

Dissertation
Enhancing Full-System Simulation:
Techniques for Maximizing Performance and Accuracy

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Niko Zurstraßen, M.Sc.
aus Münster, Deutschland

Berichter: Universitätsprofessor Dr. rer. nat. Rainer Leupers
Universitätsprofessor Dr.-Ing. Matthias Jung

Tag der mündlichen Prüfung: 11.09.2025

Diese Dissertation ist auf den Internetseiten
der Universitätsbibliothek online verfügbar.

Abstract

Simulating compute systems by pure virtual means has become a cornerstone of modern hardware and software development. These virtual twins of computers, also referred to as Full-System Simulators (FSSs), enable a plethora of unique use cases. As virtual development platforms, FSSs enable the development of software long before any hardware prototypes are available, speeding up the time to market. When also incorporating microarchitectural details, FSSs facilitate early design space exploration by estimating a system's characteristics (performance, power consumption, cache hit/miss rates, etc.). Ultimately, all use cases share one thing in common: the FSS should be as fast as possible while still providing the required accuracy. As a software development platform, the performance directly affects the developer's productivity. When conducting design space exploration, the performance determines the number of explorable designs. Moreover, and much like its real-world counterpart, a FSS can never be fast enough.

In order to meet the ever-increasing demand for more performance, this thesis focuses on the development of methods that accelerate the execution of simulations. Similar to the phases of a compiler, many of the here presented challenges are orthogonal but contribute to the same goal. More specifically, this thesis first presents a parallelized version of the popular open-source FSS `gem5` (Chapter 3). By leveraging modern multi-core systems, the parallelized simulator attains speedups of up to $24.7\times$ when simulating multi-threaded benchmarks. Based on this, analytical models for performance and accuracy prediction are presented (Chapter 4). This is followed by introducing new methods for the fast simulation of (vector) floating point instructions (Chapter 5). By using the host FPU in a sophisticated way, individual instructions see speedups of up to $5\times$ compared to a soft float implementation. Lastly, a global, static register allocation for Dynamic Binary Translators (DBTs) is presented (Chapter 6). Compared to local register allocation methods, as used by the state-of-the-art FSS `QEMU`, the method of this thesis achieves speedups of up to $1.4\times$.

Acknowledgements

I guess it is time to thank the one and only person who facilitated the completion of this document: me, myself, and I. No, seriously, I'm pretty sure that if I would not have been writing it, no one else would have done it. I even asked ChatGPT, but it encouraged me to do it myself in order to ensure academic integrity. Anyway, ChatGPT also advised me to thank a whole of bunch people, so here were go. The biggest "thank you" goes to my supervisor Prof. Rainer Leupers for the guidance, support, and the courage to employ me. Also, big thanks to Prof. Matthias Jung for giving me very thorough feedback on my dissertation (people were impressed by the number of sticky notes you put into it). Moreover, I'm grateful for all my colleagues who make each day feel like an episode of *The Office*. I love all your quirks; please never change. In particular:

Chiara Ghinami, the most German German is actually Italian; Christoph Vogt, for always looking after my machines; Christoph Zörkner, for helping me to borrow DSP's 3D printer; Dominik Sisejkovic, for teaching me how to teach; Eva Gerhada Haas, for tales about the shallows of the university administrative apparatus; Evgenii Rezunov, for pushing weights and commits; Farhad Merchant, for teaching me that good Indian food in Europe is virtually non-existent; Felix Staudigl, you truly mastered the "Phd game"; Jan Moritz Joseph, for introducing me into the academic world; Jan Weinstock, still wondering if you are some kind of Coding Jesus sent by God; José Cubero-Cascante, for delivering a successful project; Lennart Reimann, words cannot describe you; Lorenzo Pfeifer, for teaching me how to update the operating system of a flashlight; Lukas Jünger, "sounds like a you-problem"; Melvin Galicia, the Bitcoin expert; Nils Bosbach, my BlueCard provider (please eat the creatine I gifted you); Ruben Brandhofer, for praising Lennart; Rebecca Pelke, for being the person who gets so many rejections that others don't need to feel bad; Tanja Palmen, for teaching me RWTHonline; I also have a wife (Ida), two parents (Gertrud and Klaus), and a sister (Sophie). Surely do I appreciate your emotional support and eternal love, but I'm not sure how you contributed to this thesis. I'm not even sure if you know what I'm doing.



Niko Zurstraßen, Aachen 2025-10-05

Contents

1	Introduction	1
1.1	Contributions & Outline	2
1.2	Publications	3
2	Background & Related Work	5
2.1	Simulator vs. Emulator	5
2.2	Levels of Abstraction	6
2.3	Real-Time Factor	7
2.4	Instruction Set Simulator Principles	7
2.4.1	Interpreters	8
2.4.2	Binary Translators	8
2.4.3	Virtualization	12
2.4.4	Hybrid Approaches	12
3	Parallel Full-System Simulation	15
3.1	Background & Related Work	17
3.1.1	gem5's Discrete Event Simulation	17
3.1.2	Parallel Discrete Event Simulation	18
3.1.3	Timing Mode vs. Atomic Mode	21
3.2	Parallelizing gem5	22
3.2.1	Implementation of the PDES	22
3.2.2	Assignment of Objects to Event Queues	23
3.2.3	Thread-Safe Memory System	24
3.2.4	Adaptions to System Modules	25
3.2.5	Temporal Error Estimation	26
3.3	Results & Discussion	27
3.3.1	Speedup and Accuracy of Bare-Metal Bubble Sort	27
3.3.2	Speedup and Accuracy of NAS Parallel Benchmark	29
3.3.3	Other Benchmarks	30
3.4	Conclusion & Outlook	31
4	Temporal Decoupling	33
4.1	Background	35
4.1.1	Temporal Decoupling (TD)	35

4.1.2	Analytical Models	36
4.2	Related Work	37
4.3	Methods	38
4.3.1	Analytical Speedup Model	38
4.3.2	Analytical Accuracy Model	41
4.3.3	Speedup at Large Quanta	44
4.4	Results & Discussion	45
4.4.1	Sequential Temporal Decoupling	45
4.4.2	Parallel Temporal Decoupling	48
4.4.3	Qualitative Inaccuracy	51
4.5	Conclusion & Outlook	53
5	Floating Point Simulation	55
5.1	Background	57
5.1.1	IEEE 754 Floating Point Arithmetic	57
5.1.2	RISC-V Floating Point	59
5.1.3	x64 Floating Point	61
5.2	The Challenges	62
5.2.1	Canonical qNaN Encodings	62
5.2.2	Instruction Semantics	63
5.2.3	Rounding Modes	64
5.2.4	NaN Boxing	64
5.2.5	NaN Propagation	65
5.2.6	Exception Flags	65
5.2.7	Other Challenges	67
5.3	Related Work & Other Simulators	69
5.3.1	Soft Float	69
5.3.2	rv8	69
5.3.3	QEMU post-v4.0.0	70
5.3.4	Rosetta 2	72
5.3.5	You et al.	72
5.3.6	Sarrazin et al.	73
5.4	Impact of Accuracy	75
5.5	Methods	77
5.5.1	Fast Addition/Subtraction	77
5.5.2	Fast 32-bit Multiplication	78
5.5.3	Fast 32-bit Division	80
5.5.4	Fast 32-bit Square Root	81
5.5.5	Fast 32-bit Fused Multiply-Add	82
5.5.6	Fast 64-bit Operations	83
5.5.7	Fast Vector Instructions	85
5.6	Results & Discussion	88
5.6.1	Cleanroom Benchmarks	88
5.6.2	System Benchmarks	88

5.6.3	Vector Benchmarks	92
5.7	Conclusion & Outlook	94
6	Register Allocation for Dynamic Binary Translators	95
6.1	Background	96
6.1.1	RISC-V/ARM64 Registers	97
6.2	Related Work	98
6.3	Methods	99
6.3.1	Instruction-Local Register Allocation	99
6.3.2	Basic-Block-Local Register Allocation	100
6.3.3	Global Register Allocation for DBT	101
6.4	Results & Discussion	104
6.4.1	Optimal Number of Statically Mapped Registers	104
6.4.2	Benchmark Results	105
6.4.3	Result Analysis	106
6.5	Conclusion & Outlook	109
7	Summary & Future Work	111
A	Appendix	113
	Glossary	119
	List of Figures	121
	List of Tables	125
	List of Algorithms	127
	Bibliography	129

Chapter 1

Introduction

In the last 50 years, probably no technical achievement has shaped human society as much as the invention of the microprocessor. Yet not only the invention, but also the further development of the microprocessor is unprecedented in the history of technological endeavor. Already in 1965, G. Moore [138] described an observation that still holds true to this day (2025): Every two years, the integration density of transistors in microprocessors doubles. Or expressed in numbers: While the first microprocessor from 1971, the Intel 4004, comprises 2,300 transistors [97], a modern Apple M3 consists of 25 billion transistors [9]. This corresponds to a growth factor of more than $10,000,000\times$. Such an exponential growth can only be facilitated by chip manufacturing tools that advance at a similar pace. While this is often associated with physical tools, such as lithography machines, software tools are equally important for the development of microprocessors. These software tools are also referred to as Eletronic Design Automation (EDA) tools. One particular software tool is the focus of this dissertation: the so-called *Full-System Simulator (FSS)*.

