correctly retried by the guest.

Avoiding hash collision handling yields a more efficient HST implementation. By omitting the additional data structures and allocations necessitated by handling collisions using separate chaining, the table becomes both simpler and smaller. By

avoiding the linear scanning required by open addressing, the hash table operations can be implemented extremely efficiently by means of a few machine instructions. The downside is that it may result in spurious invalidations, the impact of which we present in Section 5.2.3 to answer Research Question *RQ4*.

4.3.2 Hash Table Operations

This section defines the four main operations implemented for the hash table: `set`, `check`, `lock` and `unlock`, as shown in Figure 4.4.

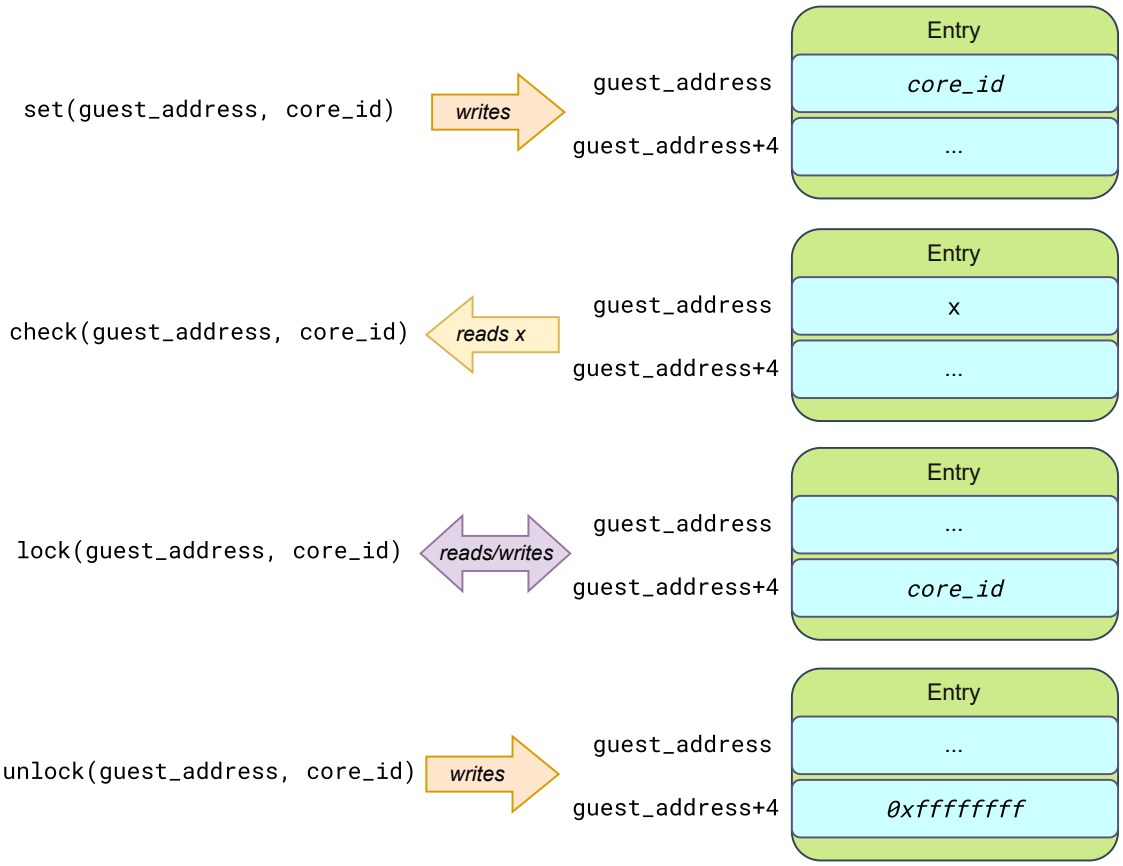


Figure 4.4: The four primary hash table operations used in our implementation of the HST scheme. Modified values are italicized. The ellipsis (...) denotes values that are neither read nor written.

The `set(guest_address, core_id)` operation writes a given core ID to the table entry of the given guest address. It is implemented by hashing the guest address and writing the core ID to the resulting host pointer. This operation is invoked on every guest store in the strongly atomic HST implementations, but only for atomic guest stores for the weakly atomic ones.

The `check(guest_address, core_id)` operation interrogates whether a given guest address was last written to by the given core ID. Its implementation consists of hashing the guest address into a host pointer, reading from the associated table entry and then comparing the entry's core ID with the one given. This is used by

`SC` instructions to check whether a reservation has been invalidated by a store from another core.

The `lock(guest_address, core_id)` operation acquires a lock for the entry associated with the given guest address. Rather than locking the entire table, only one entry is locked to minimize contention. It is implemented by hashing the guest address, incrementing the resulting host pointer to point to the lock field and writing the given core ID using a CAS. If the CAS fails, the operation is retried immediately — making it a spinlock.

The `unlock(guest_address, core_id)` operation releases the fine-grained table entry lock by writing `0xFFFFFFFF` to the lock field of the entry associated with the given guest address. This, theoretically, means a core with ID `4294967295` could view it as acquired, but in practice these IDs start at `0` and linearly increment by `1` for each additional core, making it highly unlikely.

4.3.3 Generalized HST Table Calculations

Zhao et al. describe their table alignment and hash function, but only for a 32-bit address space. We take their method and further generalize the calculations, both to support 64-bit address spaces and to study the impact of table size on performance.

The main parameters, from which the rest can be derived, are: $\text{size}_{\text{table}}$, $\text{size}_{\text{pointer}}$ and $\text{size}_{\text{entry}}$, describing the size in bytes of the hash table, a host pointer and a hash table entry, respectively. The entry size is fixed in our HST implementation at $\text{size}_{\text{entry}} = 8$, leaving table size and host pointer size as the only parameters.

The parameters can be represented in different, equivalent, ways. The parameter $\text{size}_{\text{table}}$ can be represented as $\text{bits}_{\text{table}}$, which is the number of prefix bits used in a pointer to locate the hash table in host address space. The parameter $\text{size}_{\text{pointer}}$ can be represented as $\text{width}_{\text{pointer}}$, which is the number of bits in each host pointer. Equation 4.1 calculates the number of bits inside a host address that are dedicated to addressing table contents, and by using it Equation 4.2 can calculate the number of bytes within the table.

$$\text{bits}_{\text{contents}} = \text{width}_{\text{pointer}} - \text{bits}_{\text{table}} \tag{4.1}$$

$$\text{size}_{\text{table}} = 2^{\text{bits}_{\text{contents}}} \tag{4.2}$$

Equation 4.3 shows how the number of table entries can be derived, which is useful in C code to declare how large the entry array should be.

$$\text{entries} = \frac{\text{size}_{\text{table}}}{\text{size}_{\text{entry}}} \tag{4.3}$$

Furthermore, several relevant bitmasks can be derived from the main parameters.

Equations 4.4–4.6 derive the table interior mask, which can be used to mask out the bits addressing the contents of the table within host memory. We use the notation

$\overline{x}$ to denote x as a bitmask, $\ll$ to denote logical shift left and $\neg$ to denote bitwise negation. Equation 4.4 uses $(1 \ll n) - 1$, which is an idiom used to construct a bitmask containing exactly n `1`s. Equation 4.5 shifts the $\overline{\text{nbits}}$ mask such that the n bits masked out are now those used to address the table itself. Finally, Equation 4.6 negates the table mask to derive a mask for all the bits addressing the table interior.

$$\overline{\text{nbits}} = (1 \ll \text{bits}_{\text{table}}) - 1 \tag{4.4}$$

$$\overline{\text{table}} = \overline{\text{nbits}} \ll \text{bits}_{\text{contents}} \tag{4.5}$$

$$\overline{\text{interior}} = \neg\overline{\text{table}} \tag{4.6}$$

Equations 4.7–4.9 define a mask for extracting the relevant guest address bits. First, Equation 4.7 uses the aforementioned idiom to define a mask for the bits addressing the contents of a single table entry. Equation 4.8 then negates the entry mask to ensure that those bits are zeroed, for correct entry alignment. Finally, Equation 4.9 combines the masks to extract only the relevant middle bits of guest addresses.

$$\overline{\text{entry}} = (1 \ll \text{bits}_{\text{entry}}) - 1 \tag{4.7}$$

$$\overline{\text{alignment}} = \neg\overline{\text{entry}} \tag{4.8}$$

$$\overline{\text{address}} = \overline{\text{interior}} \;\&\; \overline{\text{alignment}} \tag{4.9}$$

For example, we can recreate the numbers used by Zhao et al. from first principles. This is the same scenario illustrated by Figure 4.3. Given that $\text{size}_{\text{table}} = 268435456$ (i.e. 256 mebibytes) and $\text{width}_{\text{pointer}} = 32$ we get (using Equations 4.1–4.2) that $\text{bits}_{\text{table}} = 4$. Furthermore, given that $\text{size}_{\text{entry}} = 8$, we get (using Equations 4.4–4.9) that

$$\overline{\text{address}} = \texttt{0x0ffffff8} = \texttt{0b00001111111111111111111111111000}.$$

This precisely masks out all except the upper four and lower three bits, used for the table location and entry contents, respectively. Thus, for example, given a guest address `0xeeefbabe` and a table located at `0xb0000000`, we get that the relevant guest bits are:

$$\texttt{0xeeefbabe} \;\&\; \overline{\text{address}} = \texttt{0x0eefbab8}.$$

Then, by performing bitwise-or between these bits and the table address, we get the address of the corresponding table entry:

$$\texttt{0x0eefbab8} \mid \texttt{0xb0000000} = \texttt{0xbeefbab8}.$$

4.3.4 HST Table Initialization

In order for the hash table to behave correctly, it needs to be initialized with values that do not represent valid core IDs. This is due to the two fields of the hash table

entries storing core IDs. The ID of the first core is zero, meaning that if the table were initialized with zeroes, then all locks would be owned by that core. This could lead to deadlock.

The solution we implement uses special sentinel values, which indicate something other than their literal value. If a field for core IDs contains the value `0xffffffff` this means *no core at all*, rather than the core with ID 4294967295. Hence, the table is initialized with these sentinel values rather than zeroes.

Theoretically, this could lead to issues when emulating 4294967296 cores, as the last core would get assigned ID `0xffffffff`. Then, other cores would not treat locks acquired by that core as locked, leading to race conditions. However, we deem this scenario to be unlikely as contemporary processors usually have < 1024 cores, and therefore choose to ignore it.

4.3.5 Store Instrumentation

Store instructions need to be instrumented to achieve strong atomicity, i.e., for any shared-memory store to invalidate reservations on the accessed address. Instrumented, in this context, means that additional instructions are emitted by TLIB whenever a guest instruction that writes to shared memory is emitted.

Zhao et al. show that the only instrumentation necessary is the hash table operation `set`, which updates the table with information about which core last performed a store to a given address. For every guest instruction storing a value to an address we therefore emit an additional `set(address, core_id)`. Notably, even in their strongly atomic HST, Zhao et al. do not wrap these steps in a lock. We find that implementing this scheme in Renode results in a race condition, described in the next section (Section 4.3.6). To avoid this race condition in our implementation, we opt for guarding these two steps using a lock. The performance implications of this additional instrumentation are presented in Section 5.2.

Figure 4.5 shows how a simple RISC-V store instruction is implemented by translating it into TCG IR and then into corresponding host instructions. The instructions marked in blue are those solely implementing the store to memory, while the others are additional instructions required for the instrumentation. The red instructions perform reservation invalidation by updating the hash table, with the green ones ensuring mutual exclusion on that address. Atomic read-modify-write (RMW) guest instructions are instrumented similarly.

Figure 4.5 clearly illustrates how instrumenting every guest store instruction significantly increases the number of host instructions executed each time the guest writes to memory. The impact this has on performance is shown later in Section 5.2.

By using fine-grained table entry locks, rather than a global memory lock as is done by Zhao et al. and current Renode, we greatly decrease lock contention. This is important, as the `lock` operation is implemented as a spinlock, i.e., it repeatedly retries acquiring the lock until success. This differs from ordinary mutexes, which put the thread to sleep until the lock is available. A spinlock is used both for simplicity

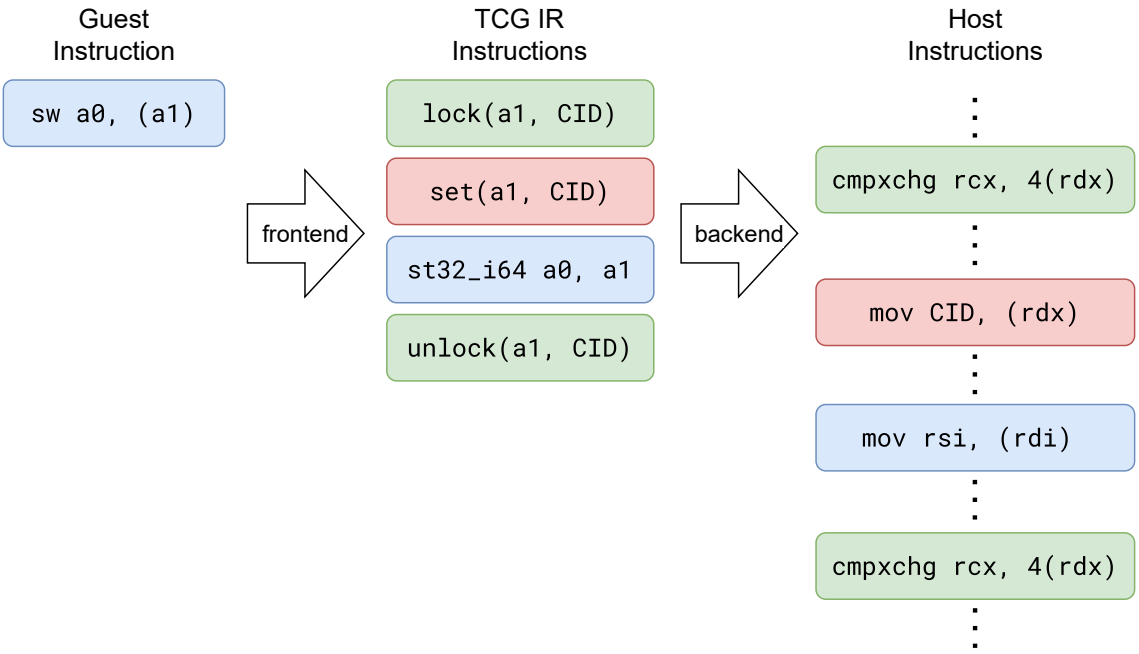


Figure 4.5: How the sw (store word) RISC-V instruction is first translated into several TCG IR instructions, which are then translated into several host instructions. Note that the hash table operations (lock, set, unlock) are pseudo instructions, which are implemented using several TCG IR instructions. The st32_i64 instruction is TCG IR that performs a 32-bit guest memory store on a 64-bit host. CID is the core ID of the currently executing core, a constant value. The ellipsis (...) denotes the omission of several host instructions emitted in the actual translation.

and efficiency, as the implementation only holds these fine-grained locks for very short periods of time (i.e. during one guest instruction).

Not only do store instructions need to be guarded with a lock, but the `LR` and `SC` instructions as well. If they are not guarded, certain interleavings can lead to incorrect behavior as will be shown in the next section (Section 4.3.6).

Figure 4.6 illustrates how we implement the `LR` instruction. The `reserved_address` is a per-core variable used for keeping track of which address is currently reserved. Note how the hash table operation `set` and the load instruction are guarded by the fine-grained locking (in green).