The aim of a FSS is to mimic a compute system (the so-called target), including main memory, peripherals, and CPU, on a desired level of abstraction. This is achieved by modeling the target system purely as a software workload for a general-purpose CPU. Due to the modeled system existing purely virtual, a single instance of a FSS is also referred to as a Virtual Platform (VP). A completely virtual solution comes with several advantages and unique use cases. For instance, a FSS can be

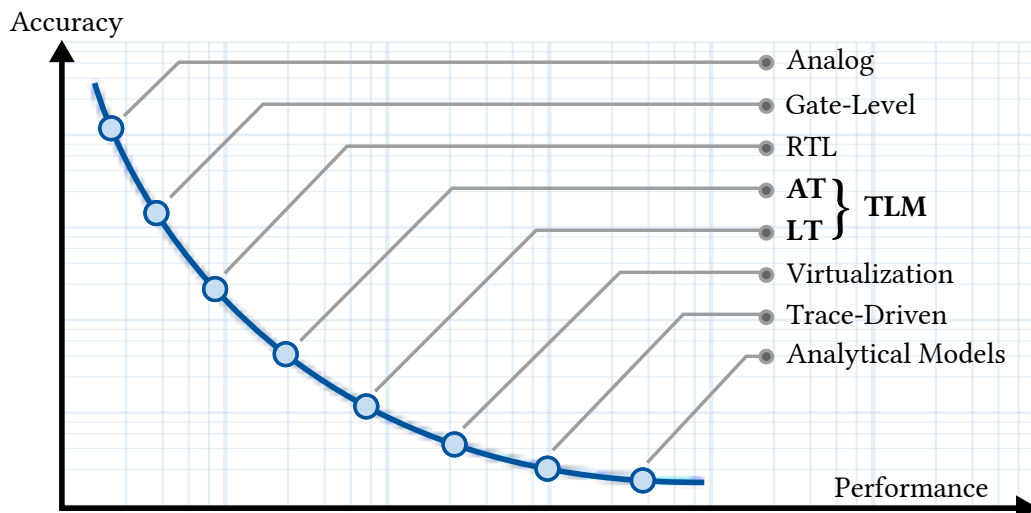


Figure 1.1: Landscape of simulation in computer engineering.

used to develop software for systems where physical prototypes are not yet available. Moreover, the parameters of a FSS can be quickly changed, which makes them an important tool for early design space exploration. Which purpose can ultimately be meaningfully pursued with a FSS depends primarily on the selected level of abstraction. As shown in Figure 1.1, a compute system can be simulated on multiple levels of abstraction, each differing in their accuracy and performance. The category of FSSs is situated in the middle of the graph, comprising the subcategories of Approximately Timed (AT) and Loosely Timed (LT). Both AT and LT aim to simulate the whole chip on a functional level, allowing the execution of software in the simulated target system. The AT style usually includes microarchitectural details making it suitable for design space exploration, while the LT style is often found in instruction-accurate simulators, which are primarily aimed at early software development. Since the details of communication among simulation objects are separated from their implementation, AT and LT are often associated with so-called Transaction Level Modeling (TLM).

Regardless of the modeling style and use case, one of the most important characteristics of a FSS is its performance. For example, if a FSS is used for design space exploration, the performance of a FSS directly determines the number of explorable designs. If a FSS is used as a tool for cross-system software development, the FSS has a direct impact on the developer's productivity. This is why most academic works and also this thesis revolve around one topic: to make FSSs faster. Much like the phases of a compiler, a FSS comprises many orthogonal challenges that can be solved to improve its performance. A summary of the challenges and contributions of this thesis is outlined in the next subsection.

1.1 Contributions & Outline

In four main chapters, this thesis presents the following contributions:

Parallel Simulation: Even though many simulators or simulation frameworks allow the modeling of complex multi-core systems, the underlying simulation kernel is often single-threaded. Consequently, the ever-growing chip complexity leads to an effective slowdown of the simulation performance. A solution to this problem is presented Chapter 3, which describes a parallel version of the popular FSS *gem5*, called *par-gem5*. By distributing target processors to individual threads on the host processor, the performance of the simulator can be increased by more than $20\times$ when simulating a 64-core system. The key idea behind the parallelization is similar to Temporal Decoupling (TD), where speedup is traded for simulation accuracy. To estimate the impact on accuracy, this chapter also presents a method for the analytical estimation of the simulation error.

Temporal Decoupling: As mentioned before, TD allows to trade speedup for simulation accuracy. This is achieved by letting individual segments of the simulation run independently for a given amount of time called *quantum*. While TD is widely

used in industry and academia, the actual relationship between speedup and accuracy is poorly understood. Especially choosing an optimal quantum is often based on empirical means. An attempt to close this gap of knowledge is described in Chapter 4. This is achieved by an analytical model to estimate performance and accuracy. Not only is the model capable of predicting these metrics, it also provides an explanation for their characteristics. The model therefore allows to find the sweet spot for optimum simulation performance with a low inaccuracy.

Floating Point Simulation: Simulating Floating Point (FP) instructions in FSSs is far from trivial. This is mainly due to differences in implementations and modeling of rounding modes and FP exception flags. While there are a few works describing fast simulation of FP for particular cases, most typical instructions remain unnoted and there is no publication regarding this topic as a whole. Remedying this circumstance, Chapter 5 first provides a complete picture of FP simulation in academia and practice. Subsequently, new methods for the simulation of scalar and vectorized FP instructions are presented. To the best of the author’s knowledge, the algorithms for simulating division, square root, and Fused Multiply-Add (FMA) have not been described anywhere else in literature. In a RISC-V-on-x64 case study, the method of this thesis outperforms the popular open-source FSS QEMU [17] by up to $1.45\times$.

Register Allocation: Similar to compilers, Dynamic Binary Translator (DBT)-based FSSs face the problem of register allocation. That means a FSS has to find an optimal mapping from the target register file to the host register file. In state-of-the-art simulators, like QEMU [17] or SIM-V [108], the prevalent register allocation algorithms are local methods, which necessarily introduce spills at the start and end of each basic block. Global methods are mostly avoided due to the associated performance penalties and dynamic nature of DBTs. However, with the proliferation of RISC computers (e.g., ARM and RISC-V) and larger register files (e.g., Intel APX [98]), existing methods need to be rethought. This thesis proposes a static global register allocation method that reserves parts of the host register file for frequently used target registers. In a RISC-V-on-ARM case study, speedups of up to $1.4\times$ could be attained in comparison to typically used basic-block-local methods.

1.2 Publications

This thesis is based on the author’s contributions that have been presented in the following publications:

1. [210] Niko Zurstraßen, Nils Bosbach, Lukas Jünger, and Rainer Leupers, “Rapid RISC-V Floating Point Vector Instructions”, in 2025 Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools (RAPIDO) (*conference proceedings*)

2. [205] Niko Zurstraßen, Ruben Brandhofer, José Cubero-Cascante, Nils Bosbach, Lukas Jünger, and Rainer Leupers, “The Art of Temporal Decoupling”, in 2025 Integration the VLSI Journal (*journal*)
3. [211] Niko Zurstraßen, Nils Bosbach, and Rainer Leupers, “FloppyFloat: An Open Source Floating Point Library for Instruction Set Simulators”, in 2025 Design, Automation and Test in Europe Conference and Exhibition (DATE) (*conference proceedings*)
4. [212] Niko Zurstraßen, Nils Bosbach, Lennart Michael Reimann, and Rainer Leupers, “Static Global Register Allocation for Dynamic Binary Translators”, in 2025 Design, Automation and Test in Europe Conference and Exhibition (DATE) (*conference proceedings*)
5. [213] Niko Zurstraßen, Ruben Brandhofer, José Cubero-Cascante, Nils Bosbach, Lukas Jünger, and Rainer Leupers, “The Optimal Quantum of Temporal Decoupling”, in 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) (*conference proceedings, best paper nominee*)
6. [206], Niko Zurstraßen, Lennart Michael Reimann, Nils Bosbach, Lukas Jünger, and Rainer Leupers, “Evaluation of the RISC-V Floating Point Extensions”, in 2023 Design and Verification Conference and Exhibition Europe (DVCon), (*conference proceedings*)
7. [209] Niko Zurstraßen, Nils Bosbach, Jan Moritz Joseph, Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers, “Efficient RISC-V-on-x64 Floating Point Simulation”, in 2023 IEEE 41st International Conference on Computer Design (ICCD) (*conference proceedings*)
8. [50] José Cubero-Cascante, Niko Zurstraßen, Jörn Nöller, Rainer Leupers, and Jan Moritz Joseph “parti-gem5: parti-gem5: gem5’s Timing Mode Parallelised”, in 2023 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) (*conference proceedings*)
9. [214] Niko Zurstraßen, José Cubero-Cascante, Jan Moritz Joseph, Li Yichao, Xie Xinghua, and Rainer Leupers, “par-gem5: Parallelizing gem5’s Atomic Mode”, in 2023 Design, Automation and Test in Europe Conference and Exhibition (DATE) (*conference proceedings*)
10. [215] Niko Zurstraßen, Lukas Jünger, Tim Kogel, Holger Keding, and Rainer Leupers, “AMAIX In-Depth: A Generic Analytical Model for Deep Learning Accelerators”, International Journal of Parallel Programming, 2022 (*journal, best paper award at SAMOS 2020*)

Chapter 2

Background & Related Work

This chapter introduces basic terms on the topic of Full-System Simulators (FSSs). Background information on more specialized topics, such as Floating Point (FP) arithmetic, can be found in the background subsections of the individual chapters.

2.1 Simulator vs. Emulator

In addition to the term “simulation” used so far, the term “emulation” is often used synonymously for the same software tools. As a semantic difference is not apparent at first glance, this subsection analyzes these terms in greater detail.

Both emulation and simulation have a similar etymological background. Emulation stems from the Latin “aemulus”, which like simulation, derived from “simulō”, can be described with the english word “imitating”. Although emulation has a stronger emotional component, which is evident from English terms such as “emulous of”, no technical differences can be derived for emulation and simulation. As there is no literature on this topic, the following section analyzes the use of language in academia, industry, and on the Internet.

Questions such as “What’s the difference between emulation and simulation?” have already been asked several times in Internet user forums such as StackOverflow. A popular answer [5] to this question classifies simulator and emulator in terms of two aspects:

Scope of modeling:

- Simulators mimic the inner and outward behavior of a system (white box).
- Emulators mimic only the outward behavior of a system (black box).

Use case:

- Emulators are used as a substitute for the target system.
- Simulators are used for research and study.

This classification fits with a large number of products and open-source projects. For example, gem5 [20, 123] labels itself as a simulator as it can model many system details, such as caches or pipeline stages, on a nearly cycle-accurate level. In addition, gem5 is primarily used for the design space exploration of compute systems. On the other hand, QEMU [17], which stands for quick emulator, only emulates on a functional level and does not model any microarchitectural details. Hence, it is primarily

used as a functional substitute for the modeled system. Interestingly, some people and projects use the exact opposite definition of simulation and emulation, or do not differentiate between the terms at all [120]. For example, tools with similar features as QEMU, such as rv8 [45] and SIM-V [108], refer to themselves as simulators. Furthermore, the meaning of simulation and emulation depends on the field of engineering. For instance, when hardware engineers talk about emulation, they usually refer to FPGA-based modelling of compute systems.

To avoid misunderstandings and to be consistent, all software tools for imitating compute systems are described as simulators in this thesis. The terms “emulator” and “emulation” are avoided whenever possible. This applies regardless of use case and scope of modeling.

2.2 Levels of Abstraction

If computer engineers are asked whether they have ever used computer simulation, most will probably respond with a “yes”. However, when going into detail, one will quickly realize that simulation is understood in different ways. In order to prevent misunderstandings, this section will explain the term “simulation” in greater detail in the context of this work.

Many, especially inexperienced users, expect computer simulation to model a system down to the smallest detail and preferably in real time. With these expectations most will quickly realize that such a simulator silver bullet does not exist. In reality, there are many different kinds of simulation that achieve sufficient accuracy at sufficient speed through abstraction. Whereby the definition of “sufficient” depends on the individual needs of the simulation user. Various types of simulation at different levels of abstraction are presented in Figure 1.1. As shown in the figure, an analog simulation can simulate transistors at the atomic level. However, the simulation of a few nanoseconds in target time can take several wall-clock hours, which is why analog simulation is highly unsuitable for use cases such as software verification.