Figure 4.7 shows that the `SC` instruction is also implemented using a table entry lock. The `check`, `set` and store are all performed within the confines of the lock to ensure that a checked reservation is not invalidated before the `SC` completes its store successfully. The if-statement at the very top is a simplification of how the per-core `reserved_address` value is checked before even acquiring the lock, to ensure the `SC` has a corresponding `LR` to pair with.

Important to note is that the above descriptions of using a fine-grained lock around stores, atomic RMWs and `LR`s apply only to the strongly atomic (i.e. fully correct)

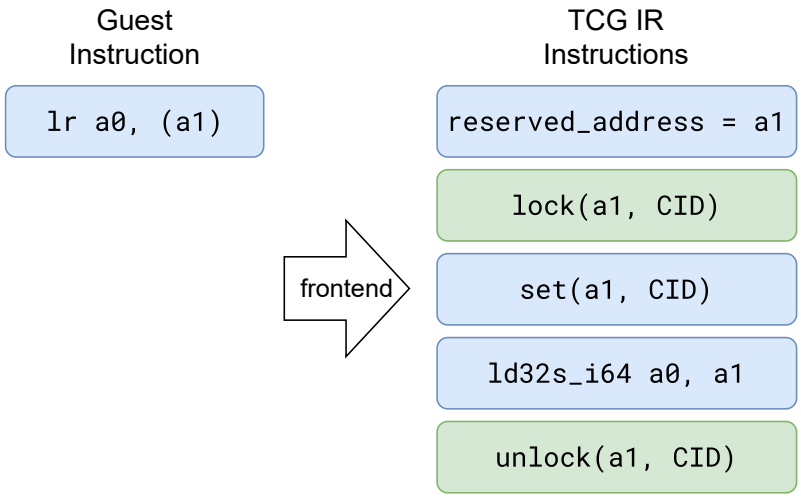


Figure 4.6: How the RISC-V guest instruction LR is translated into TCG IR instructions. CID is the core ID of the currently executing core, a constant value. The ld32s_i64 instruction is TCG IR that performs a sign-extended 32-bit guest memory load on a 64-bit host.

variants of our implementation. The weakly atomic ones forego locking around most instructions, except for around `SC`.

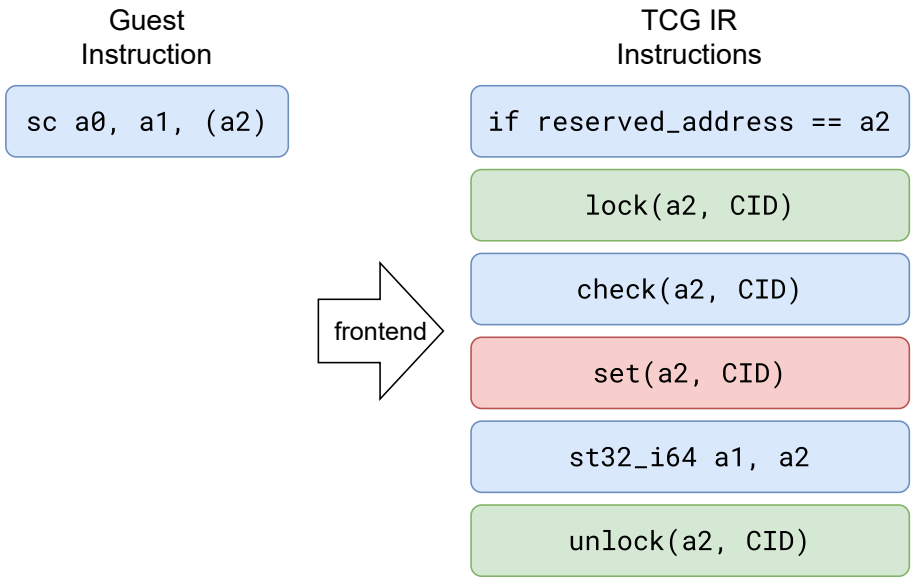


Figure 4.7: How the RISC-V guest instruction SC is translated into TCG IR instructions. CID is the core ID of the currently executing core, a constant value. The st32_i64 instruction is TCG IR that performs a 32-bit guest memory store on a 64-bit host.

4.3.6 Incorrectness of HST-Strong

While our implementations are based on Zhao et al.'s HST-strong scheme, they differ in a minor but crucial aspect: their use of locks. Our implementation locks `LR`, `SC` and `store` instructions to ensure mutual exclusion, while Zhao et al. only lock their `SC`. These additional locks worsen performance, but we have found them to be necessary to prevent a race condition present in Zhao et al.'s scheme.

The race condition was found while model checking the **LR/SC Invalidation by Store** model described in Section 3.1.2.1. The exact trace of the counterexample, showcasing the race condition, is available in Appendix C.

The cause of the race condition is a lack of mutual exclusion in critical sections within the `LR` instruction and the instrumented store instructions. Figure 4.8 shows how the `LR` instruction may interleave inside the critical section of the instrumented store, producing an incorrect result. It is incorrect because the `LR` instruction reads one value, but then the `sw` (store word) instruction overwrites that value without invalidating the reservation.

Figure 4.8 exemplifies the reason why `sw` can overwrite the shared variable without causing an invalidation. At point (a), core 1 performs its invalidation, setting the *Last written to by core ID* field to 1. Then core 0 interleaves at point (b), overwriting the field to 0. This would normally be harmless, were it not for point (c) at which core 1 finishes executing its `sw` instruction and mutates the shared variable. At this point, core 0's execution of `LR` believes it has successfully reserved one value, while in fact core 1 has changed it. This will cause the subsequent `SC` by core 0 to succeed, even though it should fail due to the value having changed.

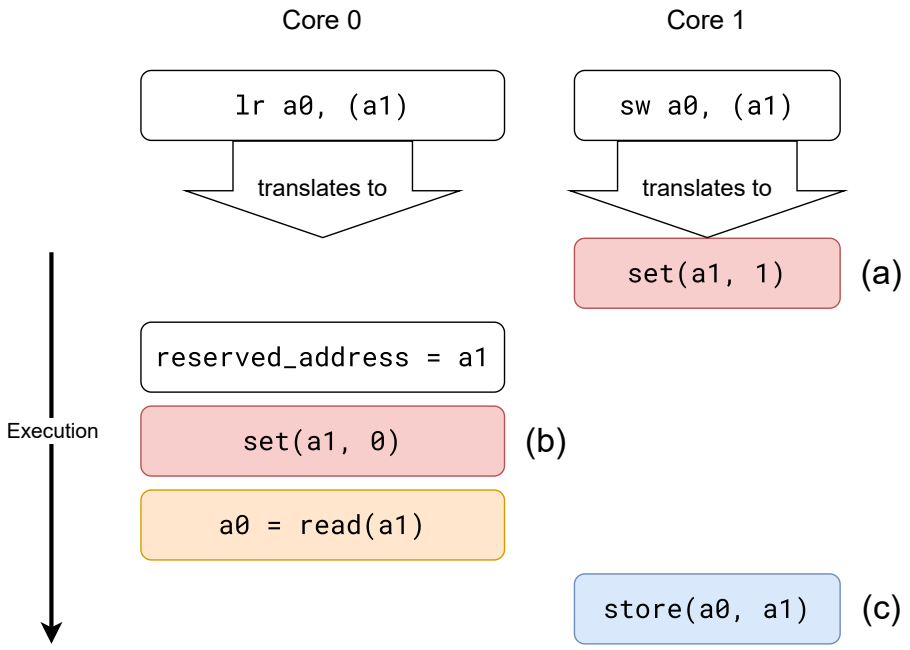


Figure 4.8: Two cores concurrently executing lock-free LR and lock-free instrumented store instructions, leading to a race condition. The a1 register contains the address of the shared-memory variable, and the syntax (a1) is used to refer to the value located at that address.

4.3.6.1 The CAS Mitigation

As previously mentioned, Zhao et al. do not mention a CAS when describing HST in their paper, even though they use one in their implementation. This is a remnant of Qemu's old LR/SC implementation.

The CAS used by Zhao et al. partially mitigates the above race condition, by keeping track of which value the `LR` instruction observes. This value is then used by a CAS in the `SC` implementation, to make sure that the value at the shared-memory location is the same as the one reserved by `LR`. While this does not eliminate the data race between `LR` and instrumented stores, it allows `SC` to detect and prevent the race condition. This explains why Zhao et al. do not encounter the race condition.

In fact, we verified using model checking that the CAS mitigation resolves the above race condition. No counterexamples were found by the exhaustive verification of our **LR/SC + CAS** model, described in Section 3.1.2.5. Note that this model is limited to two concurrent cores, with one performing LR/SC and another performing an instrumented store.

However, the CAS mitigation is insufficient for full correctness, as it does not address the inherent issue with CAS operations: the ABA problem. Figure 4.9 shows a scenario with three concurrent cores, producing a race condition. The only difference between this figure and Figure 4.8 is the addition of a third core, interleaving with yet another instrumented store instruction.

In Figure 4.9, core 0 executes part of the CAS mitigation at point (a), keeping track of the value reserved by the `LR` instruction. Then, at point (b) core 2 interleaves and overwrites the shared-memory value. At this point, the CAS mitigation would be sufficient to detect the interleaving store, as the value of the shared-memory variable has changed. Unfortunately, core 1 interleaves at point (c) and performs yet another store — changing the shared-memory variable back to its original value. From this point on, the CAS performed by the `SC` will not be able to detect the interleaving stores, as it only compares the current value with the originally reserved value. Here, the `SC` should fail, due to its reservation having been invalidated by the first store — but it succeeds, which is incorrect (see Section 4.2.1.1).

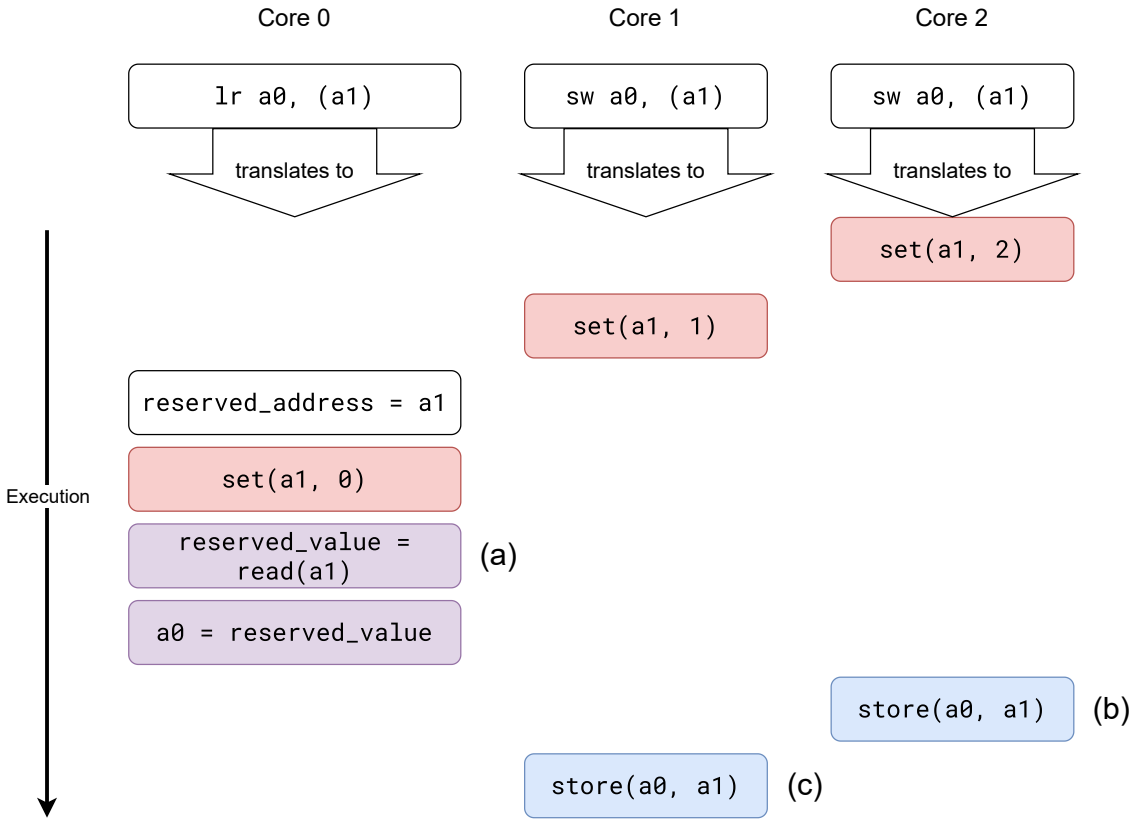


Figure 4.9: Three cores concurrently executing lock-free LR and lock-free instrumented store instructions, leading to a race condition even with the CAS mitigation of Zhao et al. The a1 register contains the address of the shared-memory variable, and the syntax (a1) is used to refer to the value located at that address.

This incorrectness is not present in our HST-based implementations within Renode. Our implementations do not need any CAS-mitigation thanks to the use of fine-grained locking around `LR`, `SC` and `store` instructions.

4.3.7 Measuring Spurious Reservation Invalidations

Even though spurious invalidations are handled by the guest, and therefore do not affect program correctness, they may impact performance. The more spurious

invalidations a platform exhibits, the more each LR/SC transaction needs to be retried, which takes more time.

Spurious invalidations may occur in our implementation whenever two or more guest addresses get hashed to the same hash table entry, and one core overwrites the active reservation of another. To indirectly measure the number of spurious invalidations, we therefore collect statistics about how many guest addresses get mapped to the same entry, i.e., how many hash collisions occur.

The number of hash collisions is not the same as the number of spurious invalidations, it is merely an indicator that the risk for spurious invalidations exists. As the number of hash collisions grows, the number of different addresses using the same table entry increases, leading to a larger risk of spurious SC failure.

Furthermore, to estimate the number of spurious invalidations, we collect statistics about how many times `SC` instructions operating on an address mapped to a particular table entry fails. We group these by the hash, i.e., the table entry, to correlate them with the number of hash collisions at that entry.

By evaluating both the number of collisions and the number of `SC` failures, we may draw conclusions about how hash table size affects the number of spurious invalidations. This is done in Section 5.2.3.1.

5

Results

This chapter presents the performance of our LR/SC implementations in Renode, and the correctness of the underlying algorithm.

5.1 Correctness

The Promela models described in Section 3.1.2 were used to exhaustively verify whether the underlying algorithm is correct. This section describes the output of the models whose results have not yet been presented, along with their relevance to our implementation.

5.1.1 Livelocking of LR/SC

The **LR/SC Livelock** model (see Section 3.1.2.3) shows that the HST scheme does not avoid livelock. It is entirely possible for two cores concurrently executing LR/SC loops to repeatedly invalidate each other in a never-ending loop. Possible mitigations and implications of this are discussed in Section 6.1.4.

5.1.2 Isolated LR/SC Instructions

Excluding livelock for verification purposes, the **LR/SC Racing** model (see Section 3.1.2.4) verifies that the HST scheme correctly handles the scenario where two cores concurrently execute LR/SC on the same shared-memory address. Notably, it is unable to find any incorrect interleavings even if the mutual exclusion around the `SC` is disabled. This may indicate that implementing the HST scheme without any mutual exclusion is enough for correctness between concurrent LR/SC instructions (i.e. weak atomicity).

However, weak atomicity is insufficient in the presence of stores to the same shared-memory address — which is why our implementations guard the `LR` and `SC` instructions with a fine-grained hash table-based lock.

5.1.3 Fine-Grained Hash Table-Based Locking

This model (see Section 3.1.2.2) verifies that replacing all the locking and CAS operations in the HST scheme with fine-grained hash table-based locks is sound in

the context of a specific problematic scenario. Neither the correctness issue from Section 4.3.6, nor any other issues were encountered by this model. Note that this does not exclude the possibility of incorrectness, as the model only checks a specific problematic scenario.