The focus of this work is the so-called Electronic System Level (ESL) simulation. This term refers to the simulation of an entire compute system, which is accelerated by abstraction. The systems are usually described in languages such as C/C++, SystemC, or SpecC. In its most accurate but also slowest variant, ESL simulation models on a functional signal level. More common are ESL simulations that use so-called Transaction Level Modeling (TLM). With TLM, communication is no longer modeled by individual signals, but by more abstract transactions. These transactions can span multiple temporal phases, which is referred to as Approximately Timed (AT), or they are modeled as more abstract, atomic communication elements, which is referred to as Loosely Timed (LT). Since AT modeling allows the implementation of pipeline structures, it is often associated with modeling hardware on a microarchitectural level. Hence, the AT style is often used for design space exploration. The LT style models hardware on a functional level and abstracts microarchitectural details in order to achieve the best possible performance. Temporal aspects are modeled, but are often

broken down to rules such as “one instruction is one cycle”. Simulations of this type are referred to as instruction-accurate and are mainly used for software development for systems where hardware is not available.

Note that the terms TLM, AT, and LT stem from the SystemC/TLM standard [144]. Other communities use different terms for similar levels of abstraction. For instance, the gem5 [20, 123] simulator uses the terms *atomic* and *timing* instead of LT and AT, respectively. In the gaming console simulator community, similar levels of abstraction are described as Low-Level Emulation (LLE) and High-Level Emulation (HLE).

To describe instances of an ESL simulation, a number of terms have become established in recent decades. These include: Virtual Platform (VP), Virtual Prototype (VP), Virtual Twin, or Full-System Simulator (FSS). The latter term is used in this thesis as it is particularly established in the academic community. It should also be mentioned that a FSS comprises the simulation of a complete system, including all peripherals, and can be composed of other simulators. For instance, a FSS may include an Instruction Set Simulator (ISS) for the modeling of the instruction set or CPU, or a cache simulator. The former is described in more detail in Section 2.4.

2.3 Real-Time Factor

Before describing the technical details of ISSs, a means of measuring the performance of a simulator is introduced – the so-called *Real-Time Factor (RTF)*. In this thesis, the RTF is calculated by dividing the time passed in the simulation by the wall-clock time required to simulate ($RTF = T_{simulation} / T_{wallclock}$). For instance, with a RTF of 0.1, simulating 1 second of the target system would require 10 seconds of compute time. Hence, the greater the RTF, the faster the simulator. Note that some authors use a reciprocal definition of the RTF, where smaller values correspond to faster simulation.

With the current definition, the RTF may not only be a measure for the performance of a simulator, but it also incorporates implementation-specific details of the target and host. For example, if a simulated target system is clocked at 1 Mhz or at 100 Mhz, has a drastic impact on the RTF when using the same simulator for both cases. To avoid host-target dependencies, this thesis extends the RTF as follows: Whenever RTF values are mentioned without a corresponding target or host system, the values refer to a system that simulates itself. That way, dependencies on clock frequencies and system complexity can be alleviated.

2.4 Instruction Set Simulator Principles

The most important component of a FSS is the Instruction Set Simulator (ISS). As the ISS models the CPU, it is decisive for the achievable simulation performance. Regardless of the modeled CPU, ISSs can be classified into different categories based on their implementation. These different categories are explained in the following. A list of open-source FSSs including ISS classifications is given in Table 2.1.

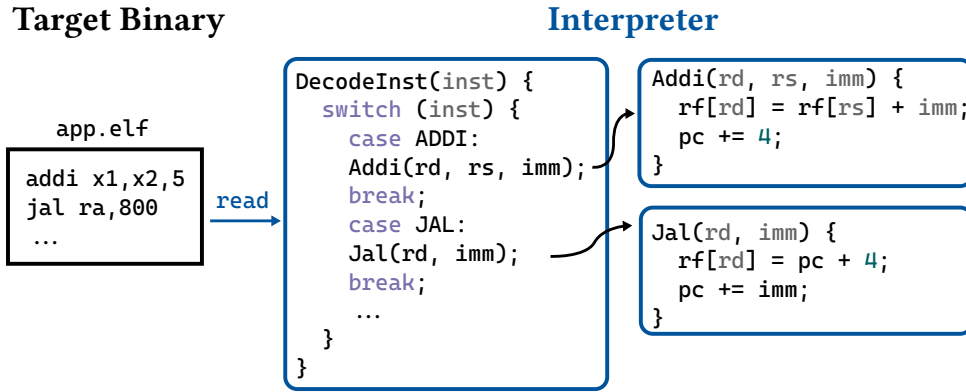


Figure 2.1: Principles of an interpreter-based ISS in C-like pseudo code.

2.4.1 Interpreters

Interpreters are the most simple and popular way to implement an ISS. In its most straightforward variation, instructions are decoded within a large switch-case statement. Every case, and thus every instruction, has a corresponding function call, which implements the functionality of the executed instruction in a high-level programming language such as C, C++, or Rust. See Figure 2.1, which depicts this principle using a RISC-V example. With interpreters handling instruction set simulation completely in the domain of a high-level programming language, they come with the advantage of being easy to maintain, extend, and port across different systems. Furthermore, as interpreters generate no code, they exhibit a relatively small memory footprint. All these advantages are contrasted by the relatively slow simulation performance. Repeatedly decoding instructions and calling service procedures has a significant negative impact on the performance. In the best case, interpreter-based ISSs can reach a RTF of up to 0.1¹ when simulating state-of-the-art systems. If the simulated system includes detailed models of the CPU or memory system, the RTF can quickly drop below 0.001. In order to increase the performance of interpreters, several techniques can be deployed. A common technique is the so-called threaded interpreter, which uses computed gotos to execute subsequent instructions. This avoids repetitive jumps into and from the decoding loop. The work of Ertl et al. [60] provides a comprehensive overview of efficient interpreter implementation. But even with these optimizations, the primary overhead of repeated service routine calls is common to all interpreters-based simulators. Examples of open-source simulators that employ interpreters include gem5 [20, 123], or1kiss [193], Spike [159], and TLMBoy [207].

2.4.2 Binary Translators

If simulation performance is of great concern, so-called Binary Translators (BTs), sometimes also termed as recompilers, are the most common choice for implementing an ISS. Hence, BT-based simulators are often instruction-accurate and focus on

¹ Value based on experiments conducted by the author where the host system models itself.

simulating functional aspects as fast as possible. The underlying concept of a BT is similar to a compiler in many regards. It translates a program from a source language to a target language, with the aim to eventually create executable machine code. While compilers usually translate from high-level languages (C, C++, etc.) to low-level languages (assembly, machine code, etc.), BTs translate machine code from one Instruction Set Architecture (ISA) (e.g., RISC-V) to another ISA (e.g., ARM64). As most ISAs are similar to large extents, translating is often a matter of mapping instructions one-to-one or one-to-few. The translation process may be conducted directly, or it may be aided by an Intermediate Representation (IR). In the case of an IR as a mediator, several languages are seen in practice: QEMU [17] brings its own IR, the work of Guo et al. [76] uses LLVM IR, and the programming language C is used in the work of Zivojnovic et al. [203]. Furthermore, the translation process can be classified as static or dynamic.

A so-called *Static Binary Translator (SBT)* translates a program pre-runtime, much like a C or C++ compiler. As machine code might be self-modifying and contain indirect branches, SBTs are challenged with many, seemingly insurmountable problems. Hence, SBTs are rather found in the embedded domain, where hardware as well as software are less complex. Academic examples include the work of Zivojnovic et al. [203], Pees et al. [151], and Zhu et al. [202]. Due to the aforementioned challenges, most general-purpose BT-based simulators employ a dynamic method.

Opposed to a SBT, a *Dynamic Binary Translator (DBT)* executes and translates a target binary in a coupled process. Note that during the beginning of DBT development, the term *dynamic compiler* was more prevalent. Also, initially stemming from the Java community [32], the term *JIT compiler* is used synonymously as well. However, both terms are also used for dynamically compiled programming languages, which is why this thesis and many more recent works prefer the less ambiguous term *DBT*. To the best of the author's knowledge, the term DBT was coined by Witchel et al. [197] in 1996.

The dynamic nature of a DBT allows to easily deal with the aforementioned challenges of self-modifying code and indirect branches. Figure 2.2 gives a simplified overview of the typical workflow of a DBT. As a first step, the target binary (e.g., an ELF file) is read by the DBT. Subsequently, the initialization process starts, which

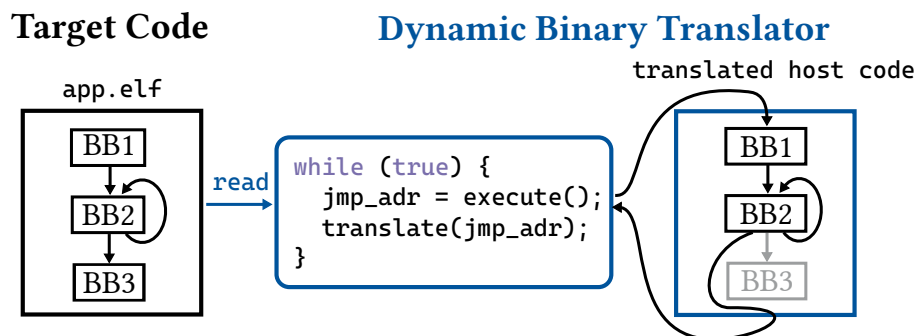


Figure 2.2: Principles of Dynamic Binary Translators (DBTs).

includes translating the first basic block. To avoid repetitive recompilations, translated basic blocks are typically inserted into a cache. Then the main loop of the DBT is entered, where code is executed and translated in an alternating fashion. During execution, the program counter may reach parts of the Control Flow Graph (CFG) that have not yet been translated. For example, BB3 in Figure 2.2 has not been translated yet. At this point, the control is handed back to the main loop, which then repeats the translation/execution cycle until the process calls for an exit. To account for self-modifying code, DBTs usually mark a host page write-protected if a translated basic block can be found within it. Thus, writes into executable code can be tracked by simply intercepting a SIGSEGV signal. If a write is detected, already translated basic blocks are invalidated. Depending on the executed workloads, DBTs attain a RTF between 0.01 and 1. That means, in a best-case scenario, a DBT is as fast as the modeled system. Theoretically, also speedups of greater 1 are possible, especially if unoptimized workloads are executed. In this case, DBTs with standard code optimizing techniques will significantly improve the machine code quality post-compile time. Apart from low-level software debugging, executing unoptimized binaries is rarely a deliberate decision but rather an accident. If a DBT does not reach native performance it may be challenged by one of the following problems:

- *Address Translation*: To quickly translate virtual addresses to physical addresses, modern compute systems rely on dedicated hardware called Memory Management Units (MMUs). Simulating these hardware units in software can be costly in terms of performance, which is why workloads with many loads and stores often see poor performance.
- *Floating Point Instructions*: As FP arithmetic changes from ISA to ISA, finding generic ways to handle the different formats and instructions is challenging. This thesis addresses the problem in Chapter 5.
- *Indirect Branches*: Indirect branches are instructions that change the control flow based on an argument register. Hence, the jump address is only available at runtime, requiring expensive cache lookups for DBT-based simulators.
- *Memory Ordering*: When simulating systems with strong memory orders (e.g., x86) on systems with weak orders (e.g., RISC-V), guaranteeing a correct order of memory transactions is challenging. A simple, but performance-diminishing solution is to add memory barriers to load and store instructions.
- *Register Allocation*: Similar to a classical compiler, also a DBT faces the problem of register allocation. Due to the dynamic nature, many DBTs use local methods leading to additional spill code. Chapter 6 highlights this problem in greater detail.
- *Status Flags*: Especially older ISAs usually provide an ample set of status flags. For instance, x86 includes a sign flag, a carry flag, an overflow flag, and many other different status flags. More recent ISAs, such as RISC-V, do not include

such flags. Hence, x86-on-RISC-V DBT-based simulation requires additional effort for the modeling of status flags.