5.2 Performance Results

These results compare six versions of Renode:

- HST-weak using IR
- HST-strong using IR
- HST-weak using helpers
- HST-strong using helpers
- the current Renode implementation
- Renode without bypassing the softMMU

The benchmarks were carried out on an Intel Core i9-9980HK clocked at 2.4 GHz with 8 physical cores and 50GB of RAM. Both boost clock and hyperthreading were disabled to ensure a more reliable benchmarking environment.

5.2.1 Microbenchmarks

The two microbenchmarks evaluated the LR/SC implementations in two specific scenarios: one with a highly contended synchronization variable and another with no contention at all.

5.2.1.1 Uncontended LR/SC

Figure 5.1 shows the uncontended scenario defined in Section 3.2.1.2. Note the difference between subfigures (a) and (b). Subfigure (a) displays absolute performance, in seconds. There, it is clear that the two versions based on Renode's current atomics implementation (in red and pink) — which use the global memory lock — perform worse the more threads are added. In (a), all the HST implementations appear to perform the same, when in fact they do not — as exemplified by (b). There, the performance differences are more apparent.

In Figure 5.1 (b) it can be seen that the best-performing implementation is HST-weak written in IR (dark green), with HST-weak written using helpers (light green) coming in a distant second. This may be due to how the HST-weak implementations do not perform any locking (fine-grained or otherwise) around the `LR` instruction, with only the `SC` instruction being guarded. The HST-strong implementations (in blue), on the other hand, guard both `LR` and `SC` with a fine-grained lock and thus perform worse.

Furthermore, Figure 5.1 (b) illustrates the performance difference between helpers- and IR-based implementations. Note the distance between IR weak (dark green)

and Helpers weak (light green), indicating that IR weak always performs better than Helpers weak. The same pattern is seen in the HST-strong implementations (in blue). This uncontended benchmark shows that IR is, on average, 33.75% faster than Helpers for HST-strong and 39.5% faster than helpers for HST-weak.

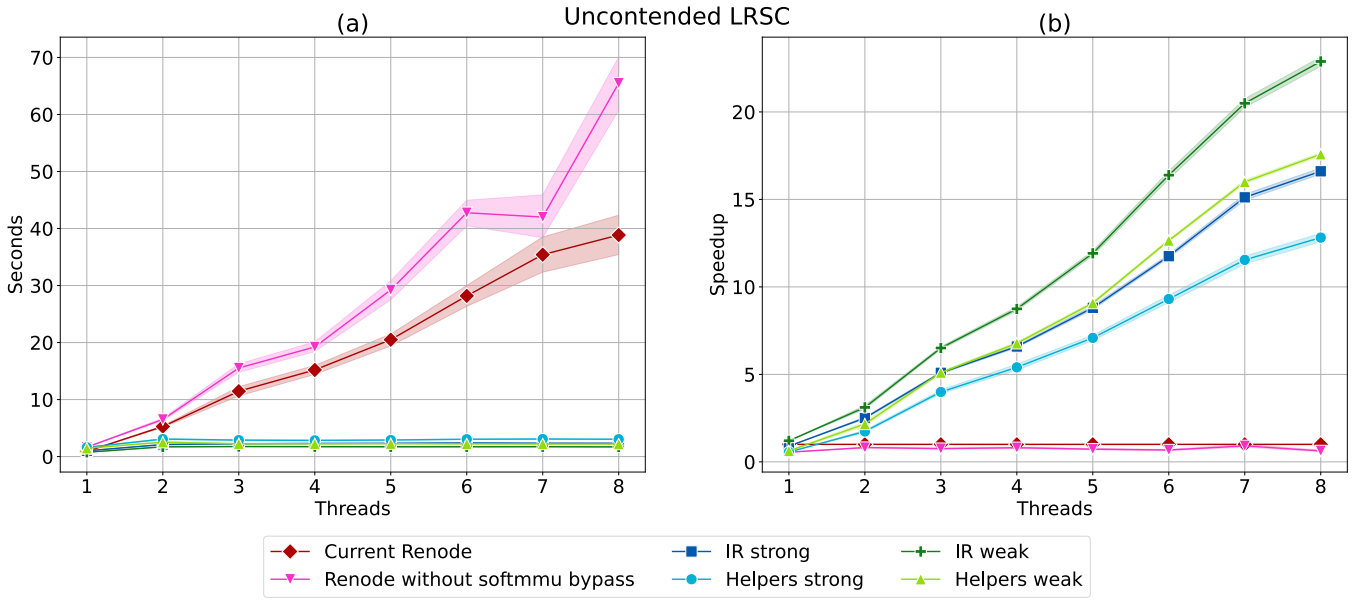


Figure 5.1: How the different implementations scale in a microbenchmark scenario with uncontended LR/SC, as the number of threads increase.

The reason that the HST implementations scale much better than current Renode in this benchmark is that they avoid the global memory lock. One of the main benefits of the HST scheme is that it allows for fine-grained locking of reservations, allowing concurrent threads accessing disjoint reservation sets to proceed without synchronization. The implementations that make use of a global memory lock, however, need to constantly synchronize due to the lock being shared between threads.

5.2.1.2 Contended LR/SC

Figure 5.2 shows the contended scenario defined in Section 3.2.1.1. Here, the confidence intervals are larger (indicated by the translucent error bands), making it more difficult to draw conclusions.

Both subfigures (a) and (b) of Figure 5.2 clearly indicate that the HST implementations perform worse than current Renode (red) in the contended scenario. Using 6 threads, the HST implementations (green, blue) converge in that they all perform their worst: IR weak is 36.2x slower than current Renode and IR strong is 21.4x slower. There is no performance data from the HST-strong implementations for 7 and 8 threads due to their runtimes making it infeasible to complete in time. The HST-weak implementations all completed in time, indicating that they perform better than HST-strong in this microbenchmark. The reason for the recovery of HST-weak implementations beyond six threads remains unclear.

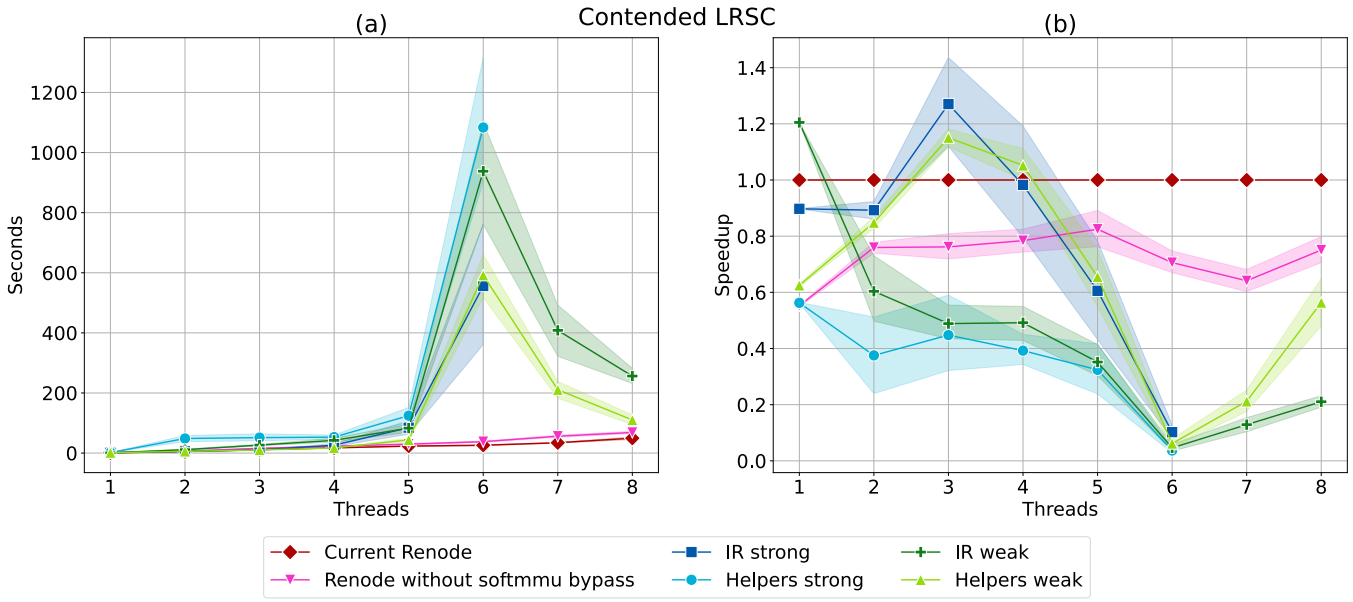


Figure 5.2: How the different atomics implementations scale in a microbenchmark scenario with contended LR/SC, as the number of threads increase.

The HST scheme performs much worse than the previous, global memory lock-based, solution due to how the mutual exclusion is implemented. The global memory lock uses an OS-provided mutex, which puts threads to sleep while waiting for the lock. The fine-grained locking in the HST scheme, however, busy-waits in a loop until the lock is released.

Putting a thread to sleep, and subsequently waking it up, is a slow process due to the associated syscall overhead [28]. This leaves room for other threads to acquire the mutex. The tighter loops of the HST-based implementations do not leave much room, due to how fast they retry acquiring the lock. There appears to be a correlation between how quick an implementation is and how slow it performs in this highly contended scenario, with quicker implementations performing much worse than slower implementations.

Figure 5.3 is another view of the same data shown in Figure 5.2, highlighting the large variance using box plots. Note the wide whiskers of the boxes, indicating how the range of observed values far exceeds the upper and lower quartiles indicated by the box itself. Furthermore, numerous outliers — automatically identified and excluded using a method that is a function of the inter-quartile range — are present and denoted by circles. Therefore, the data used to plot Figure 5.2 may not be entirely representative of real-world performance.

The large variation observed in the contended scenario is most likely caused by nondeterminism in how the host OS schedules the threads. Depending on the order threads execute in, they may have no issues acquiring a lock since it may not be under contention during that time slice — resulting in better performance. The opposite may occur instead, where threads that get to execute their time slice are not the ones holding the lock and thus can only busy-wait — resulting in worse

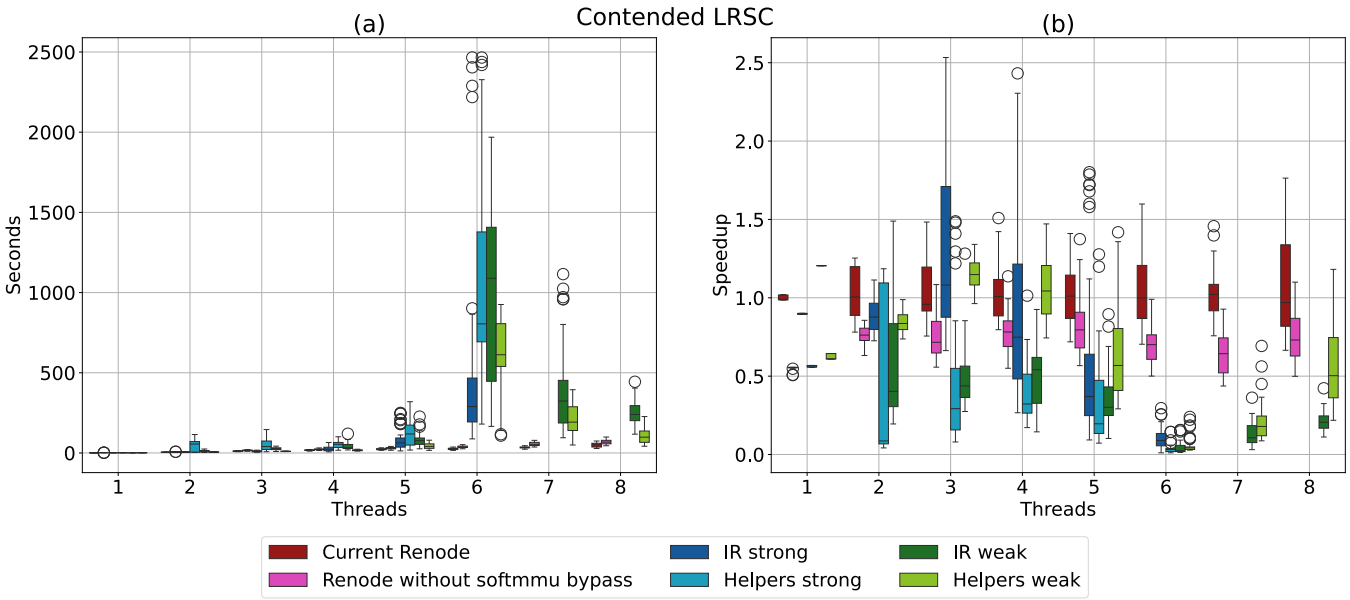


Figure 5.3: The variation in performance in a microbenchmark scenario with contended LR/SC, as the number of threads increases. The whiskers of the box plot extend to a value within the 1.5 IQR.

performance. Possible mitigations to this are discussed later in Section 6.1.4.

5.2.2 PARSEC Benchmarks

Figure 5.4 shows the PARSEC benchmarks' performance results. The performance of current Renode (in red) is measured with global serial execution enabled (see Section 2.2.3). The reason is because current Renode does not successfully emulate the BeagleV-Fire platform, on which the benchmarking was carried out, without global serial execution enabled. The emulation deadlocks without global serial execution, preventing the benchmark from proceeding. This may be due to synchronization issues caused by current Renode's incorrect LR/SC implementation (see Section 4.1.2).

Table 5.1 shows how many of each benchmark's executed guest instructions are LR/SC or stores. Notably, *Fluidanimate* has the largest proportion of LR/SC, while *Freqmine* has the lowest. Across all PARSEC benchmarks, $< 0.1\%$ of all executed instructions are LR/SC. In contrast, store instructions dominate the execution of several benchmarks. In three out of seven benchmarks, the number of stores surpasses 50% — making its performance the dominating factor. These percentages differ greatly from Zhao et al. [2], who measured the same benchmarks. One reason for this difference may be that these numbers were collected from PARSEC compiled for RISC-V rather than for ARM, which is what they measured.

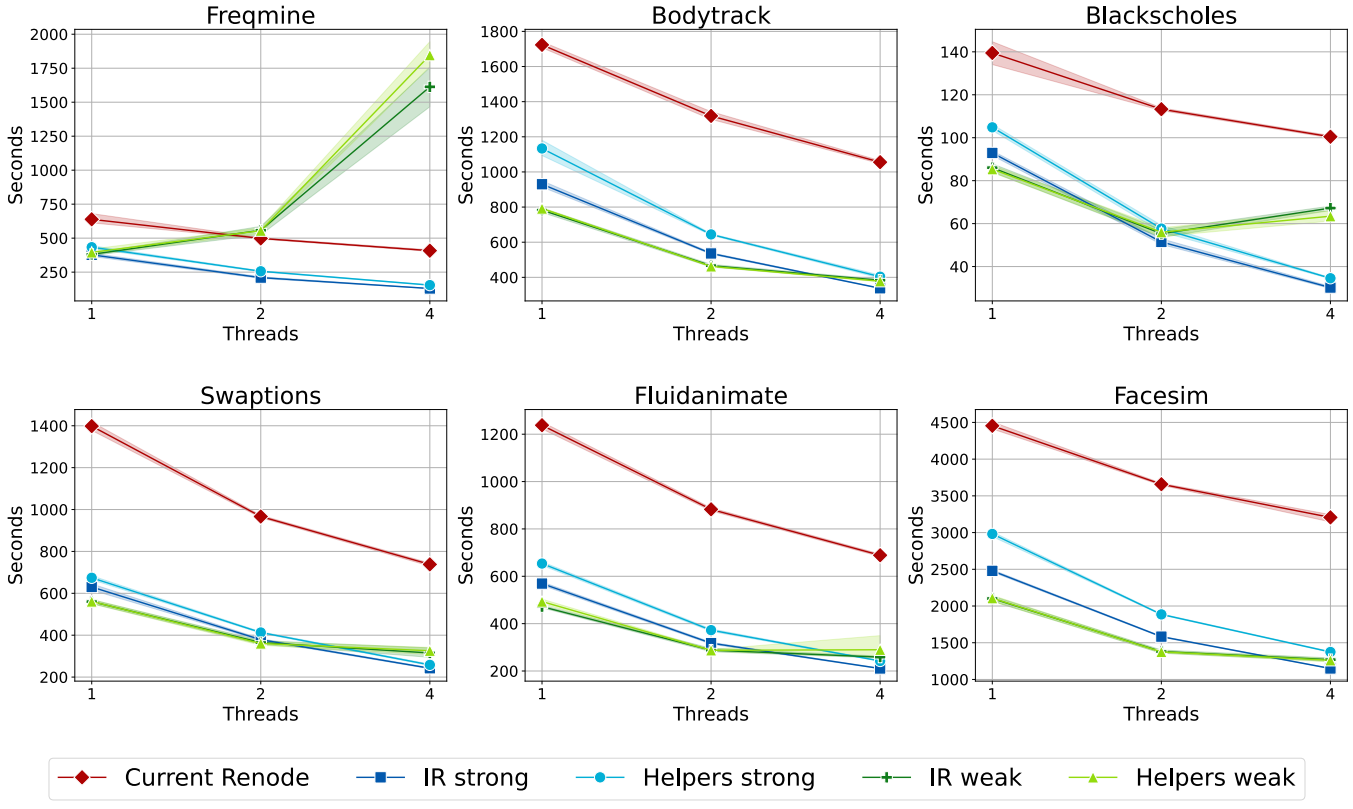


Figure 5.4: The absolute performance of the different PARSEC benchmarks, emulated using different Renode implementations. Each data point represents the mean of ten iterations and the translucent error band displays the 95% confidence interval. The hash table size is 256 MiB. Note that the Y-axis has been truncated.

Benchmark	LR/SC (%)	AMO (%)	Stores (%)	Instrumented (%)
Fluidanimate	0.06832	0.03361	24.92	25.02
Swaptions	0.05302	0.02659	57.88	57.96
Linuxboot	0.02172	0.28742	4.21	4.52
Facesim	0.01944	0.00851	51.77	51.80
Blackscholes	0.00246	0.00139	1.27	1.27
Bodytrack	0.00168	0.00282	52.02	52.02
Freqmine	0.00005	0.00018	13.56	13.56

Table 5.1: The dynamic counts of LR/SC, other Atomic Memory Operations (AMO) and store instructions in the benchmarks. The PARSEC benchmarks were measured with 4 threads and using the largest available input (simlarge). The *Instrumented* column depicts how many of the total executed instructions were LR/SC, stores or AMO (i.e. the sum of the preceding columns).

A consistent pattern in Figure 5.4 is that the HST-strong implementations (in blue) scale well, speeding up as the number of threads increase. The IR strong implementation performs about 6%-18% better than Helpers strong across the whole set of benchmarks.

It is clear in Figure 5.4 that both HST-weak implementations (in green) perform similarly. This is due to how HST-weak implementations do not instrument store instructions, which means that any overhead incurred by helpers over IR is not significant in the total runtime of the program, which is dominated by stores. For a clearer view of the runtime variation, box plots are available in Appendix D.

The benchmarks in Figure 5.4 follow the same performance curve, with an increase in performance between one and two threads. For some benchmarks, there is an arguably insignificant increase in performance between two and four threads. In others, there is even a decrease. *Freqmine* deviates significantly from the other benchmarks, showing a notable drop in performance between two and four threads. It is unclear why *Freqmine* behaves differently from the others. Across all benchmarks, the HST-weak implementations perform better than HST-strong on one and two threads, but scale worse at 4 threads. The reason for this behavior at 4 threads remains unclear.

5.2.3 Table Size and Spurious Invalidations

The hash table size is configurable and impacts how many spurious reservation invalidations occur, making it important to choose a suitable size. This section presents both collision statistics and performance results at different hash table sizes.

5.2.3.1 Collision and SC Failure Statistics

Figure 5.5 (a) depicts the number of hash collisions as the hash table size increases. It is clear that as the hash table grows smaller, the number of collisions greatly increase. At the smallest table size — 8 bytes, i.e., one hash table entry — more

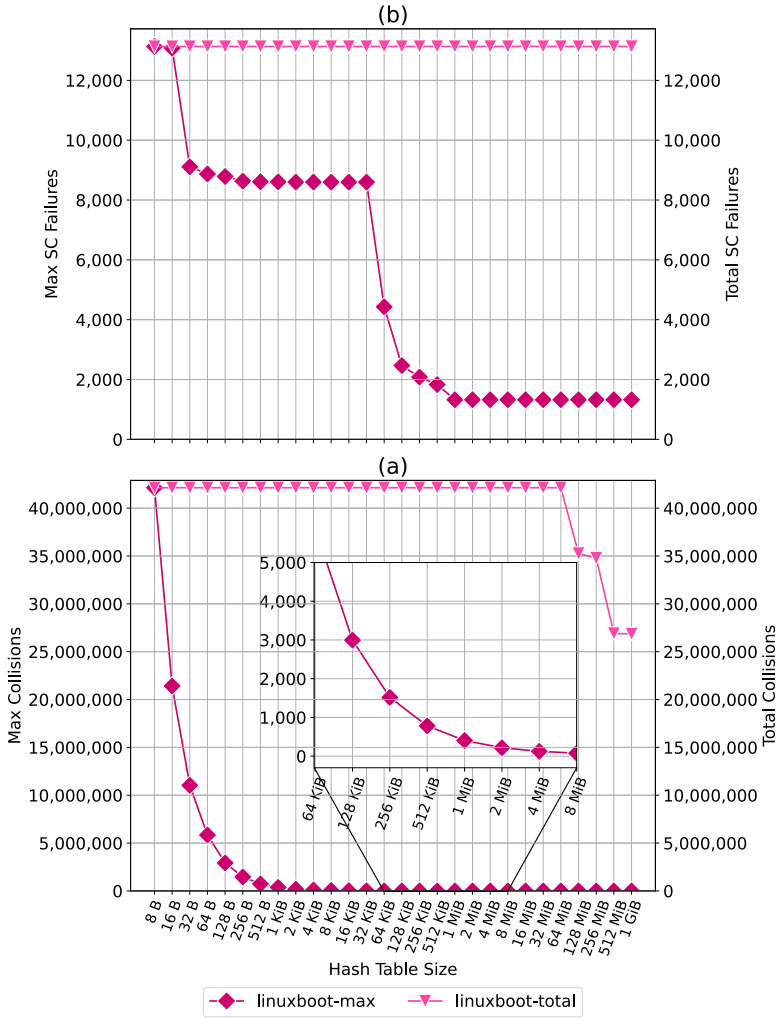


Figure 5.5: How the number of hash table collisions and SC failures scale for the BeagleV-Fire Linux boot, as the table size increases. The values shown for *linuxboot-max* are the *maximum* values encountered at a given size, i.e., the table entry with the most collisions or most SC failures. The values shown for *linuxboot-total* are the *total* number of values, i.e., the sum of all table entries' collisions or SC failures.

than 40 million collisions are encountered. At the other extreme, with a 1 GiB table, only 18 collisions are experienced.

Figure 5.5 (b) depicts how many times SC instructions fail as the hash table size increases. The correlation with subfigure (a) is not as close as one would first assume. Largely, the same pattern emerges — with smaller table size causing more SC failures. These are most likely due to spurious reservation invalidations, caused by hash table collisions. Notably, there are plateaus where the number of SC failures stay consistent, even as the table size changes.

The two most prominent plateaus of Figure 5.5 (b) are between table sizes 32 B–32 KiB and 1 MiB–1 GiB. One possible explanation for this distinct change in the

number of SC failures is that there may be contended synchronization variables that move from a colliding table entry to two distinct entries.

The inset plot of Figure 5.5 (a) highlights the region of 64 KiB–8 MiB table sizes, where a point of diminishing returns is reached. There, as the table grows, the number of collisions plateaus. Combined with the apparent plateau of subfigure (b) in the same region, this indicates that a table size in the range of 512 KiB–8 MiB results in a good balance between size and collisions.

The number of hash collisions experienced by the PARSEC benchmarks at different table sizes is shown in Figure 5.6 (a). The pattern here is similar to Figure 5.5 (a), where a larger hash table results in fewer collisions. However, the point of diminishing returns is reached earlier — at around 4 KiB–16 KiB, where the maximum number of collisions is on average 28. Note that the scale of Figure 5.6 (a) is vastly different than Figure 5.5 (a): it is in the tens of thousands rather than the tens of millions. This indicates that the Linux boot writes to a greater variety of memory locations than the PARSEC benchmarks, causing hash collisions at several different memory locations.

Figure 5.6 (b) shows how the total number of hash collisions varies with table size. It is clear that there is a large variance, but there is still a common pattern. Unlike the Linux boot in Figure 5.5 (a), the total number of collisions does not remain consistent. It drops vastly after the table grows to 64 KiB, indicating that the PARSEC benchmarks experience fewer collisions (and thus less risk of spurious invalidation) as table size grows.

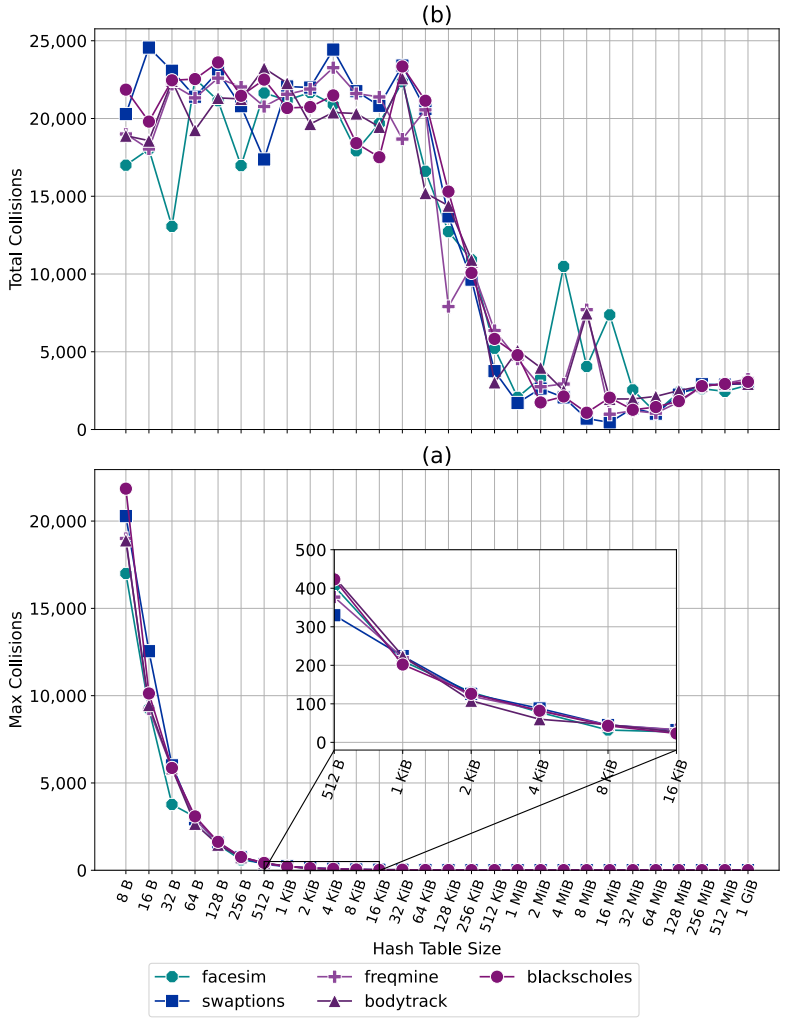


Figure 5.6: How the number of hash table collisions scales for the PARSEC benchmarks, as the table size increases. The values shown in (b) are the total number of collisions, i.e., the sum of all table entries' collisions. The values shown in (a) are the *maximum* number of collisions encountered at a given size, i.e., the table entry with the most collisions.

5.2.3.2 The Impact of Table Size on PARSEC Performance

Figure 5.7 shows the performance of the PARSEC benchmarks at different table sizes. At table sizes above 128 B, the HST-strong implementations (in blue) outperform the global memory lock-based current Renode. At 128 bytes, the hash table consists of 16 entries — allowing 16 concurrent fine-grained locks to be held by different cores. At 8 bytes, consisting of a single entry, the HST-strong-based implementations (in blue) perform considerably worse than current Renode, indicating that one hash table entry is not entirely equivalent to a global memory lock.

Furthermore, Figure 5.7 displays that the performance of HST-weak-based implementations (in green) is not affected by table size. This is due to how HST-weak performs no store instrumentation, which is significant as Table 5.1 shows that the

dominating factor in these benchmarks is the store instructions. Since HST-strong (in blue) instruments every single store instruction, this adversely affects its performance when the table size decreases and contention increases on the locks guarding the instrumented store instructions.

It is also clear in Figure 5.7 that the HST-strong-based implementations (in blue) reach a plateau at around 8 KiB, which appears to be the point of diminishing returns in terms of store table size. No significant performance gains are achieved by increasing the table size beyond that.