- *Self-Modifying Code*: As mentioned before, many DBTs mark pages with code write-protected to account for self-modifying code. If code and data reside in the same page, simply writing data in these pages can lead to significant performance penalties. This problem may be further complicated if host and target system use different page sizes.
- *Translating Instructions*: If applications tend to rarely reexecute basic blocks, the translation process of the DBT can have a significant impact on the performance. Especially, if the translation process includes numerous compiler-like optimizations (constant propagation, common subexpression elimination, etc).
- *Vector/Matrix Instructions*: Many ISAs offer extensions to process vectors and matrices in a parallelized fashion. However, from fixed vector widths to strip-mining - the vector/matrix extensions between individual ISAs often exhibit so many differences, that translating instructions one-to-one is out of question. An approach addressing this problem for FP vector instructions is presented in Chapter 5.

An overview of attainable RTF values for the open-source DBT QEMU [17] is provided in Figure 2.3. The high performance of DBTs comes with several implementation-related disadvantages. For instance, writing executable code into memory and executing it is way more complicated than a simple interpreter. Furthermore, many programming languages do not even permit such a behavior which is often associated with severe security flaws. Since generating code is host-specific, every host system on which a DBT shall be executed requires its own code-emitting backend.

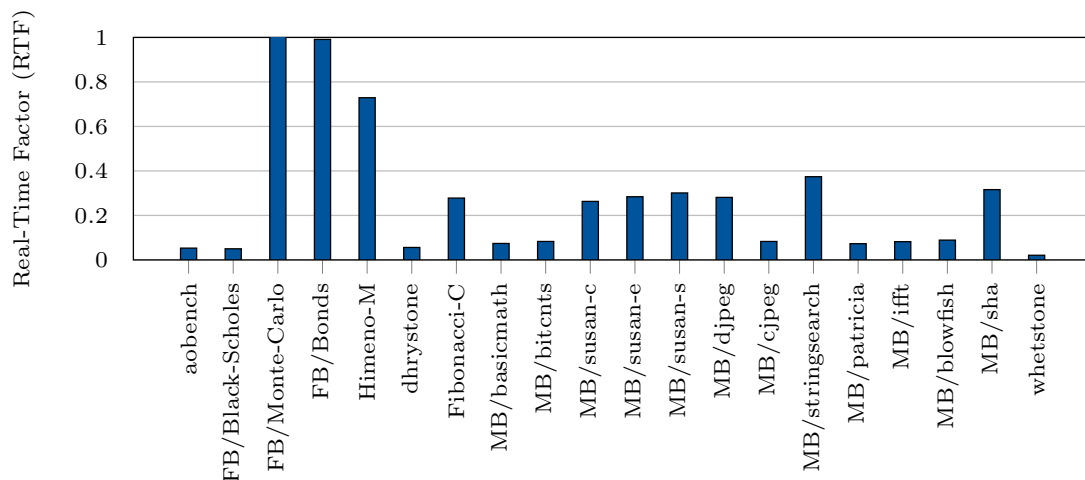


Figure 2.3: RTF values of QEMU for various benchmarks when simulating RISC-V user mode on an x86 host (AMD Threadripper 3990X). Benchmarks from Figure A.2.

This further complicates code development, portability, and maintenance. Examples of open-source DBTs include QEMU [17], Dolphin [1], and rv8 [45]. Academic works include Shade [46], Embra [197], and HQEMU [88]. Commercial DBTs are available from ARM (ARM Fast Models), Imperas (OVPsim, recently acquired by Synopsys), Intel (Simics), and MachineWare (SIM-V, SIM-A).

2.4.3 Virtualization

The key idea of virtualization is to execute code natively. Consequently, virtualization requires target and host system to implement the same ISA. To allow for efficient address translation, many modern CPUs provide hardware-assisted features. Examples of such virtualization extensions include Intel VT-x, AMD-V, or the RISC-V hypervisor extension. The open-source simulators QEMU [17] and gem5 [20, 123] can leverage these extensions given that the host system includes them. For a wide range of applications virtualization can reach near-native performance, i.e. a RTF of 1 as it faces none of the aforementioned DBT challenges. Besides host ISA and target ISA matching, virtualization-based approaches are often limited to certain exception/privilege levels. Furthermore, entering and exiting virtualization is far more expensive than entering DBT-generated code. If a simulation frequently switches between executed and the simulation environment, for instance, because of simulated peripherals, any performance gains may be offset. In a case study by J. Engblom [59] using the Intel Simics simulation, virtualization is faster than a DBT only if more than 10,000 instructions can be executed continuously on average.

2.4.4 Hybrid Approaches

All of the aforementioned approaches are not necessarily mutually exclusive, but can be combined to leverage their advantages for individual use cases. For instance, the Intel Simics simulator switches between an interpreter and DBT mode dynamically [59]. The interpreter mode is used for every first execution of a basic block, as the initial translation cost of a DBT is high. If a basic block is executed more than once, it is likely to not be part of initialization code. Hence, more executions are to be expected and the simulator switches to a DBT mode. A similar technique was also used by Transmeta's "Code Morphing Software" [115].

Also, the gem5 [20, 123] simulator is designed to dynamically switch between different modes of execution. With gem5 however, the different modes are motivated by their performance/accuracy characteristics. That means, the virtualization-based KVM mode can be used to fast-forward the system to a region of interest. From that region on, gem5 can be switched to an accurate interpreter mode, which allows to model system components like caches or CPU pipeline stages in fine detail. A similar method is also used by MARSS-x86 [150], and by HySim [116].

Table 2.1: List of popular open-source FSS. All links accessed on 2024-07-01.

Name	Target Systems	Host System	Written in	Engine	GitHub Stars
Stella	Atari 2600 VCS	multiple	C++	IP	589
simavr	AVR	multiple	C	IP	1,532
AVR8js	AVR	multiple	Javascript	IP	460
PyBoy	Game Boy	multiple	Python	IP	4,492
SameBoy	Game Boy	multiple	C	IP	1,569
MetroBoy	Game Boy	multiple	C++	IP	1,101
MoonEye GB	Game Boy	multiple	Rust	IP	885
Gearboy	Game Boy	multiple	C++	IP	823
NoobBoy	Game Boy	multiple	C++	IP	275
TLMBoy	Game Boy	multiple	C++	IP	14
mGBA	Game Boy Advance	multiple	C	IP	5,449
Visual Boy Advance - M	Game Boy Advance	multiple	C++	IP	3,252
QEMU	multiple	multiple	C	DBT/VT	9,536
gem5	multiple	multiple	C++	IP/VT	1,482
SimpleNES	NES	multiple	C++	IP	4,754
smolnes	NES	multiple	C	IP	608
puNES	NES	multiple	C/C++	IP	585
DeSmuME	Nintendo DS	x64, ARM	C++	DBT	2,814
SkyEmu	Nintendo DS	multiple	C	?	783
LimeDS	Nintendo 3DS	multiple	C++	DBT/IP	1,385
Corgi3DS	Nintendo 3DS	multiple	C++	IP	217
Project64	Nintendo 64	x64, ARM	C++	DBT	2,432
simple64	Nintendo 64	x64	C/C++	DBT	873
N64 Wasm	Nintendo 64	multiple	C/C++	IP	605
or1kiss	OpenRISC 1000	multiple	C++	IP	8
NanoBoyAdvance	Game Boy Advance	multiple	C++	IP	944
DuckStation	PlayStation	multiple	C++	DBT	6,393
Avocado	PlayStation	multiple	C++	IP	529
PCSX2	PlayStation 2	x64	C++	DBT	10,566
Play!	PlayStation 2	x64, ARM	C++	DBT	2,049
DobbieStation	PlayStation 2	x64	C/C++	DBT	522
RPCS3	PlayStation 3	multiple	C++	DBT/IP	14,599
Orbital	PlayStation 4	x64	C	VT	3,577
PPSSPP	PlayStation Portable	multiple	C++	DBT	10,727
Vita3K	PlayStation Vita	x64, ARM	C++	DBT	3,677
Spike	RISC-V	multiple	C++	IP	2,266
rvemu	RISC-V	multiple	Rust	IP	722
rv8	RISC-V	x64	C++	DBT	665
riscv-rust	RISC-V	multiple	Rust	IP	597
libriscv	RISC-V	x64	C++	SBT/IP	512
Uni Bremen RISC-V VP	RISC-V	multiple	C++	IP	138
terminus	RISC-V	multiple	Rust	IP	58
Flycast	Sega Dreamcast	multiple	C++	DBT	1,254
Yabaus	Sega Saturn	x64, ARM	C	DBT	352
Snes9x	SNES	multiple	C++	IP	2,525
Cxbx-Reloaded	Xbox	multiple	C++	IP	2,119
Xenia	Xbox 360	x64	C++	DBT	7,820
Bochs	x64	multiple	C++	IP	766
Sniper	x64	multiple	C++	IP	82
MARSS-x86	x86	multiple	C/C++	DBT/IP	122
Dolphin	Wii/GameCube	x64, ARM	C++	DBT	12,351
Cemu	Wii U	x64	C++	DBT	6,652

Chapter 3

Parallel Full-System Simulation

In the first decades of microprocessor development, the performance gain of each new generation was primarily achieved by exponentially increasing clock frequencies. However, the dream of ever-growing frequencies saw an abrupt end in the early 2000s, as increasingly smaller transistors were encountering new physical borders. Consequently, computer architects pivoted their design paradigm to increased core count and system complexity. This also complicates the design process, creating a need for sophisticated design tools to understand the impact of architectural decisions. As highlighted in Chapter 1, so-called FSSs can be used to fulfill this role. These tools allow to analyze the impact of nearly every part of the compute system on a microarchitectural level, including CPU pipeline stages, the cache system, interconnects, dynamic memories, etc. Since every model is functionally equivalent to its real-world counterpart, the FSS can simulate any kind of target software from operating systems to bare-metal applications. Consequently, a FSS allows to accurately assess Key Performance Indicators (KPIs), such as power consumption or compute performance, even before physical prototypes are available.