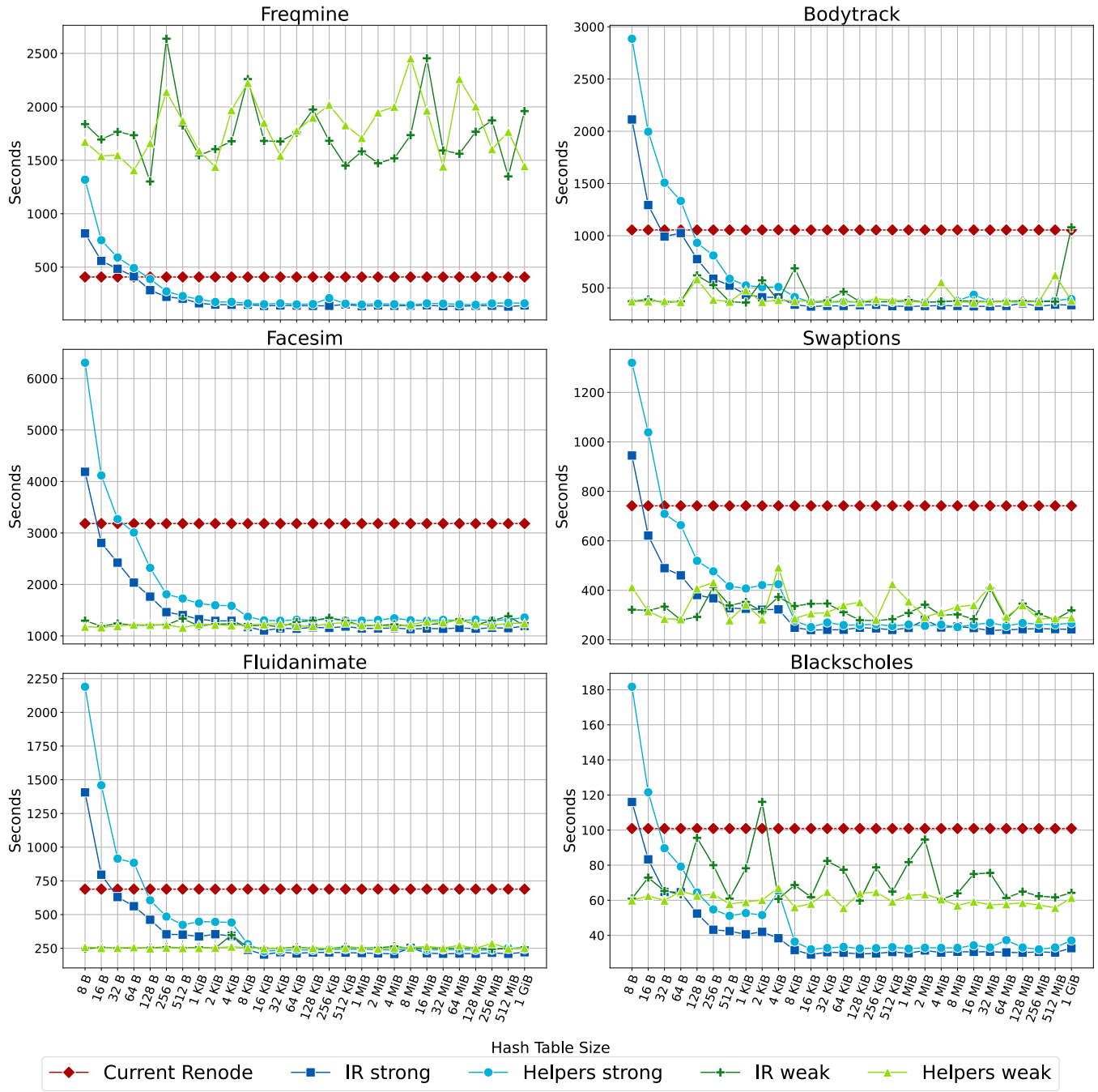


Figure 5.7: How the performance of the PARSEC benchmarks scales with different hash table sizes. Current Renode (in red) does not use a hash table, but is included for reference. The PARSEC benchmarks were measured on 4 threads and the simlarge input.

6

Conclusion

This chapter begins by discussing interesting aspects of the results and implementation, followed by a review of previous work in this field. The chapter then concludes the thesis by summarizing the main results.

6.1 Discussion

This section contains various discussions about interesting, or otherwise noteworthy, topics relating to the results shown in Chapter 5.

6.1.1 Spurious Invalidations and Reservation Sets

In Section 5.2.3 we show that our implementations exhibit spurious reservation invalidations, to varying degrees depending on table size. These invalidations are not harmful to correctness; rather, they harm performance by necessitating additional retries in guest code.

These spurious invalidations, however, are also present in hardware implementations and are specifically allowed by the RISC-V specification [14, Section 14.3]. The precise definition of `LR` does not state that it reserves an address, rather it "registers a *reservation set*—a set of bytes that subsumes the bytes in the addressed word." [14, Section 14.2]. Furthermore, the specification plainly states:

> "An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword." [14, Section 14.2]

In our implementation, one could argue that we create a reservation set through the hashing mechanism. The hash function maps several addresses onto a hash table entry, effectively forming a reservation set consisting of those addresses.

6.1.2 Model Checking Limitations

The model checking described in Section 3.1.2 and Section 5.1 is performed exhaustively, using Spin, which may create the false impression of complete and rigorous correctness. However, this is not the case, as the choice of models significantly limits

the verification process. While the verification of a single model is exhaustive, this does not prove anything outside of the case being modeled.

For example, several of the models used in this thesis set up a scenario with two cores. The purpose of these models is to detect specific race conditions between the two cores, which they do. What they do not do, however, is assert anything about the existence of race conditions in scenarios with more than two cores. This is exemplified in Section 4.3.6.1, where our two-core model is not able to detect the ABA problem that is present in a scenario with three cores.

To more robustly assert correctness, one would need a formal correctness proof of the algorithm. This thesis does not provide such a proof.

6.1.3 ISA-Independence of the Implementation

We describe and implement our version of the HST scheme in the context of RISC-V guests and x86 hosts, but the problem at hand — emulating the LR/SC family of instructions on hosts without native support — is more general. Zhao et al. [2], for example, do their work on Arm guests.

The implementation presented in Section 4.3 makes assumptions both about the guest's semantics and the host's capabilities.

One host-assumption is the atomicity of naturally aligned stores. Our implementation critically relies on ordinary store instructions being globally visible by all cores, to implement the fine-grained locking mechanism correctly. On a host platform where this is not the case, care would need to be taken to swap these instructions for atomic ones. This would most likely not be a complicated endeavor, but it is a non-portable aspect of the implementation nonetheless.

Several guest-assumptions lie in the implemented semantics of LR/SC. As the RISC-V specification was used during development, the implemented semantics are tailored to match those of RISC-V guests. This may make the current implementation unsuitable for Armv8-A guests' LL/SC instructions.

Although few modifications are theoretically necessary to support other guests and hosts, care must be taken regarding host atomics and guest semantics.

6.1.4 Livelock and a Lack of Forward Progress Guarantee

We show in Section 5.1.1 that there is a risk of livelock in our implementation, where no forward progress is made by cores repeatedly invalidating each other's reservations. This may be an issue, both in regards to real-world performance and in regards to implementing the RISC-V specification.

The RISC-V specification defines *constrained LR/SC loops*, unto which they bestow an architectural forward progress guarantee. Among other properties, a constrained LR/SC loop must fulfill these:

- "The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory.
- The LR and SC addresses must lie within a memory region with the LR/SC eventuality property. The execution environment is responsible for communicating which regions have this property." [14, Section 14.3]

We do not implement this guarantee.

One could argue that it would be enough to implement a way of communicating to guest code that no regions of Renode's emulated guest memory possess the eventuality property. Then one of the properties necessary for the forward progress guarantee to be valid are eliminated, thus technically fulfilling the specification.

We instead choose to argue that in practice we do effectively guarantee eventual forward progress. This is due to several factors. One is how Renode slices execution of guest code into translation blocks, the other is how the host OS' scheduling of threads affects livelock.

Renode does not continue executing guest code ad infinitum, until it is stopped by the user. Instead, it rather frequently takes over control from tlib and executes non-guest code. This is necessary for handling of guest MMU faults, guest interrupts and correct flow of virtual time. These transitions of control flow away from guest code may break any livelock that is occurring, simply due to the execution variance it introduces. The variance may be enough to allow one guest core to succeed with its `SC` without being invalidated by any other core.

The guest cores emulated by Renode execute on separate host OS-provided threads, which also introduces variance by means of non-deterministic scheduling. If two guest cores are livelocking, all it takes for the livelock to be broken is for the OS to happen to schedule one tlib thread alone for a duration of time long enough that it can succeed with its `SC`. In real-world scenarios, where the OS is managing several other concurrent processes, this seems likely to happen eventually.

While these, by no means, constitute a formal forward progress guarantee — they are enough in practice. Thus, it may not be necessary to implement a forward progress guarantee.

One benefit of implementing some form of forward progress guarantee is that pathological scenarios, like the contended LR/SC microbenchmark from Section 5.2.1.2, would most likely perform better. This is because a forward progress mechanism would put an upper limit on how long two cores may stay livelocked, decreasing the variance of that benchmark.

6.2 Previous Work

As processor core counts continue to increase, efficient multi-threading support in DBTs becomes increasingly relevant. Much of prior DBT synchronization research

has been implemented in the DBT Qemu. Since Qemu and Renode originally shared the same translation logic, Qemu research is applicable in Renode and vice versa.

In the early days of binary translators, multi-core support was less of a concern, as single-core processors were the norm. In 2005, the creator of Qemu described its threading support as immature due to locking issues [29].

In 2011, Wang et al. proposed an emulation framework called COREMU [8], aimed at improving Qemu's multi-threading support. Prior to this, Qemu implemented multi-core emulation by executing each emulated core in a round-robin fashion — thereby serializing execution. This ensured that no interleaving could occur during execution of a single guest instruction, effectively making all instructions atomic. This is similar to the global serial execution functionality in Renode, defined in Section 2.2.3. Due to this serialization, Qemu could not scale past 32 threads because of the increasingly thinner time slices. COREMU improved this by parallelizing execution of guest cores, allowing it to emulate up to 255 threads.

Previously, Qemu implemented atomics without any synchronization, which was correct due to their serialized execution. When COREMU parallelized execution, this necessitated defining new locking mechanisms to ensure the correctness of atomic instructions. COREMU uses *lightweight memory transactions* to implement atomics by using a well-known algorithm called *Multi-Word Compare and Swap* (CASN). It is an extension of CAS with the ability to conditionally update N memory locations atomically. Still, depending on implementation, the ABA problem was not eliminated.

PQEMU (Parallel Qemu) [7] was published at the same time as COREMU with the same goal of parallelizing Qemu. They proposed a weak atomicity model — incorrect in the presence of stores — by serializing LR/SC and guarding them with a host mutex.

In 2016, RIGO [11] addressed the ABA problem observed in COREMU. RIGO proposed a scheme where LR/SC instrumentation is handled in Qemu's softMMU, similar to what Renode previously attempted (see Section 4.1.2). What RIGO does differently from Renode's failed softMMU-based approach is that it marks pages as *exclusive* whenever they contain a reserved address. This allows RIGO to ensure that even the stores which bypass the softMMU are able to invalidate reservations.

PICO [9], in 2017, also addresses the ABA problem of COREMU. They propose the PICO-ST scheme, where a bitmap and hash table are used to keep track of reservations. PICO-ST is similar to HST [2], with the addition of a bitmap. The bitmap is meant to be an efficient preview of the hash table, indicating whether to attempt to acquire the lock for specific hash table accesses. Another proposed scheme is PICO-HTM, where Hardware Transactional Memory (HTM) can emulate LL/SC, but requires specific hardware support.

In 2021, Zhao et al. [2] improved upon RIGO and PICO by implementing the HST scheme using TCG IR instead of helpers. They claim that helpers incur "extremely heavy runtime overheads" and that is why their HST solution performs better. Furthermore, they improve upon PICO-HTM, proposing HST-HTM. Their

improvement is that HST-HTM only places `SC` in a transaction, eliminating the livelock that PICO-HTM may encounter since it surrounds the entire LL/SC region in a transaction.

We further develop the HST scheme of Zhao et al. by improving correctness and adjustability. To remain portable across multiple hardware platforms, we avoid HTM-based schemes.

6.3 Future Work

Several questions remain. Some remain unexplored, while others arise from our results. This section introduces these questions, such that they may fuel further research.

6.3.1 Dynamic Hash Table Size

We evaluate different static hash table sizes, set before the virtual machine is created, but we do not explore varying the size during runtime. It may be viable to use a dynamically sized hash table, to automatically adjust based on runtime needs.

Dynamic sizing would allow table memory usage to start very low, leading to low memory usage in programs that do not encounter many hash collisions. Furthermore, it would allow the table size to scale to a larger size than would have been set statically, for programs that encounter many collisions.

6.3.2 Hash Table Entry Size

While we explore varying the total number of entries in the hash table, we do not evaluate different entry sizes. In our implementation, they are fixed at 8 bytes.

It may be the case that adding additional padding, increasing entries from 8 bytes up to e.g. 64 bytes, improves performance. This is due to the size of the processor's cache lines, the fixed size at which values are loaded from memory into cache. Having multiple cores accessing the same cache line leads to contention, requiring synchronization through the cache coherency protocol, slowing down execution. Therefore, having a lower number of table entries, with each entry being the same size as a cache line, may lead to improved performance.

6.3.3 Contention Backoff

Backoff is a technique utilized in ordinary spinlock implementations, which we do not use in our implementation. Its use may reduce contention, improving performance in contended scenarios. As our results show that our implementation performs poorly under extreme contention, this may remediate that.

The backoff consists of having spinning threads, trying to acquire the lock, eventually yield back to the scheduler. This allows other threads to execute, potentially improving performance by not wasting processor cycles waiting on a lock. The thread

yields its current timeslice, freeing it up for other threads. The scheduler may then allocate another timeslice to the thread at a later time, at which point the lock may be available.

As yielding to the scheduler involves a system call, i.e. significant overhead, another option may be to busy-wait with an exponential backoff. If the spinlock is unable to acquire the lock after some number of attempts, a busy-wait backoff could be performed rather than yielding. This involves not trying to acquire the lock for some number of cycles, instead spending that time busy-waiting in another loop. This reduces contention on the lock itself, which may improve performance. The number of cycles spent in the busy-wait loop could be exponentially increased, a common strategy in spinlock backoff systems [30]–[32].

6.3.4 Forward Progress Guarantee

We have shown that the HST scheme exhibits a risk of livelocking, which breaks the forward progress guarantee of RISC-V. Implementing this may be useful in both improving performance and emulating systems with real-time constraints.

Further research into techniques to implement this guarantee is necessary. As this is a guarantee which should be provided by hardware implementations, it may be fruitful to apply the same techniques used in hardware to DBTs.

6.3.5 Compiling Helpers to IR

One of our results is that implementing the HST scheme with TCG IR, rather than helper functions, results in up to 34% better performance. This, however, does not mean implementing the entire emulator using IR is optimal. For instance, code readability and developers' ability to iterate on and improve the implementation may be affected negatively by IR. In some instances, outside of the most performance-critical regions, it may be a better choice to write the implementation using helpers.

Exploring ways to improve the performance of helper functions may be fruitful, potentially allowing developers to write C implementations that perform nearly as well as those written in IR. One, currently unexplored, alternative is to implement a compiler from C to TCG IR. This compiler could allow developers to write helper functions that can then be compiled into TCG IR — achieving performance and readability.

Another unexplored alternative is performing just-in-time linking of helpers. As IR is emitted into translation blocks by the DBT, it also emits calls to helper functions. Instead of emitting a call, it may be viable to inline the compiled helper code directly into the translation block. This would eliminate the overhead of the function call, while preserving the optimizations of the C compiler.

6.3.6 Distribution of Atomic Variables in Guest Memory

The distribution of atomic synchronization variables within guest memory is highly relevant to the performance of our implementation. This is because the hashing we

perform places nearby guest addresses into the same hash table entry. Hence, if most synchronization variables in guest memory are co-located then this will lead to unnecessarily many hash collisions and therefore spurious invalidations.

Analyzing the distribution of synchronization variables would give insight into whether most guest programs distribute them relatively evenly, or if they are co-located. This data could then be taken into account, to improve the hashing function.

One way to improve the hashing function, in case synchronization variables are co-located in memory, is to left-shift the guest address before applying our hashing function. Currently, we discard the lower three bits of the guest address to align the table entry. If two 32-bit synchronization variables are adjacent, then their addresses only differ in the third bit. Our current hash function would cause these two variables to collide in the hash table, no matter the hash table size. Left-shifting one bit before applying the hash function resolves this.

If the analysis states that synchronization variables are evenly distributed throughout guest memory, then the hashing function could be augmented to take this into account. Rather than focusing on the lower bits of the guest address, it could right-shift the guest address and take more of the most significant bits into account. This would result in synchronization addresses that are far apart in the address space being assigned to different hash table entries.

6.4 Conclusion

This section summarizes our findings, relating them to our research questions.

The findings provide a clear answer to *RQ1*: the HST scheme emulates atomics correctly in Renode. Even though Renode is a highly complex system, with several edge cases and complex interactions with systems such as interrupt handling and softMMU faults, we were able to successfully implement the HST scheme for correct LR/SC emulation.