FSSs often face performance problems when simulating massively parallel and complex systems, as their simulation kernels are single-threaded. Prominent examples include the SystemC [2] reference implementation, and gem5 [20, 123], with the latter being among the most popular open-source FSSs for ARM-based systems. Although gem5 allows to model complex server systems with more than 100 cores, the

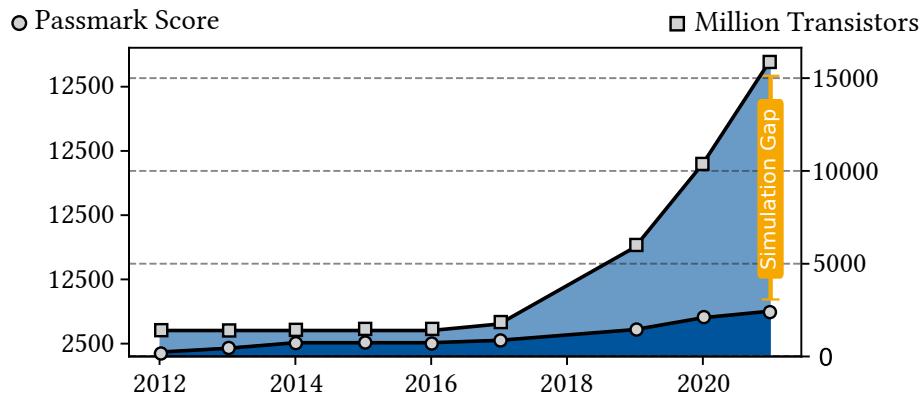


Figure 3.1: Development of transistors per processor and highest single-threaded PassMark results [149] for state-of-the-art general-purpose processors released in the given year. From 2012 (Intel Xeon E3-1290 v2) to 2021 (Apple M1), the number of transistors increased by 1042%, while single-thread performance only grew by 73%.

simulation kernel is always executed sequentially. Accumulated over all simulated cores, the simulator hardly achieves more 3 Million Instructions Per Seconds (MIPS) on most modern systems, even if the highest level of abstraction is used. When using more detailed models, the simulation speed barely exceeds 0.5 MIPS. If a 128-core server has to be simulated, the simulation performance corresponds to a RTF (simulation performance divided by real-world performance) of roughly 0.00001. With a performance on this level, the execution of representative benchmarks takes days, if not weeks. For example, the popular SPEC2017 integer benchmark suite requires approximately ten minutes of execution time on a modern host system. When executed in gem5, the same benchmark requires more than two years when modeling a 128-core system. This issue is likely to get worse, as newer CPUs only show marginal gains in sequential compute performance, while their complexity continues to grow exponentially. Figure 3.1 depicts the transistor count of high-performance general-purpose CPUs and their respective score in the sequential PassMark [149] benchmark. As shown in the Figure, from 2012 to 2021, the system complexity grew $6\times$ faster than the single-thread performance. If development continues at this pace, and if computer architects do not transition to faster methods and tools, the big freeze of simulation will be inevitable.

To tackle the crippling performance of simulation tools, multiple non-parallel solutions have been proposed to increase the simulation speed. For example, system simulation with KVM [157], trace-driven simulations like gem5’s ElasticTraces [100], or the Simpoint method [153]. Although significant speedups can be achieved with these methods, these speedups are always based on an abstraction of partial aspects of an FSS. Hence, the results can be unreliable and inaccurate. For example, the ElasticTraces framework can yield worst-case errors of up to 60% [100] for the target execution time. Therefore, these methods are no replacements for FSSs. The most promising solution to tackle the growing complexity of systems is to execute the FSS in parallel. For parallel simulations, the most crucial challenge arises from the synchronization of different parallel simulation segments. It is essential to find a solution that does not flaw the simulator’s functional correctness and also ensures high performance without too many points of synchronization.

In this chapter, a parallel implementation of the FSS gem5, named *par-gem5*, is presented. The simulator employs a modified synchronous approach in the Parallel Discrete Event Simulation (PDES) kernel with relaxed causality constraints. This approach is akin to *temporal decoupling (TD)* - a concept known from sequential SystemC simulations. With the proposed approach, the simulation time is divided into slices, in which different threads of the simulation run independently in parallel. These threads only synchronize at fixed times during the simulation. The time between two synchronizations is given by a user-provided constant called *quantum*. Compared to other approaches in the field of parallel simulation, the presented approach accepts errors in causality to increase simulation performance. In other words, speedup from parallelism is traded for simulation accuracy. As the most important finding of this chapter, the proposed approach retains the target system’s functional correctness, and the errors in simulation statistics are negligible. Specifically, this chapter contributes:

1. A TD-based PDES for gem5 that achieves a speedup of up to $24.7\times$ in bare-metal simulations and between $2\times$ to $12\times$ running Ubuntu OS and NAS Parallel Benchmarks (NPB) [143] for up to 64 simulated cores on a 64-core/-128 thread host machine.
2. An analysis showing that errors in simulation statistics, such as cache KPIs and target simulation time, are negligible from a system perspective.
3. A novel temporal error estimation that assesses the simulation time error without requiring sequential reference simulations.

Note that this chapter is based on previous works of the author [214, 50].

3.1 Background & Related Work

This section provides information on gem5's Discrete Event Simulation (DES) as well as intrinsics and related work of PDES in general.

3.1.1 gem5's Discrete Event Simulation

The core of gem5's FSS is a so-called Discrete Event Simulation (DES). Mathematically speaking, a DES can be described by a set of events scheduled at discrete points in time. Furthermore, events are connected by edges representing causality relations. That means if there is an edge from Event 0 to Event 1, Event 0 caused Event 1. In gem5, events implement the functionality of simulation objects, such as the CPU, peripherals, or the cache system. A scheduled event is initially added to the event queue, which is processed step by step and in chronological order by the simulation engine. Additionally, gem5 allows to specify a priority for events scheduled at the

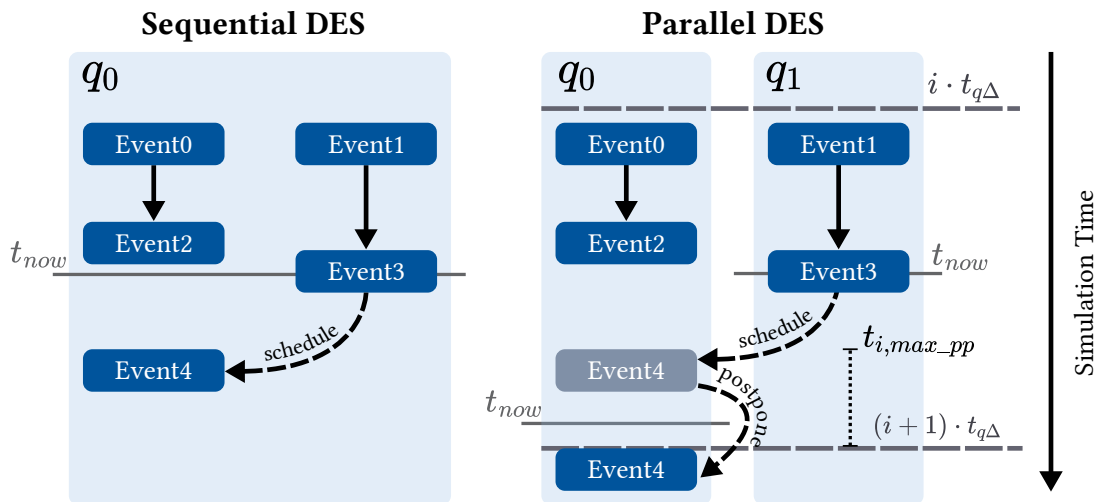


Figure 3.2: gem5 Discrete Event Simulation (DES) in sequential and parallel mode.

same time. When an event is executed, a function assigned to the event is called, fulfilling implementation-specific tasks. In most cases, events also schedule further events. For example, the *tickEvent* of gem5's atomic CPU executes the current instruction and also schedules the CPU's next cycle. The (de)scheduling of events by other events results in a dependency graph, as depicted in Figure 3.2. Excluding priority, this dependency graph G of a DES can be described by the following mathematical model:

$$G = (V, E) \quad (3.1)$$

$$V \subseteq \{(v, t, q) \mid v \in \text{Events}, t \in \mathbb{Z}^+, q \in \text{EventQ}\} \quad (3.2)$$

$$E \subseteq \{((v_0, t_0, q_0), (v_1, t_1, q_1)) \mid (v_i, t_i, q_i) \in V \wedge t_0 \leq t_1\} \quad (3.3)$$

As shown in the figure, the graph comprises a set of scheduled vertices/events V , whereby each event v has an associated timestamp t and an event queue q . In the standard sequential gem5 simulation, there is only one event queue ($|\text{EventQ}| = 1$). The edges E represent schedule or deschedule dependencies. Due to the system being causal, edges can only advance in time. The complete graph can only be determined after finishing the simulation, since events are scheduled dynamically. Eventually, when all events have been processed or when the simulation stop event is reached, the simulation terminates.

3.1.2 Parallel Discrete Event Simulation

Generally, the idea of Parallel Discrete Event Simulation (PDES) is to execute the events of a dependency graph, such as illustrated in Figure 3.2, in parallel. One of the most difficult challenges of implementing any PDES is the synchronization of the individual threads. Depending on how synchronization is achieved, simulators can be either classified as *synchronous* or *asynchronous* [168].

In an *asynchronous* PDES, each individual thread has its own local time, which is exchanged with other threads when communicating. The first fundamental work on PDES focussed on asynchronous simulation. It was independently published [71] by Chandy et al. [36] in 1979 and Bryant [27] in 1977, and is referred to as the Chandy-Misra-Bryant (CMB) Algorithm. The algorithm assumes a set of threads (logic processes), communicating packets over statically defined channels. Each thread advances asynchronously in time by always processing packets from the channel with the lowest timestamp. Assuming that the timestamps of arriving packets are non-decreasing, a parallel execution can be facilitated. However, there are situations where advancing in time is deemed unsafe for all threads, leading to a deadlock. Therefore, asynchronous simulations require a sophisticated deadlock prevention/recovery [70]. In the case of the CMB algorithm, this problem is solved by sending synchronization messages, called null messages, whenever a logic process blocks. Practical works and examples that employ the CMB algorithm can be found in the field of digital circuit simulation in [112, 15] and SystemC TLM-2.0 simulation [184, 130, 126, 189].