Furthermore, we found that implementing the HST scheme of Zhao et al. [2] directly in Renode, as described in their paper, results in a race condition. We show the existence of this race condition by means of model checking, and then describe how we prevent it from occurring in our implementation.

In regards to preserving performance scaling of multi-core guest code — research question *RQ2* — we find success. Our HST-strong implementations are able to scale well as the number of cores increase, while not sacrificing correctness.

We find that implementing the same HST-strong scheme using TCG IR results in 34% better performance in microbenchmarks than the same implementation using helper functions. In real-world benchmarks, the IR advantage shrinks to 6%–18%. This answers research question *RQ3* in the affirmative. However, as discussed in 6.3.5, this does not preclude the possibility of improving helper function performance to reach parity with IR in the future.

Regarding *RQ4*, we conclude that there is a noticeable performance impact in making the hash table size too small. Notably, we find that there are diminishing returns in increasing the table size beyond a certain point. The range we believe effectively balances size and performance in most scenarios is 512 KiB–8 MiB.

In conclusion, implementing the HST scheme in Renode results in fewer instances of incorrect emulation and better multi-core performance scaling.

7

Ethics

No major ethical issues were uncovered during the work presented in this thesis. The main contribution of this thesis is a technical improvement on an existing emulation tool, by means of applying publicly available research in a new context. Like with any other tool, the emulator we improve in this thesis may be utilized for ethically positive, neutral or questionable work.

We have no reason to believe that our work uniquely contributes to ethically questionable use cases. We therefore posit that our work is ethically neutral. It is up to the user what they do using this tool, but our contributions do not specifically provide any additional support for ethically questionable work.

Bibliography

[1] Antmicro, *Renode*, en. [Online]. Available: `https://renode.io/about/` (visited on 2025-02-27).

[2] Z. Zhao, Z. Jiang, Y. Chen, X. Gong, W. Wang, and P.-C. Yew, "Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 351–362. DOI: `10.1109/CGO51591.2021.9370312`.

[3] X. Zhan, Y. Bao, C. Bienia, and K. Li, "PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 5, pp. 1–16, 2017.

[4] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java just in time," *IEEE micro*, vol. 17, no. 3, pp. 36–43, 1997.

[5] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," *Document Version*, 2024. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[6] *ABA problem*, en, Page Version ID: 1254389542, 2024-10. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=ABA_problem&oldid=1254389542` (visited on 2025-03-04).

[7] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung, "PQEMU: A parallel system emulator based on QEMU," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, IEEE, 2011, pp. 276–283.

[8] Z. Wang, R. Liu, Y. Chen, *et al.*, "COREMU: a scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011, pp. 213–222.

[9] E. G. Cota, P. Bonzini, A. Bennée, and L. P. Carloni, "Cross-ISA machine emulation for multicores," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2017, pp. 210–220.

[10] Z. Zhao, Z. Jiang, X. Liu, X. Gong, W. Wang, and P.-C. Yew, "Dqemu: A scalable emulator with retargetable dbt on distributed platforms," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[11] A. Rigo, A. Spyridakis, and D. Raho, "Atomic Instruction Translation Towards A Multi-Threaded QEMU.," in *ECMS*, 2016, pp. 587–595.

[12] H. Gao, Y. Fu, and W. H. Hesselink, "Practical lock-free implementation of ll/sc using only pointer-size cas," in *2009 First International Conference on Information Science and Engineering*, IEEE, 2009, pp. 320–323.

[13] M. M. Michael, "Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS," in *International Symposium on Distributed Computing*, Springer, 2004, pp. 144–158.

[14] A. Waterman, K. Asanovic, *et al.*, "The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture," *Document Version*, vol. 20240411, pp. 1–4, 2024. [Online]. Available: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications.

[15] P. Mundkur, *GitHub - riscv/sail-riscv: Sail RISC-V model*, en. [Online]. Available: https://github.com/riscv/sail-riscv (visited on 2025-03-06).

[16] Antmicro, *Renode source code*, 2025-05. [Online]. Available: https://github.com/renode/renode (visited on 2025-05-17).

[17] Antmicro, *RISC-V vector instructions support in Renode*, en. [Online]. Available: https://renode.io/news/riscv-vector-instructions-in-renode/ (visited on 2025-05-20).

[18] S. F. Conservancy, *QEMU*. [Online]. Available: https://www.qemu.org/ (visited on 2025-03-04).

[19] Antmicro, *Time framework - Renode - documentation*. [Online]. Available: https://renode.readthedocs.io/en/latest/advanced/time_framework.html (visited on 2025-05-20).

[20] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[21] REMS, *Sail*, 2018-01. [Online]. Available: https://github.com/rems-project/sail (visited on 2025-06-11).

[22] B. Foundation, *BeagleV®-Fire*, en-US. [Online]. Available: https://www.beagleboard.org/boards/beaglev-fire (visited on 2025-05-13).

[23] Robot Framework Foundation, *Robot Framework*. [Online]. Available: https://robotframework.org/ (visited on 2025-04-04).

[24] ARM, *Arm® Architecture Reference Manual: for A-profile architecture*, version L.a, 2024-11-30. [Online]. Available: https://developer.arm.com/documentation/ddi0487/la/?lang=en.

[25] ARM, *Armv8-A memory model*, version 1.0, 2019-04-01. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20memory%20model%20guide.pdf?revision=58b1dd0a-3800-4218-b21a-f95a0332034c.

[26] C. Blundell, E. C. Lewis, and M. Martin, "Deconstructing transactional semantics: The subtleties of atomicity," in *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2005, pp. 48–55.

[27] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[28] Z. Zhou, Y. Bi, J. Wan, Y. Zhou, and Z. Li, "Userspace bypass: Accelerating syscall-intensive applications," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 33–49.

[29] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, California, USA, vol. 41, 2005, pp. 10–5555.

[30] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[31] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors.," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[32] J. Corbet, *Improving ticket spinlocks*, en, 2013. [Online]. Available: `https://lwn.net/Articles/531254/` (visited on 2025-06-17).

A

Promela Models

Listing 2: The Promela model for **LR/SC Invalidation by Store**.

```
mtype:sc_result = {success, failure};

int initial_value = 0;

/*
 * Globally accessible state.
 */

// A global shared-memory variable.
int shared_variable = initial_value;

// The thread ID of the thread which last wrote to the shared variable.
// Note that this represents the hash table entry for the address of
// the global shared variable.
int last_written_to_by_thread = 4294967295; // 0xFFFFFFFF in decimal

/*
 * Thread-local state. Stored as arrays with per-thread values.
 */
// Keeps track of a thread's current reservation. To minimize the
// state space, we use a bool instead of an address since there's
// only one memory location in this model.
// This is equivalent to `llsc_addr` from the HST paper.
bool is_address_reserved[2] = false;

// Emulates the hashing operation which would normally be necessary.
inline htable_hash() {
    // Perform one noop to represent the computational steps required
    // for hashing the address.
```

```
30      skip;
31  }
32
33  // Marks the address as stored to by the given thread. Note that
34  // to minimize the state space there is only one possible address
35  // and therefore it doesn't need to be provided.
36  inline htable_set(thread_id) {
37      htable_hash();
38      last_written_to_by_thread = thread_id;
39  }
40
41  // Checks whether the address is reserved by the given thread.
42  inline htable_check(result, thread_id) {
43      htable_hash();
44      result = last_written_to_by_thread == thread_id;
45  }
46
47  // Executes a store instruction.
48  inline sw(value, thread_id) {
49      // As described in the HST paper, these two steps are not
        ↪ performed
50      // surrounded by a lock, which is what causes the
        ↪ incorrectness.
51      // To fix it, these two steps need to execute atomically.
52      //atomic {
53          htable_set(thread_id); // Causes reservation invalidation.
54          shared_variable = value;
55      //}
56  }
57
58  // Executes an LR instruction.
59  inline lr(variable, thread_id) {
60      // Reserve address.
61      is_address_reserved[thread_id] = true;
62      // Update hash table.
63      htable_set(thread_id);
64      // Load current value.
65      variable = shared_variable;
66  }
67
68  // Executes an SC instruction.
69  inline sc(result, value, thread_id) {
70      if
71      :: is_address_reserved[thread_id] ->
72          // Using an atomic block here to replace the
            ↪ start_exclusive()
```

```
73          // and end_exclusive() calls. This assumes that those are
74          // completely correct and result in correct atomicity.
75          atomic {
76              bool check_result;
77              htable_check(check_result, thread_id);
78              if
79              :: check_result ->
80                  // Reservation is valid, perform store and
                    ↪ succeed.
81                  sw(value, thread_id);
82                  result = success;
83              :: else ->
84                  // Reservation was invalidated, so SC fails.
85                  result = failure;
86              fi
87          }
88      :: else ->
89          // Address is not reserved, so this SC cannot pair to
            ↪ anything.
90          result = failure;
91      fi
92  }
93
94  // Thread performing LR followed by an increment and an SC,
95  // retrying if the SC fails.
96  active proctype tlib_thread_lrsc() {
97      int thread_id = 0;
98
99      do
100     :: true ->
101         int value;
102         lr(value, thread_id);
103
104         int new_value = value + 1;
105
106         mtype:sc_result result;
107         sc(result, new_value, thread_id);
108
109         // Retry LR/SC until success.
110         if
111         :: result == success -> break;
112         :: else;
113         fi
114     od
115 end:
116 }
```

```
117
118
119 // Thread performing a store to a shared variable, which may
120 // cause a reservation in other threads to be invalidated.
121 active proctype tlib_thread_store() {
122     int thread_id = 1;
123
124     sw(1234, thread_id);
125 end:
126 }
127
128
129 ltl scEventuallySucceeds { <>tlib_thread_lrsc@end }
130 ltl sharedVariableContainsValue {
131     [](
132        tlib_thread_lrsc@end && tlib_thread_store@end
133        -> (shared_variable == 1234 || shared_variable == 1235)
134     )
135 }
```

Listing 3: The Promela model for **LR/SC Livelock**.

```
1  #define THREADS 2 /* number of threads */
2
3  mtype:sc_result = {success, failure};
4
5  int initial_value = 0;
6
7  /*
8   * Globally accessible state.
9   */
10
11 // A global shared-memory variable.
12 int shared_variable = initial_value;
13
14 // The thread ID of the thread which last wrote to the shared
    ↪  variable.
15 // Note that this represents the hash table entry for the address
    ↪  of
16 // the global shared variable.
17 int last_written_to_by_thread = 4294967295; // 0xFFFFFFFF in
    ↪  decimal
18
19 // Whether thread N is done executing.
20 bool done[THREADS] = false;
21
22 /*
```

```
23  * Thread-local state. Stored as arrays with per-thread values.
24  */
25 // Keeps track of a thread's current reservation. To minimize the
26 // state space, we use a bool instead of an address since there's
27 // only one memory location in this model.
28 // This is equivalent to `llsc_addr` from the HST paper.
29 bool is_address_reserved[THREADS] = false;
30
31 // Emulates the hashing operation which would normally be
    ↪  necessary.
32 inline htable_hash() {
33     // Perform one noop to represent the computational steps
       ↪  required
34     // for hashing the address.
35     skip;
36 }
37
38 // Marks the address as stored to by the given thread. Note that
39 // to minimize the state space there is only one possible address
40 // and therefore it doesn't need to be provided.
41 inline htable_set(thread_id) {
42     htable_hash();
43     last_written_to_by_thread = thread_id;
44 }
45
46 // Checks whether the address is reserved by the given thread.
47 inline htable_check(result, thread_id) {
48     htable_hash();
49     result = last_written_to_by_thread == thread_id;
50 }
51
52 // Executes a store instruction.
53 inline sw(value, thread_id) {
54     htable_set(thread_id); // Causes reservation invalidation.
55     shared_variable = value;
56 }
57
58 // Executes an LR instruction.
59 inline lr(variable, thread_id) {
60     // Reserve address.
61     is_address_reserved[thread_id] = true;
62     // Update hash table.
63     htable_set(thread_id);
64     // Load current value.
65     variable = shared_variable;
66 }
```

```
67
68  // Executes an SC instruction.
69  inline sc(result, value, thread_id) {
70      if
71      :: is_address_reserved[thread_id] ->
72          // Using an atomic block here to replace the
            ↪ start_exclusive()
73          // and end_exclusive() calls. This assumes that those are
74          // completely correct and result in correct atomicity.
75          atomic {
76              bool check_result;
77              htable_check(check_result, thread_id);
78              if
79              :: check_result ->
80                  // Reservation is valid, perform store and
                    ↪ succeed.
81                  sw(value, thread_id);
82                  result = success;
83              :: else ->
84                  // Reservation was invalidated, so SC fails.
85                  result = failure;
86              fi
87          }
88      :: else ->
89          // Address is not reserved, so this SC cannot pair to
            ↪ anything.
90          result = failure;
91      fi
92  }
93
94  // Thread performing LR followed by an increment and an SC,
95  // retrying if the SC fails.
96  proctype tlib_thread_lrsc(int thread_id) {
97      do
98      :: true ->
99          int value;
100         lr(value, thread_id);
101
102         int new_value = value + 1;
103
104         mtype:sc_result result;
105         sc(result, new_value, thread_id);
106
107         // Retry LR/SC until success.
108         if
109         :: result == success -> break;
```

```
        :: else;
        fi
    od

    done[thread_id] = true;
}

/*
 * Initialization
 */
init {
  byte i = 0; /* initialize the threads */
  do
  :: i >= THREADS -> break;
  :: else   -> run tlib_thread_lrsc(i); i++;
  od;
}

ltl scEventuallySucceeds { <>(done[0] && done[1]) }
ltl sharedVariableContainsValue {
    [](
       done[0] && done[1] -> shared_variable == 2
    )
}
```

Listing 4: The Promela model for **LR/SC Racing**.

```
#define THREADS 2 /* number of threads */

mtype:sc_result = {success, failure, none};

int initial_value = 0;

/*
 * Globally accessible state.
 */

// A global shared-memory variable.
int shared_variable = initial_value;

// The thread ID of the thread which last wrote to the shared
↪  variable.
// Note that this represents the hash table entry for the address
↪  of
// the global shared variable.
int last_written_to_by_thread = 4294967295; // 0xFFFFFFFF in
↪  decimal
```

```
18
19  // The final result of the SC.
20  mtype:sc_result sc_results[THREADS] = none;
21
22  /*
23   * Thread-local state. Stored as arrays with per-thread values.
24   */
25  // Keeps track of a thread's current reservation. To minimize the
26  // state space, we use a bool instead of an address since there's
27  // only one memory location in this model.
28  // This is equivalent to `llsc_addr` from the HST paper.
29  bool is_address_reserved[THREADS] = false;
30
31  // Emulates the hashing operation which would normally be
    ↪  necessary.
32  inline htable_hash() {
33      // Perform one noop to represent the computational steps
        ↪  required
34      // for hashing the address.
35      skip;
36  }
37
38  // Marks the address as stored to by the given thread. Note that
39  // to minimize the state space there is only one possible address
40  // and therefore it doesn't need to be provided.
41  inline htable_set(thread_id) {
42      htable_hash();
43      last_written_to_by_thread = thread_id;
44  }
45
46  // Checks whether the address is reserved by the given thread.
47  inline htable_check(result, thread_id) {
48      htable_hash();
49      result = last_written_to_by_thread == thread_id;
50  }
51
52  // Executes a store instruction.
53  inline sw(value, thread_id) {
54      htable_set(thread_id); // Causes reservation invalidation.
55      shared_variable = value;
56  }
57
58  // Executes an LR instruction.
59  inline lr(variable, thread_id) {
60      // Reserve address.
61      is_address_reserved[thread_id] = true;
```

```
62      // Update hash table.
63      htable_set(thread_id);
64      // Load current value.
65      variable = shared_variable;
66  }
67
68  // Executes an SC instruction.
69  inline sc(result, value, thread_id) {
70      if
71      :: is_address_reserved[thread_id] ->
72          // Using an atomic block here to replace the
            ↪ start_exclusive()
73          // and end_exclusive() calls. This assumes that those are
74          // completely correct and result in correct atomicity.
75          atomic {
76              bool check_result;
77              htable_check(check_result, thread_id);
78              if
79              :: check_result ->
80                  // Reservation is valid, perform store and
                    ↪ succeed.
81                  sw(value, thread_id);
82                  result = success;
83              :: else ->
84                  // Reservation was invalidated, so SC fails.
85                  result = failure;
86              fi
87          }
88      :: else ->
89          // Address is not reserved, so this SC cannot pair to
            ↪ anything.
90          result = failure;
91      fi
92  }
93
94  // Thread performing LR followed by an increment and an SC,
95  // NOT retrying if the SC fails (since that will lead to
    ↪ livelock).
96  proctype tlib_thread_lrsc(int thread_id) {
97      int value;
98      lr(value, thread_id);
99
100     int new_value = value + 1;
101
102     mtype:sc_result result;
103     sc(result, new_value, thread_id);
```

```
104
105     sc_results[thread_id] = result;
106 }
107
108 /*
109  * Initialization
110  */
111 init {
112   byte i = 0; /* initialize the threads */
113   do
114   :: i >= THREADS -> break;
115   :: else   -> run tlib_thread_lrsc(i); i++;
116   od;
117 }
118
119 ltl scEventuallySucceeds {
120     <>(
121         (sc_results[0] == success && sc_results[1] == failure)
122         || (sc_results[0] == failure && sc_results[1] == success)
123         || (sc_results[0] == success && sc_results[1] == success)
124     )
125 }
126 ltl bothScNeverFail {
127     [](sc_results[0] != failure || sc_results[1] != failure)
128 }
129 ltl sharedVariableContainsValue {
130     [](
131        (sc_results[0] == success && sc_results[1] == success ->
        ↪  shared_variable == 2)
132        || (sc_results[0] == success && sc_results[1] == failure ->
        ↪  shared_variable == 1)
133        || (sc_results[0] == failure && sc_results[1] == success ->
        ↪  shared_variable == 1)
134     )
135 }
```

Listing 5: The Promela model for **LR/SC + CAS**.

```
1 mtype:sc_result = {success, failure};
2
3 int initial_value = 0;
4
5 /*
6  * Globally accessible state.
7  */
8
9 // A global shared-memory variable.
```

```
10 int shared_variable = initial_value;
11
12 // The thread ID of the thread which last wrote to the shared
    ↪ variable.
13 // Note that this represents the hash table entry for the address
    ↪ of
14 // the global shared variable.
15 int last_written_to_by_thread = 4294967295; // 0xFFFFFFFF in
    ↪ decimal
16
17 /*
18  * Thread-local state. Stored as arrays with per-thread values.
19  */
20 // Keeps track of a thread's current reservation. To minimize the
21 // state space, we use a bool instead of an address since there's
22 // only one memory location in this model.
23 // This is equivalent to `llsc_addr` from the HST paper.
24 bool is_address_reserved[2] = false;
25
26 // The value which was present at the shared memory address when
27 // the reservation was created.
28 int reserved_value[2] = 0;
29
30 // Executes a store instruction.
31 inline sw(value, thread_id) {
32     // As described in the HST paper, these two steps are not
        ↪ performed
33     // surrounded by a lock, which is what causes the
        ↪ incorrectness.
34     // To fix it, these two steps need to execute atomically.
35     //atomic {
36         htable_set(thread_id); // Causes reservation invalidation.
37         shared_variable = value;
38     //}
39 }
40
41 // An atomic compare-and-swap (CAS) operation,
42 // which takes in an expected value and if the value at the memory
    ↪ address
43 // is equal to it, stores the new value in its place. The old
    ↪ memory
44 // value is written to the 'actual' variable in either case.
45 inline compare_and_swap(actual, expected, new, thread_id) {
46     atomic {
47         actual = shared_variable;
48         if
```

```
49          :: shared_variable == expected -> sw(new, thread_id);
50          :: else -> skip;
51          fi
52      }
53  }
54
55  // Emulates the hashing operation which would normally be
    ↪ necessary.
56  inline htable_hash() {
57      // Perform one noop to represent the computational steps
        ↪ required
58      // for hashing the address.
59      skip;
60  }
61
62  // Marks the address as stored to by the given thread. Note that
63  // to minimize the state space there is only one possible address
64  // and therefore it doesn't need to be provided.
65  inline htable_set(thread_id) {
66      htable_hash();
67      last_written_to_by_thread = thread_id;
68  }
69
70  // Checks whether the address is reserved by the given thread.
71  inline htable_check(result, thread_id) {
72      htable_hash();
73      result = last_written_to_by_thread == thread_id;
74  }
75
76  // Executes an LR instruction.
77  inline lr(result, thread_id) {
78      // Reserve address.
79      is_address_reserved[thread_id] = true;
80      // Update hash table.
81      htable_set(thread_id);
82      // Load current value.
83      result = shared_variable;
84      // Update reserved value.
85      reserved_value[thread_id] = result;
86  }
87
88  // Executes an SC instruction.
89  inline sc(result, value, thread_id) {
90      if
91      :: is_address_reserved[thread_id] ->
```

```
92          // Using an atomic block here to replace the
            ↪  start_exclusive()
93          // and end_exclusive() calls. This assumes that those are
94          // completely correct and result in correct atomicity.
95          atomic {
96              bool check_result;
97              htable_check(check_result, thread_id);
98              if
99              :: check_result ->
100                 // Reservation is valid, try to perform store.
101                 int reserved = reserved_value[thread_id];
102                 int actual;
103                 compare_and_swap(actual, reserved, value,
                    ↪  thread_id);
104                 // Check if CAS succeeded.
105                 if
106                 :: actual == reserved -> result = success;
107                 :: else -> result = failure;
108                 fi
109                 /*sw(value, thread_id);
110                  result = success;*/
111             :: else ->
112                 // Reservation was invalidated, so SC fails.
113                 result = failure;
114             fi
115         }
116     :: else ->
117         // Address is not reserved, so this SC cannot pair to
            ↪  anything.
118         result = failure;
119     fi
120 }
121
122 // Thread performing LR followed by an increment and an SC,
123 // retrying if the SC fails.
124 active proctype tlib_thread_lrsc() {
125     int thread_id = 0;
126
127     do
128     :: true ->
129         int value;
130         lr(value, thread_id);
131
132         int new_value = value + 1;
133
134         mtype:sc_result store_result;
```

```
        sc(store_result, new_value, thread_id);

        // Retry LR/SC until success.
        if
        :: store_result == success -> break;
        :: else;
        fi
    od
end:
}


// Thread performing a store to a shared variable, which may
// cause a reservation in other threads to be invalidated.
active proctype tlib_thread_store() {
    int thread_id = 1;

    sw(1234, thread_id);
end:
}


ltl scEventuallySucceeds { <>tlib_thread_lrsc@end }
ltl sharedVariableContainsValue {
    [](
       tlib_thread_lrsc@end && tlib_thread_store@end
       -> (shared_variable == 1234 || shared_variable == 1235)
    )
}
```

Listing 6: The Promela model for **Lock-free Stack ABA**.

```
#define THREADS 3
#define NODE_COUNT 3
// todo: reenable #define USE_CAS

mtype:sc_result = {success, failure};

// A node in a linked list.
typedef node {
    // The value stored in this node.
    int value;
    // The index of the next node (-1 for null).
    int next_index;
}

/*
```

```
16  * Globally accessible state.
17  */
18
19  // A pointer to (i.e. index into memory-array) the start of the
    ↪  global shared-memory stack.
20  int head_index = -1;
21
22  // Size arbitrarily chosen to fit the operations performed.
23  #define MEMORY_ELEMENTS 10
24
25  // An emulation of malloc, allowing threads to create new
    ↪  heap-allocated nodes.
26  node memory[MEMORY_ELEMENTS];
27  // Points at the index of an empty, unallocated spot in the
    ↪  memory-array.
28  int malloc_empty_index = 0;
29
30  // The thread ID of the thread which last wrote to the shared
    ↪  variable.
31  // Note that this represents the hash table entry for the address
    ↪  of
32  // the global shared variable.
33  int last_written_to_by_thread = 4294967295; // 0xFFFFFFFF in
    ↪  decimal
34
35  /*
36  * Thread-local state. Stored as arrays with per-thread values.
37  */
38
39  // Keeps track of a thread's current reservation. To minimize the
40  // state space, we use a bool instead of an address since there's
41  // only one memory location in this model.
42  // This is equivalent to `llsc_addr` from the HST paper.
43  bool is_address_reserved[THREADS] = false;
44
45  // The value which was present at the shared memory address when
46  // the reservation was created.
47  int reserved_value[THREADS] = 0;
48
49  // Whether thread N is done executing.
50  bool done[THREADS] = false;
51
52  /*
53  * Instruction implementations.
54  */
55
```

```
56 // Executes a store instruction.
57 inline sw(value, thread_id) {
58
59
60     // As described in the HST paper, these two steps are not
       ↪ performed
61     // surrounded by a lock, which is what causes the
       ↪ incorrectness.
62     // To fix it, these two steps need to execute atomically.
63     //atomic {
64         htable_set(thread_id); // Causes reservation invalidation.
65         head_index = value;
66     //}
67 }
68
69 // An atomic compare-and-swap (CAS) operation,
70 // which takes in an expected value and if the value at the memory
   ↪ address
71 // is equal to it, stores the new value in its place. The old
   ↪ memory
72 // value is written to the 'actual' variable in either case.
73 inline compare_and_swap(actual, expected, new, thread_id) {
74     atomic {
75         actual = head_index;
76         if
77         :: actual == expected -> sw(new, thread_id);
78         :: else -> skip;
79         fi
80     }
81 }
82
83 // Emulates the hashing operation which would normally be
   ↪ necessary.
84 inline htable_hash() {
85     // Perform one noop to represent the computational steps
       ↪ required
86     // for hashing the address.
87     skip;
88 }
89
90 // Marks the address as stored to by the given thread. Note that
91 // to minimize the state space there is only one possible address
92 // and therefore it doesn't need to be provided.
93 inline htable_set(thread_id) {
94     htable_hash();
95     last_written_to_by_thread = thread_id;
```

```
96  }
97
98  // Checks whether the address is reserved by the given thread.
99  inline htable_check(result, thread_id) {
100     htable_hash();
101     result = last_written_to_by_thread == thread_id;
102 }
103
104 /*
105  * LR/SC implementations.
106  */
107
108 // Executes an LR instruction.
109 inline lr(result, thread_id) {
110     // Reserve address.
111     is_address_reserved[thread_id] = true;
112     // Update hash table.
113     htable_set(thread_id);
114     // Load current value.
115     result = head_index;
116     // Update reserved value.
117     reserved_value[thread_id] = result;
118 }
119
120 // Executes an SC instruction.
121 inline sc(result, value, thread_id) {
122     if
123     :: is_address_reserved[thread_id] ->
124         // Using an atomic block here to replace the
            ↪ start_exclusive()
125         // and end_exclusive() calls. This assumes that those are
126         // completely correct and result in correct atomicity.
127         atomic {
128             bool check_result;
129             htable_check(check_result, thread_id);
130             if
131             :: check_result ->
132                 #ifdef USE_CAS
133
134                 // Reservation is valid, try to perform store.
135                 int reserved = reserved_value[thread_id];
136                 int actual;
137                 compare_and_swap(actual, reserved, value,
                    ↪ thread_id);
138                 // Check if CAS succeeded.
139                 if
```

```
140                 :: actual == reserved -> result = success;
141                 :: else -> result = failure;
142                 fi
143
144                 #else
145
146                 sw(value, thread_id);
147                 result = success;
148
149                 #endif
150             :: else ->
151                 // Reservation was invalidated, so SC fails.
152                 result = failure;
153             fi
154         }
155     :: else ->
156         // Address is not reserved, so this SC cannot pair to
        ↪ anything.
157         result = failure;
158     fi
159 }
160
161 /*
162  * Malloc implementation.
163  */
164 inline malloc(result_index) {
165     // We don't care about testing the atomicity of this
    ↪ implementation, so
166     // just force it to be done atomically.
167     atomic {
168         // Initialize new node.
169         memory[malloc_empty_index].value = -1;
170         memory[malloc_empty_index].next_index = -1;
171         // Return newly allocated index.
172         result_index = malloc_empty_index;
173         // Increment malloc pointer.
174         malloc_empty_index++;
175     }
176 }
177
178 /*
179  * Stack operations.
180  */
181
182 // Pops an element from the global stack in a thread-safe manner
    ↪ using LR/SC.
```

```
183 inline pop(result, thread_id) {
184     do
185     :: true ->
186         int local_head_index;
187         lr(local_head_index, thread_id);
188 
189         result = memory[local_head_index].value;
190         int next_index = memory[local_head_index].next_index;
191 
192         mtype:sc_result store_result;
193         sc(store_result, next_index, thread_id);
194 
195         // Retry LR/SC until success.
196         if
197         :: store_result == success -> break;
198         :: else;
199         fi
200     od
201 }
202 
203 // Pushes an element to the top of the global stack in a
    ↪  thread-safe manner using LR/SC.
204 inline push(new_value, thread_id) {
205     int new_node_index;
206     malloc(new_node_index);
207     do
208     :: true ->
209         int local_head_index;
210         lr(local_head_index, thread_id);
211 
212         memory[new_node_index].value = new_value;
213         memory[new_node_index].next_index = local_head_index;
214 
215         mtype:sc_result store_result;
216         sc(store_result, new_node_index, thread_id);
217 
218         // Retry LR/SC until success.
219         if
220         :: store_result == success -> break;
221         :: else;
222         fi
223     od
224 }
225 
226 /*
227  * Processes.
```

```
228  */
229
230 proctype popper_pusher(int thread_id) {
231     int original_value = -2;
232     pop(original_value, thread_id);
233     push(original_value, thread_id);
234
235     done[thread_id] = true;
236 }
237
238 /*
239  * Initialization
240  */
241 init {
242     byte i = 0; /* initialize memory */
243     do
244     :: i >= MEMORY_ELEMENTS -> break;
245     :: else -> memory[i].value = -1; memory[i].next_index = -1;
        ↪  i++;
246     od;
247
248     // Set up stack as:
249     // head --> 8 --> 16 --> 32 --> null
250     push(32, 0);
251     push(16, 0);
252     push(8, 0);
253
254     i = 0; /* initialize the threads */
255     atomic {
256         do
257         :: i >= THREADS -> break;
258         :: else   -> run popper_pusher(i); i++;
259         od;
260     }
261 }
262
263 /*
264  * Formula helpers.
265  */
266
267 #define ALL_THREADS_DONE (done[0] && done[1] && done[2])
268
269 ltl threadsEventuallyComplete { <>ALL_THREADS_DONE }
270
271 #define STACK_CONTAINS(n) (memory[head_index].value == n \
272         || memory[memory[head_index].next_index].value == n \
```

```
273         || memory[memory[memory[head_index].next_index].next_inde
            ↪  x].value ==
            ↪  n)
274
275 // Ensure that at the end the stack contains all original values,
276 // though they may be reordered.
277 ltl stackNotSmashed {
278     [](
279         ALL_THREADS_DONE -> STACK_CONTAINS(8) && STACK_CONTAINS(16)
             ↪  && STACK_CONTAINS(32)
280     )
281 }
282
```