Opposed to asynchronous PDES, *synchronous* PDES uses global synchronization events to advance the simulation in a lockstep fashion. The simplest form is a synchronization on delta cycle level. That means all events scheduled for the same simulation timestamp are executed in parallel. Within a delta cycle, there is no partial order for the given events, allowing the simulator to exploit parallelism. However, the restriction to delta cycles also limits potential performance gains [194]. Nevertheless, this approach is used in several works on parallel SystemC simulation [41, 168, 43].

To achieve greater speedups, distance-based approaches allow threads to advance in time without synchronization for a time $t_{q\Delta}$. While this concept significantly increases the exploitable parallelism, it introduces the risk of causality errors as there is no guarantee of a chronological execution order for the events. To prevent causality errors from happening, PDES implementations have to resort to *conservative* and *optimistic* methods [16]. Optimistic methods were first presented in a work of D. Jefferson in 1983 [101]. With this approach, causality errors are not prevented in the first place but detected and rectified. The rectification can be implemented by rolling back the system to a known correct state, and reexecuting the error-causing segment with tighter synchronization. Yet, resetting the system to a correct state requires constant checkpointing and occasional reloading, which may come with a significant performance penalty. Examples of the optimistic approach in FSSs can be found in the works of Jung et al. [105] and Busnot et al. [29].

Opposed to optimistic approaches, *conservative* methods prevent causality errors from occurring. This is achieved by exploiting design-specific timing information, which is either derived from the simulation models or set manually. With this approach, multiple simulation threads synchronize every $t_{q\Delta}$, whereby a lookahead of t_{la} is provided for each thread. If $t_{la} \geq t_{q\Delta}$, causality can be retained

Although gem5 is one of the most popular FSSs, almost all existent publications on parallel FSSs refer to the parallelization of SystemC or other frameworks. See Table 3.1, which provides a comprehensive list of all works on parallel FSSs since 2004. As shown in the table, the only parallel extension of gem5 to date (September 2024) is *dist-gem5* [136], a work resulting from the consolidation of *pd-gem5* [4] and *multi-gem5*. It can be classified as a conservative, synchronous PDES, and is limited to the simulation of distributed systems connected via a Network Interface Controller (NIC). With *dist-gem5*, a simulated compute node runs as an individual gem5 process, and synchronizes with the other nodes at simulation time intervals of $t_{q\Delta}$, also called *quantum*. Causality errors are prevented by setting $t_{q\Delta}$ to the simulated latency of the NIC t_{NIC} . This approach retains causality without a sophisticated algorithm, but limits the applicability of *dist-gem5* to distributed systems connected via NICs.

According to the gem5 developers, parallelizing gem5 is one of the community's future goals. But more than 10 years after the publication of this statement, a parallel version of gem5 did not progress beyond a mode primarily intended to support a parallel Kernel-based Virtual Machine (KVM) simulation mode. To the best of the author's knowledge, a generalized parallel gem5 does not exist. Based on this fundament, the following subsections introduce *par-gem5*; a parallelized version of gem5 for the atomic mode.

Table 3.1: Published work on parallel FSSs since 2004.

Author	Year	FOSS	Based on	Driver	Synch.	Causality	Modeling	Name
Bosbach et al. [23, 22]	2023	✗	SC	CPU	synch.	TD	LT	sc-async
Cubero et al. [50]	2023	✗	gem5	CPU	synch.	TD	AT	parti-gem5
Zurstraßen et al. [214]	2023	✗	gem5	CPU	synch.	TD	LT	par-gem5
Galicía et al. [74]	2022	✗	SC	CPU	asynch.	conservative	-	-
Jünger et al. [109]	2022	✗	SC	CPU	asynch.	conservative	LT	X-on-X
Dömer et al. [55]	2021	✓	SC	CPU	custom	conservative	-	RISC
Cheng et al. [39]	2020	✗	SC	CPU	custom	conservative	-	-
Busnot et al. [29, 30]	2020	✗	SC	CPU	synch.	optimistic	LT	-
Cheng et al. [38]	2019	✗	SC	CPU	custom	conservative	-	-
Brokalakis et al. [26, 176]	2018	✓	gem5	CPU	synch.	conservative	-	COSSIM
Schmidt et al. [167]	2017	✗	SC	CPU	custom	conservative	-	-
Mohammad et al. [136]	2017	✗	gem5	CPU	synch.	conservative	-	dist-gem5
Ventroux et al. [183]	2016	✗	SC	CPU	synch.	TD	-	SCale
Virten et al. [186]	2016	✗	SC	CPU	untimed	-	untimed	IPtLM
Weinstock et al. [194]	2016	✗	SC	CPU	asynch.	conservative	LT	SC-Link
Weinstock et al. [195]	2016	✗	SC	CPU	asynch.	conservative	LT	SCope
Khan et al. [114]	2015	✗	SC/Simics	CPU	asynch.	conservative	-	-
Alian et al. [4]	2015	✗	gem5	CPU	synch.	conservative	-	pd-gem5
Chung et al. [43]	2014	✗	SC	CPU	Δ -synch.	conservative	-	SimParallel
Haetzer et al. [80]	2014	✗	SC	CPU	asynch.	conservative	LT	-
Sauer et al. [166]	2014	✗	SC	CPU	asynch.	TD	LT	CoMix
Ventroux et al. [182]	2014	✗	SC	ASIP	Δ -synch.	conservative	-	RAVES
Weinstock et al. [196]	2014	✗	SC	CPU	asynch.	conservative	LT	SCope
Wang et al. [189]	2014	✗	custom	CPU	(a)synch.	con./TD	AT	Manifold
Moy et al. [140]	2013	✓	SC	CPU	custom	conservative	LT	sc-during
Schumacher et al. [169]	2013	✗	SC	CPU	-	-	-	legaSCi
Chen et al. [37]	2012	✗	SC	CPU	custom	conservative	-	-
Dömer et al. [54]	2012	✗	SpecC	CPU	Δ -synch.	conservative	-	-
Roth et al. [162]	2012	✗	SC	SCC	Δ -synch.	conservative	-	-
Sinha et al. [172]	2012	✗	SC	GPU	Δ -synch.	conservative	LT/AT	-
Vinco et al. [185]	2012	✗	SC	GPU	Δ -synch.	conservative	-	SAGA
Certner et al. [34]	2011	✗	custom	CPU	asynch.	TD	-	SiMany
Haetzer et al. [79]	2011	✗	SC	CPU	asynch.	conservative	LT	-
Liao et al. [121]	2011	✗	SC	CPU	synch.	conservative	-	-
Peeters et al. [152]	2011	✗	SC	CPU	asynch.	TD	-	-
Ding et al. [52]	2011	✓	QEMU	CPU	untimed	-	-	PQEMU
Mello et al. [130]	2010	✗	SC	CPU	asynch.	TD	TLM-DT	SystemC-SM
Nanjundappa al. [142]	2010	✗	SC	GPU	synch.	conservative	-	SCGPSim
Schumacher et al. [168]	2010	✗	SC	CPU	synch.	conservative	-	parSC
Miller et al. [134]	2010	✓	Pin	CPU	multiple	multiple	-	Graphite
Sirowy et al. [173]	2009	✗	SC	ASIP	synch.	conservative	-	-
Khaligh et al. [164, 113]	2009	✗	SC	CPU	synch.	TD	-	-
Ziyu et al. [204, 81]	2009	✗	SC	CPU	synch.	conservative	-	ArchSC
Ezudheen et al. [62]	2009	✗	SC	CPU	Δ -synch.	conservative	-	-
Kaouane et al. [111]	2008	✗	SC	Cell	synch.	conservative	-	SysCell
Huang et al. [92]	2008	✗	SC	CPU	Δ -synch.	conservative	-	SCD
Combes et al. [47]	2008	✗	SC	CPU	Δ -synch.	conservative	-	-
Viaud et al. [184]	2006	✗	SC	CPU	asynch.	conservative	-	-
Chopard et al. [42]	2006	✗	SC	CPU	Δ -synch	conservative	-	-
Cox et al. [49]	2005	✗	SC	CPU	synch.	conservative	-	RITSim
Meftali et al. [129]	2004	✗	SC	CPU	synch.	conservative	-	-
Zheng et al. [200]	2004	✗	custom	CPU	custom	optimistic	-	BigSim

3.1.3 Timing Mode vs. Atomic Mode

To trade off accuracy and simulation performance, gem5 offers different levels of abstraction for its protocols and models. In general, the so-called *timing mode* and *atomic mode* are distinguished. As explained in Section 2.2, similar levels of abstraction are also available in SystemC, where the modes are referred to as *Approximately Timed (AT)* and *Loosely Timed (LT)* [2], respectively.

The timing mode allows to accurately model complex communication channels, where exchanging data between two simulation objects comprises multiple phases. As can be seen in Figure 3.3, Case ② and ③, there is a request phase and response phase in gem5. Between requesting and receiving data, a simulation time of t_Δ passes, and simulation objects can also request/respond by returning false instead of true. A rejection is depicted in Figure 3.3, Case ③, and leads to a recovery mechanism that involves multiple further phases. Ultimately, the relatively low level of abstraction allows to model any kind of hardware, including in-order/out-of-order CPUs, multi-level cache systems with outstanding transactions, and arbitrarily shaped Network On Chips (NOCs). However, this level of detail comes at the price of relatively low simulation performance. In most scenarios, the timing mode attains a performance of less than 0.1 MIPS on modern host systems, which is roughly $10\times$ slower than the atomic mode.

In contrast to the timing mode, the atomic mode models a communication transaction in a single phase. An example of atomic mode transactions is illustrated in Figure 3.3, Case ①. Note that a latency t_Δ is returned by the function call. This latency is not part of the request/receive process, and it is up to the requester to annotate it

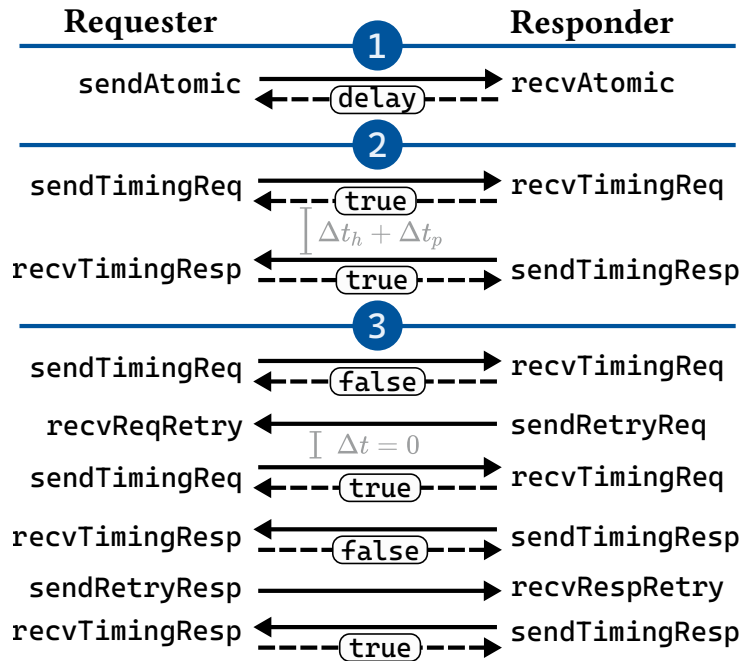


Figure 3.3: gem5 port protocols.

correctly in its following processing. With only one phase, the atomic mode cannot represent dynamic effects like outstanding transactions or bus contention. This also limits the complexity of the simulation models. For instance, in the atomic mode, gem5 only offers a single-cycle CPU model. The decreased complexity of simulation models and communication phases is beneficial for the simulation performance and code complexity of the simulator.

3.2 Parallelizing gem5

In the following subsections, the key idea of par-gem5 and its implementation are presented. Note that this chapter primarily focuses on gem5's atomic mode.

3.2.1 Implementation of the PDES

As described in Section 3.1, there are numerous different ways to implement a PDES, each with individual advantages and drawbacks. Which method works best, ultimately depends on the individual use case. For instance, asynchronous algorithms work best if there is frequent communication among simulation models and if there is a fixed channel latency. Both conditions are not met for gem5, where CPUs may run without communicating for longer periods of time, and where CPUs might issue interrupts to other CPUs instantaneously. Hence, synchronous methods may be more suitable. Here, one can choose between optimistic or conservative approaches. Optimistic approaches may be feasible for gem5, but snapshotting and reloading the whole gem5 process is likely to incur a high performance penalty. Since simulation objects can be connected via zero-latency communication channels, the only applicable conservative approach would be a delta cycle simulation. However, the frequent synchronizations would likely impede simulation performance.

As no established PDES method seems to work well with the given situation, par-gem5 uses a method that allows causality errors. The principle of this method is inspired by SystemC's Temporal Decoupling (TD), and partitions the simulation into threads, which run independently for a simulation time $t_{q\Delta}$, called *quantum*. Each thread is synchronized globally every $t_{q\Delta}$ seconds, regardless of any communication delays. Hence, the implementation can be classified as a synchronous simulation with causality errors. However, with sophisticated event queue partitioning, and a well-chosen $t_{q\Delta}$, the occurrence of causality errors can be reduced to a minimum. With the chosen approach, causality errors can occur whenever an event (de)schedules another event outside the event queue within a quantum (see Figure 3.2, "Parallel DES"):

$$((v_0, t_0, q_0), (v_1, t_1, q_1)) \in E \wedge q_0 \neq q_1 \quad (3.4)$$

$$i \cdot t_{q\Delta} < t_0, t_1 < (i+1) \cdot t_{q\Delta} \quad \text{with } i \in \mathbb{N}_0 \quad (3.5)$$

For example, it is possible that event queue q_0 is at the beginning of the quantum $(i \cdot t_{q\Delta})$, while event queue q_1 is already waiting for synchronization at the end of the quantum $((i+1) \cdot t_{q\Delta})$. If an event in q_1 would be scheduled with $t_1 < (i+1) \cdot t_{q\Delta}$,

this event would not be executed as the simulation time had already exceeded point t_1 . Such a scenario is depicted in Figure 3.2 where Event 3 initially attempts to cross-schedule Event 4 into the event queue q_0 within the same quantum.

One of the most important findings of this thesis is that in the context of general-purpose FSSs the “if” has a greater impact on the simulation than the “when” for most cross-scheduled events. For example, a timer interrupt being issued a few microseconds too late affects the simulation accuracy only slightly, while a non-issued timer may bring the system into an unstable state. Consequently, a postpone mechanism was implemented that reschedules causality-breaking events to the next quantum border, (see Figure 3.2, “Parallel DES”).

Even though this mechanism deeply intervenes in the simulation mechanics, it retains the system’s functionality for most cases. That means most benchmarks and a Linux boot sequence can be executed without any problems.

3.2.2 Assignment of Objects to Event Queues

With gem5, individual simulated hardware modules are referred to as simulation objects. Examples of these simulation objects include models for CPU cores, memory controllers, or UART devices. Full systems are constructed by hierarchically instantiating objects to mirror the structure of the target hardware platform. Since simulation objects are always assigned to an event queue, minimizing causality errors is one of the key challenges. Empirical observations suggest that leveraging the inherent concurrency of multi-core architectures produces the best results. That means simulation objects are mapped to event queues according to the following rules:

1. Each CPU subsystem with its L1 cache has a dedicated event queue.
2. All other simulation objects (main memory, peripherals, etc.) are assigned to the default event queue q_0 .

The event queue assignment takes place during the initialization phase of the simulation where one thread is created for each event queue. Thus, the total number of simulation threads is the number of CPU cores plus one. A simplified example of a 2-core platform with its corresponding event queue assignment is depicted in Figure 3.4. In

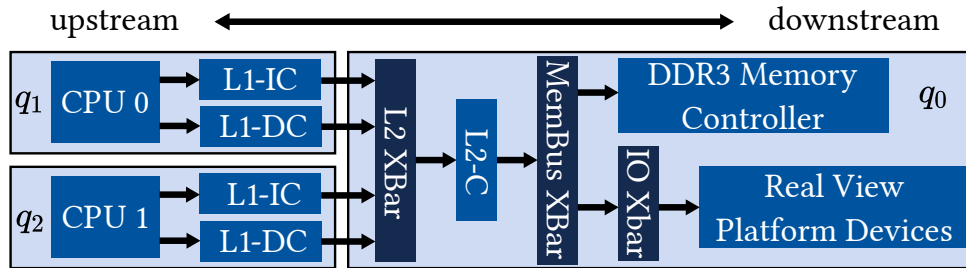


Figure 3.4: Assignment of event queues (q_i) in atomic mode. Transactions travelling from the CPUs to the memory system are referred to as travelling downstream, while the other direction is referred to as upstream.

the figure, the object *Real View Platform Devices* includes all peripheral devices, such as system timers, the generic interrupt controller, real-time clock, and IO interfaces. Following the aforementioned rules, all objects of the CPU subsystems (core, private L1 caches, and TLBs) are assigned to the same queue (q_0 and q_1). This assignment allows each CPU core to advance in the simulation using a local time within the quantum borders. With the L1 caches being included in the CPU subsystem, most data accesses can be handled by a single simulation thread. All other accesses to modules such as L2 caches, DDR memory, and peripherals, cross simulation threads.

3.2.3 Thread-Safe Memory System

The theoretical aspects of PDES have been the primary focus of this work thus far. However, considerable effort on *par-gem5* had been dedicated to addressing the practical problem of thread safety. The software stack of *gem5* was designed for sequential execution, creating the risk of race conditions when function calls or data accesses occur between objects assigned to different event queues. In the chosen assignment of event queues, this issue particularly impacts communication from L1 caches to the crossbar (see Figure 3.4). To better understand the intricacies of the L1-to-crossbar communication, a flow analysis of transactions was conducted, with special attention to transactions moving between different event queue domains. Based on this analysis, it was decided to extend the *gem5* port class with a new thread-safe locking functionality. Over the course of this work, a complete rework of the memory system

```

1 BaseCache::recvAtomic(...) {
2   port.lock();
3   if (xBarNeeded()) {
4     port.unlock(); // Prevent deadlocks from snoops.
5     port.lock_peer(); // Take ownership of XBar.
6   }
7   ... // Untouched gem5 cache code.
8   if (xBarNeeded()) {
9     port.unlock_peer();
10  } else {
11    port.unlock();
12  }
13 }
14
15 CoherentXBar::forwardAtomic(...) {
16   ...
17   p.lock_peer(); // Locks snoop target caches.
18   Tick latency = p->sendAtomicSnoop(pkt);
19   p.unlock_peer();
20   ...
21 }
```

Code 3.1: Simplified C++ code of the thread-safe cache and crossbar model.

was also considered. However, with a size of 50 kSLOC, completely rebuilding the memory system would require several person-years. Hence, this work focussed on a lightweight solution that builds on a modified port class.

Ports in gem5 are similar to ports in SystemC and connect initiators to peers / targets, allowing them to communicate. In par-gem5, ports now include the functions `lock()`, `unlock()` and `unlock_peer()`, `lock_peer()`. These functions can be used to lock/unlock their own or the peer's recursive mutex, respectively.

As depicted in Code 3.1, the new port class sequentializes accesses to the shared crossbar. Before a cache tries to satisfy an upstream request, it has to contend its port mutex (Line 2). This avoids incoming snoop requests altering the data of the upstream request's address. Once the mutex is held, it is determined whether the request requires access to the downstream crossbar (Line 3). Access is required for any transaction that cannot be served by the local cache and thus requires communication to other caches or memories. For example, this includes read misses, write misses, cache maintenance operations, or load-linked/store-conditionals. If access to the crossbar is required, the cache mutex is released to prevent deadlocks from incoming snoops, and the crossbar mutex is contended using the `port.lock_peer()` function (Line 4 and Line 5). Next in Line 7, unmodified gem5 cache mechanisms are executed, which may include calls to the crossbar. As the crossbar communicates to other caches via snoop requests, an additional lock in the crossbar code (Line 17) ensures that snoop requests do not lead to race conditions. However, cache snoops can lead to writebacks that need access to the crossbar in return. To avoid deadlocking by multiple crossbar accesses, the ports use recursive mutexes, allowing multiple accesses as long as they are performed by the same thread. Note that despite being in a separate event queue, the transportation functions of the crossbar (e.g., `CoherentXBar::forwardAtomic`) are performed in the initiators' (i.e. the caches) threads. Finally, the transaction concludes with the release of any held mutex (Line 9 and Line 11).