Listing 7: The Promela model for **Fine-grained LR/SC Locking**.

```
1  mtype:sc_result = {success, failure};
2
3  int initial_value = 0;
4
5  /*
6   * Globally accessible state.
7   */
8
9  // A global shared-memory variable.
10 int shared_variable = initial_value;
11
12 // The thread ID of the thread which last wrote to the shared
    ↪  variable.
13 // Note that this represents the hash table entry for the address
    ↪  of
14 // the global shared variable.
15 int last_written_to_by_thread = -1; // 0xFFFFFFFF in decimal
16
17 #define HST_UNLOCKED -1
18 // The thread ID of the thread which currently owns the lock to
    ↪  the shared variable.
19 // Note that this corresponds to the hash table entry fine-grained
    ↪  lock.
20 int locked_by_thread = HST_UNLOCKED;
21
22 /*
23  * Thread-local state. Stored as arrays with per-thread values.
24  */
25 // Keeps track of a thread's current reservation. To minimize the
26 // state space, we use a bool instead of an address since there's
27 // only one memory location in this model.
```

```
28 // This is equivalent to `llsc_addr` from the HST paper.
29 bool is_address_reserved[2] = false;
30
31 // An atomic compare-and-swap (CAS) operation,
32 // which takes in an expected value and if the value at the memory
    ↪ address
33 // is equal to it, stores the new value in its place. The old
    ↪ memory
34 // value is written to the 'actual' variable in either case.
35 // NOTE: does _not_ cause reservation invalidation.
36 inline compare_and_swap(actual, expected, new) {
37     atomic {
38         actual = locked_by_thread;
39         if
40         :: actual == expected -> locked_by_thread = new;
41         :: else -> skip;
42         fi
43     }
44 }
45
46 // Emulates the hashing operation which would normally be
    ↪ necessary.
47 inline htable_hash() {
48     // Perform one noop to represent the computational steps
        ↪ required
49     // for hashing the address.
50     skip;
51 }
52
53 // Marks the address as stored to by the given thread. Note that
54 // to minimize the state space there is only one possible address
55 // and therefore it doesn't need to be provided.
56 inline htable_set(thread_id) {
57     htable_hash();
58     last_written_to_by_thread = thread_id;
59 }
60
61 // Checks whether the address is reserved by the given thread.
62 inline htable_check(result, thread_id) {
63     htable_hash();
64     result = last_written_to_by_thread == thread_id;
65 }
66
67 // Acquires a fine-grained table entry lock.
68 // Note that we only model a single table entry, so it is
    ↪ unnecessary
```

```
69  // to provide an address for determining which entry to lock.
70  inline htable_lock(thread_id) {
71      htable_hash();
72      int expectedLock = HST_UNLOCKED;
73      // Spinlock acquire.
74      do
75      :: true ->
76          int actual;
77          compare_and_swap(actual, expectedLock, thread_id);
78
79          // Retry until CAS success.
80          if
81          :: actual == expectedLock -> break;
82          :: else;
83          fi
84      od
85  }
86
87  // Releases a fine-grained table entry lock.
88  // See the note in htable_lock.
89  inline htable_unlock(thread_id) {
90      htable_hash();
91      assert(locked_by_thread == thread_id);
92      locked_by_thread = HST_UNLOCKED;
93  }
94
95  // Executes a store instruction.
96  inline sw(value, thread_id) {
97      // As described in the HST paper, these two steps are not
        ↪ performed
98      // surrounded by a lock, which is what causes the
        ↪ incorrectness.
99      // To fix it, these two steps need to execute within a lock.
100     htable_lock(thread_id);
101     htable_set(thread_id); // Causes reservation invalidation.
102     shared_variable = value;
103     htable_unlock(thread_id);
104 }
105
106 // Executes an LR instruction.
107 inline lr(variable, thread_id) {
108     htable_lock(thread_id);
109     // Reserve address.
110     is_address_reserved[thread_id] = true;
111     // Update hash table.
112     htable_set(thread_id);
```

```
113     // Load current value.
114     variable = shared_variable;
115     htable_unlock(thread_id);
116 }
117 
118 // Executes an SC instruction.
119 inline sc(result, value, thread_id) {
120     if
121     :: is_address_reserved[thread_id] ->
122         htable_lock(thread_id);
123 
124         bool check_result;
125         htable_check(check_result, thread_id);
126         if
127         :: check_result ->
128             // Reservation is valid, perform store and succeed.
129             htable_set(thread_id); // Causes reservation
                ↪  invalidation.
130             shared_variable = value;
131             result = success;
132         :: else ->
133             // Reservation was invalidated, so SC fails.
134             result = failure;
135         fi
136 
137         htable_unlock(thread_id);
138     :: else ->
139         // Address is not reserved, so this SC cannot pair to
            ↪  anything.
140         result = failure;
141     fi
142 }
143 
144 // Thread performing LR followed by an increment and an SC,
145 // retrying if the SC fails.
146 active proctype tlib_thread_lrsc() {
147     int thread_id = 0;
148 
149     do
150     :: true ->
151         int value;
152         lr(value, thread_id);
153 
154         int new_value = value + 1;
155 
156         mtype:sc_result result;
```

```
157         sc(result, new_value, thread_id);
158
159         // Retry LR/SC until success.
160         if
161         :: result == success -> break;
162         :: else;
163         fi
164     od
165 end:
166 }
167
168
169 // Thread performing a store to a shared variable, which may
170 // cause a reservation in other threads to be invalidated.
171 active proctype tlib_thread_store() {
172     int thread_id = 1;
173
174     sw(1234, thread_id);
175 end:
176 }
177
178
179 ltl scEventuallySucceeds { <>tlib_thread_lrsc@end }
180 ltl sharedVariableContainsValue {
181     [](
182        tlib_thread_lrsc@end && tlib_thread_store@end
183        -> (shared_variable == 1234 || shared_variable == 1235)
184     )
185 }
```

B

Source Code

The six different Renode implementations referenced throughout this thesis, along with the benchmarking setups, are all available on Antmicro's GitHub. This appendix links to all of them.

B.1 Renode Source Code

Renode's source code consists of several Git submodules. The topmost repository is called Renode, which contains a submodule called Renode Infrastructure, which contains a submodule called tlib. Our main contributions lie in tlib, and Renode Infrastructure contains the hash table allocation logic. Here, we link to both the topmost repository (necessary to run the emulation) and the tlib repository.

- Current Renode:
  `github.com/renode/renode/tree/46ab57eff283b2654e7c3ecc6dd9130c5fd89bc0`
  `github.com/antmicro/tlib/tree/b682ca48bccddd88d97f2238fa01c126d09811cb`
- HST-weak IR:
  `github.com/renode/renode/tree/public-hst-weak-ir`
  `github.com/antmicro/tlib/tree/public-hst-weak-ir`
- HST-weak Helpers:
  `github.com/renode/renode/tree/public-hst-weak-helpers`
  `github.com/antmicro/tlib/tree/public-hst-weak-helpers`
- HST-strong IR:
  `github.com/renode/renode/tree/public-hst-strong-ir`
  `github.com/antmicro/tlib/tree/public-hst-strong-ir`
- HST-strong Helpers:
  `github.com/renode/renode/tree/public-hst-strong-helpers`
  `github.com/antmicro/tlib/tree/public-hst-strong-helpers`
- Renode without softMMU bypass:
  `github.com/renode/renode/tree/always-go-through-softmmu-qemu-st`
  `github.com/antmicro/tlib/tree/always-go-through-softmmu-qemu-st`

B.2 Benchmark Source Code

The PARSEC benchmarking setup is available at:
`github.com/antmicro/renode-parsec-benchmarking`.

The TLIB microbenchmarking setup is available at:
`github.com/antmicro/cpu-microbenchmarking`.

C

Race Condition Trace

Listing 8: The counterexample trace showing which interleaving of LR and store instructions leads to incorrectness. The model being run is shown in Listing 2.

```
[ -f ./hst-strong-incorrectness.pml.trail ] && spin -t -p -l -g -r -s ./hst-strong-incorrectness.pml || true
ltl scEventuallySucceeds: <> ((tlib_thread_lrsc@end))
ltl sharedVariableContainsValue: [] ((! (((tlib_thread_lrsc@end)) && ((tlib_thread_store@end)))) || (((shared_variable==1234)) || ((shared_variable==1235))))
starting claim 3
  1:    proc  1 (tlib_thread_store:1) ./hst-strong-incorrectness.pml:30 (state 1)       [(1)]
  2:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:100 (state 1)       [(1)]
  2:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:102 (state 2)       [value = 0]
                tlib_thread_lrsc(0):value = 0
  3:    proc  1 (tlib_thread_store:1) ./hst-strong-incorrectness.pml:38 (state 3) [last_written_to_by_thread = thread_id]
                last_written_to_by_thread = 1
  4:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:61 (state 3) [is_address_reserved[thread_id] = 1]
                is_address_reserved[0] = 1
                is_address_reserved[1] = 0
  5:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:30 (state 4)       [(1)]
  6:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:38 (state 6) [last_written_to_by_thread = thread_id]
                last_written_to_by_thread = 0
  7:    proc  0 (tlib_thread_lrsc:1) ./hst-strong-incorrectness.pml:65 (state 8)       [value = shared_variable]
```

```
18                tlib_thread_lrsc(0):value = 0
19   8:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:106 (state 10)      [new_value
     ↪  = (value+1)]
20                tlib_thread_lrsc(0):new_value = 1
21   8:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:107 (state 11)      [result =
     ↪  0]
22                tlib_thread_lrsc(0):result = 0
23                tlib_thread_lrsc(0):new_value = 1
24   9:    proc  1 (tlib_thread_store:1)
     ↪  ./hst-strong-incorrectness.pml:54 (state 5)
     ↪  [shared_variable = 1234]
25                shared_variable = 1234
26  10:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:71 (state 12)
     ↪  [(is_address_reserved[thread_id])]
27  11:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:77 (state 13)
     ↪  [check_result = 0]
28                tlib_thread_lrsc(0):check_result = 0
29  12:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:30 (state 14)       [(1)]
30  12:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:44 (state 16)
     ↪  [check_result = (last_written_to_by_thread==thread_id)]
31                tlib_thread_lrsc(0):check_result = 1
32  13:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:79 (state 18)
     ↪  [(check_result)]
33  14:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:30 (state 19)       [(1)]
34  14:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:38 (state 21)
     ↪  [last_written_to_by_thread = thread_id]
35  14:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:54 (state 23)
     ↪  [shared_variable = new_value]
36                shared_variable = 1
37  14:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:82 (state 25)       [result = 2]
38                shared_variable = 1
39                tlib_thread_lrsc(0):result = 2
40  15:    proc  0 (tlib_thread_lrsc:1)
     ↪  ./hst-strong-incorrectness.pml:111 (state 36)
     ↪  [((result==2))]
```

```
41 spin: trail ends after 15 steps
42 #processes: 2
43                 initial_value = 0
44                 shared_variable = 1
45                 last_written_to_by_thread = 0
46                 is_address_reserved[0] = 1
47                 is_address_reserved[1] = 0
48                 end = 0
49  15:    proc  1 (tlib_thread_store:1)
    ↪  ./hst-strong-incorrectness.pml:126 (state 7) <valid end state>
50                 tlib_thread_store(1):thread_id = 1
51  15:    proc  0 (tlib_thread_lrsc:1)
    ↪  ./hst-strong-incorrectness.pml:116 (state 44) <valid end state>
52                 tlib_thread_lrsc(0):check_result = 1
53                 tlib_thread_lrsc(0):result = 2
54                 tlib_thread_lrsc(0):new_value = 1
55                 tlib_thread_lrsc(0):value = 0
56                 tlib_thread_lrsc(0):thread_id = 0
57  15:    proc  - (sharedVariableContainsValue:1) _spin_nvr.tmp:9
    ↪  (state 6)
58 2 processes created
```

D

Additional Graphs

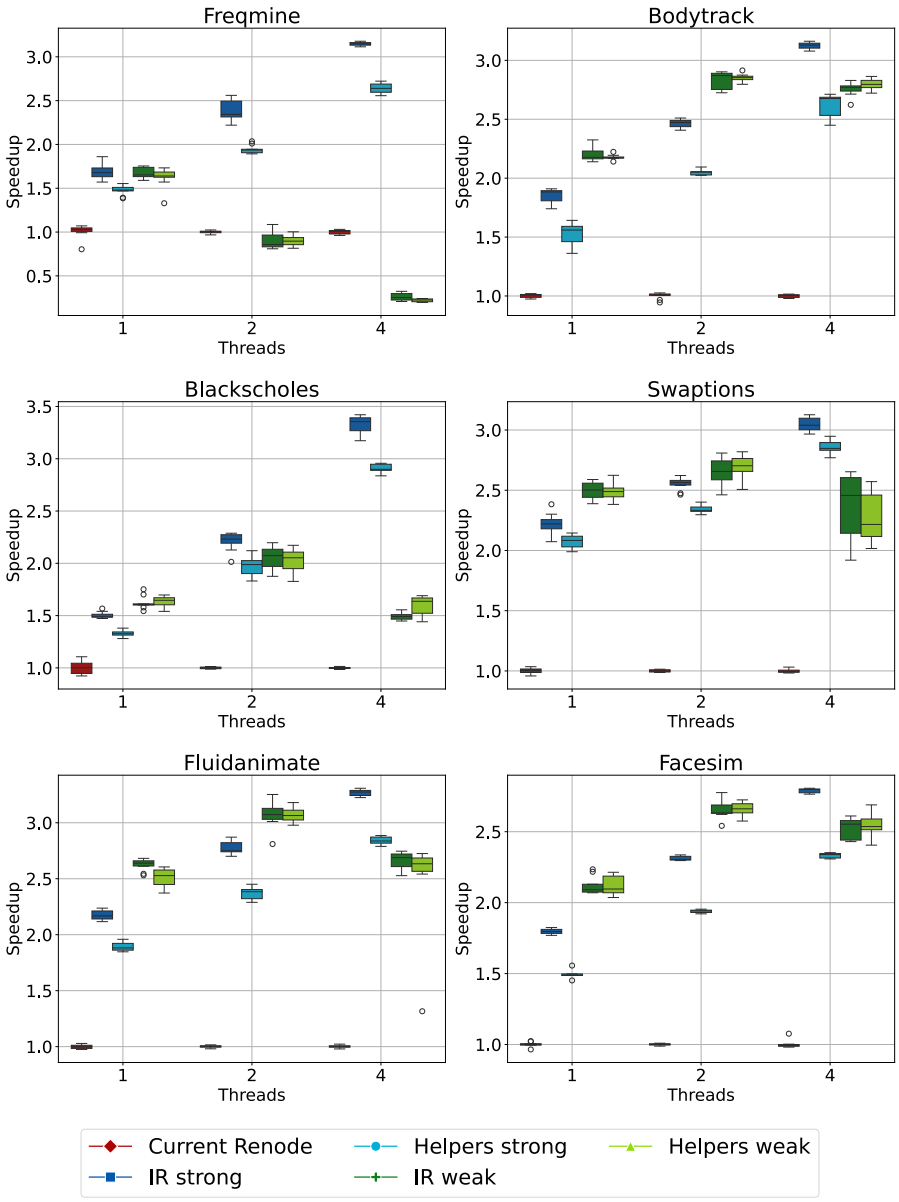


Figure D.1: Box plots of a set of PARSEC benchmarks on five versions of Renode. Speedups are computed relative to current Renode.