3.2.4 Adaptions to System Modules

Besides the aforementioned gem5 simulation objects, such as ports and crossbars, the code of other simulation objects had to be modified as well to account for the parallel execution. For example, the ARM timer needed changes, as the current gem5

```

1 void SystemCounter::updateValue() {
2     uint64_t new_value = _value;
3     // Update only if time progressed.
4     if (curTick() >= _updateTick) {
5         new_value = _value + ((curTick() - _updateTick)
6                               / _period) * _increment;
7     }
8     // ...
9 }

```

Code 3.2: Modified code from gem5 arm/generic_timer.cc.

model could not handle accesses from different time domains. On ARM systems, the current virtual system time (physical time minus virtual offset) is derived from the `CNTVCT_ELO` register. Whenever a CPU in the simulation accesses this register, `gem5` determines its value by calculating a timing delta from the current and the last access. However, with `par-gem5` and multiple time domains, the timestamp of the last access can be greater than the current access. That leads to negative deltas, which are ultimately cast to unsigned integers, leading to undefined behavior. This problem can be solved by allowing only greater or equal than zero updates (see Code 3.2, Line 4). Although this slightly changes the semantics of the timer, no problems in practice could be observed. In a similar way, the negative delta problem also occurs with several other modules like statistics objects. A more elaborated description of the encountered problems is given in Chapter 4, Subsection 4.4.3.

3.2.5 Temporal Error Estimation

One of the main challenges of the proposed approach is determining a quantum that balances accuracy and speedup effectively. For maximizing speedup, a larger quantum is preferable as it reduces the number of synchronization points and improves load balancing. Conversely, for higher accuracy, a smaller quantum is needed due to the lower likelihood of temporal errors. Additionally, determining the impact of the quantum in the first place is not straightforward. While speedup can be roughly estimated from the required wall-clock time, assessing the impact on accuracy necessitates a sequential simulation as a reference, which would negate the speedup benefits of `par-gem5`.

To address this, `par-gem5` has been enhanced with a temporal error estimation feature. At the end of each parallel simulation, an analytical worst-case estimation of the temporal error is provided as a simulation statistic. This estimation allows users to evaluate the accuracy of the simulation without needing a reference simulation.

The concept behind this error estimation is to accumulate timing errors of postponed events. For this purpose, `par-gem5` tracks for each quantum i , which postponed event had the largest shift in time t_{i,max_pp} caused by the postpone-mechanism. It is assumed that the postponed event was on the program's critical path, extending the simulation time by t_{i,max_pp} . By accumulating the prolongation times t_{i,max_pp} over all quanta, the following worst-case temporal error estimation can be derived:

$$e_{rel,t} = \frac{t_{sim,meas}}{t_{sim,meas} - \sum_{i=1}^Q t_{i,max_pp}} - 1 = \frac{t_{sim,meas}}{t_{sim,est}} - 1 \quad (3.6)$$

With $t_{sim,meas}$ being the measured simulation time and Q being the number of simulated quanta. The results of the temporal error estimation are provided in the next section. An analytical approach to assess accuracy and performance is presented in Chapter 4 of this thesis.

3.3 Results & Discussion

To assess the accuracy and performance of par-gem5, a representative set of parallel benchmarks was executed. The configurations of the target and host system are given in Table 3.2. Simulation objects were mapped to event queues according to the strategy explained in Section 3.2.2. Thus, every CPU subsystem is mapped to a thread on the host system.

Table 3.2: Configuration of the host and target system.

	host system	target system
ISA	x64	ARM64
CPU	AMD Ryzen Threadripper 3990x	AtomicSimpleCPU
clock speed	2.9 GHz-4.3 GHz	2 GHz
# cores	64 physical, 128 logical	{1, 2, 4, 8, 16, 32, 64, 128}
L1 cache	2 MiB L1-D, 2 MiB L1-I,	64 kiB L1-D, 32 kiB L1-I
L2 cache	32 MiB L2	2 MiB L2 shared
L3 cache	256 MiB L3	-
main memory	DDR4 RAM 128 GiB @ 3200 MHz	DDR3 RAM @ 1600 MHz
periph. sub-system	-	Real View Virtual Express V1
OS	Ubuntu 20.04	{Bare-metal, Ubuntu 14.04}

3.3.1 Speedup and Accuracy of Bare-Metal Bubble Sort

For an initial examination of the parallel simulation approach, a simple multi-core bare-metal bubble sort benchmark was implemented. This benchmark comprises N threads, with the number of available CPU cores being N , where each thread independently sorts its own array of integers. It is designed to achieve near-best-case simulation throughput by minimizing thread synchronizations and accesses to shared memory. Moreover, the benchmark runs bare-metal, i.e., without any Operating System (OS), which reduces any possible thread synchronizations even further. The simple program was run multiple times, varying the number of simulated cores and the simulation quantum. Figure 3.5 shows the throughput in MIPS achieved in this experiment. The accumulated throughput increases with the number of simulated cores in parallel mode but remains almost constant in regular single-threaded simulation. A maximum throughput of 22.1 MIPS is reached with the 128-core configuration, representing a speedup factor of $24.7\times$. The achievable speedup grows with the quantum size and levels off between $10\mu\text{s}$ and $1000\mu\text{s}$. This observation aligns with findings from other studies [107, 136]. A detailed explanation for the diminishing performance is later provided in Chapter 4.

Due to potential causality errors in par-gem5, the simulation's accuracy must be assessed as well. In Figure 3.6, a comparison of the simulated time for the individual settings is presented. The inaccuracy increases with both the quantum and the

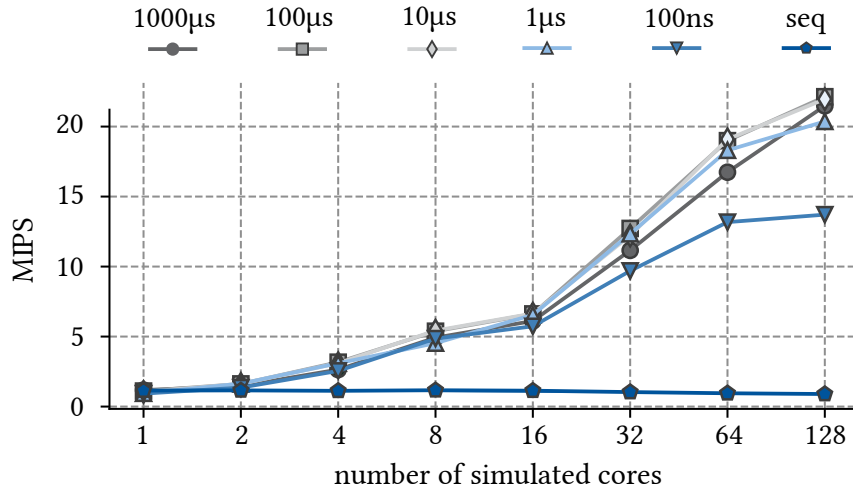


Figure 3.5: Number of simulated instructions for bubble sort using various quanta.

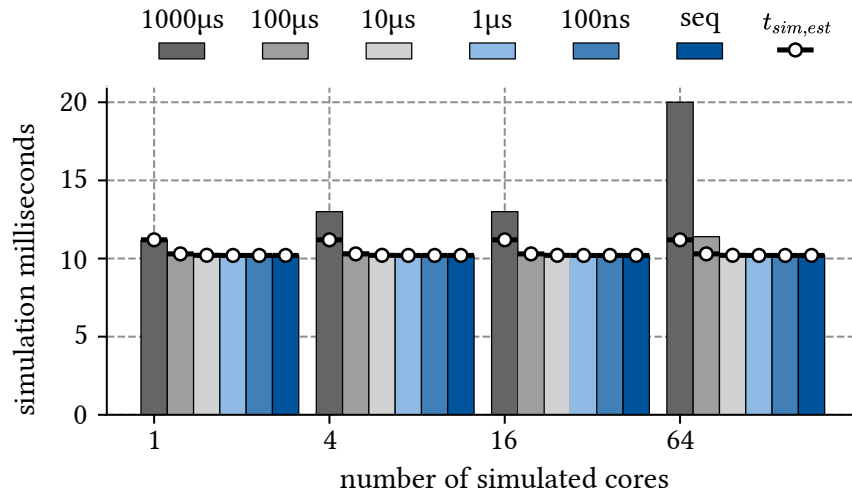


Figure 3.6: Accuracy of simulated time for bubble sort with various quanta.

number of cores and reaches noticeable levels at a quantum of $1000\mu s$ when simulating a 2-core system. In the worst case (64 cores at $1000\mu s$) the timing accuracy is off by 100%. Also, if there is any significant inaccuracy, the simulation time is always greater than the sequential reference simulation. This observation can be explained by the postponing of CPU wake-up events, which are used to synchronize the threads of the benchmark. More specifically, the ARM SEV¹ instruction is implemented by cross-scheduling a global wake-up event. This event is issued immediately to all cores in regular mode, but the postpone mechanism of par-gem5 shifts it to the next quantum border when running in parallel.

Besides measured accuracy, Figure 3.6 also presents the results of the temporal error estimation. A significant discrepancy between the measured value and the estimated value $t_{sim,est}$ indicates greater inaccuracy in the simulation. Any substantial inaccuracy is accurately detected by the temporal error estimation, as can be seen with multiple examples, such as (64 cores, $1000\mu s$) or (4 cores, $1000\mu s$).

¹ The ARM SEV (Set Event) instruction signals an event to all cores within a multi-core system.

3.3.2 Speedup and Accuracy of NAS Parallel Benchmark

To evaluate par-gem5 with industry-standard multi-core benchmarks, NPB [143] was executed on a target with Ubuntu Linux 14.04. Specifically, the original eight benchmarks specified in NPB 1 (IS, EP, CG, MG, FT, BT, SP, LU) were run. The OpenMP implementation from NPB version 3.4.2 and the Wide (W) data size configuration were utilized. Although the different test programs of the NPB are all designed for multi-core platforms, they do not scale homogeneously due to significant variations in memory access patterns, types of operations, and frequency of synchronization points. A detailed analysis of each benchmark is presented in [171].

For each benchmark, the same experiment as described in the previous section was conducted, incrementing the number of simulated cores. In this case, the quantum was fixed at $1\ \mu\text{s}$ due to the length of the simulations. Based on empirical analysis, a quantum of $1\ \mu\text{s}$ provides nearly maximum speedup while maintaining a low inaccuracy. An elaborated explanation for the quantum choice is provided in Chapter 4. With the default single-thread mode, a throughput between 0.57 and 0.96 MIPS was recorded, while in parallel mode, this value reached up to 7.37 MIPS. The speedup factor for each test was computed and the results are presented in Figure 3.7.

The highest speedup of $12.1\times$ is observed for the EP benchmark in the 64-core configuration. In addition to possible constraints from the host platform hardware, the potential speedup of the parallelization approach is also affected by the simulated application. Software workloads with a high parallelization potential and a low core-to-core communication achieve the best throughput in our simulator. This is the case of EP and BT, which show significant speedups when running both natively [171] and on par-gem5. Nevertheless, it can be concluded that all benchmarks in all configurations benefited from parallelization, consistently achieving speedups greater than 1. Each NPB test program prints a report, including several values, such as the total execution time and the measured performance of the system. It can be regarded as a macro statistic that sums up all performance-influencing micro parameters, like

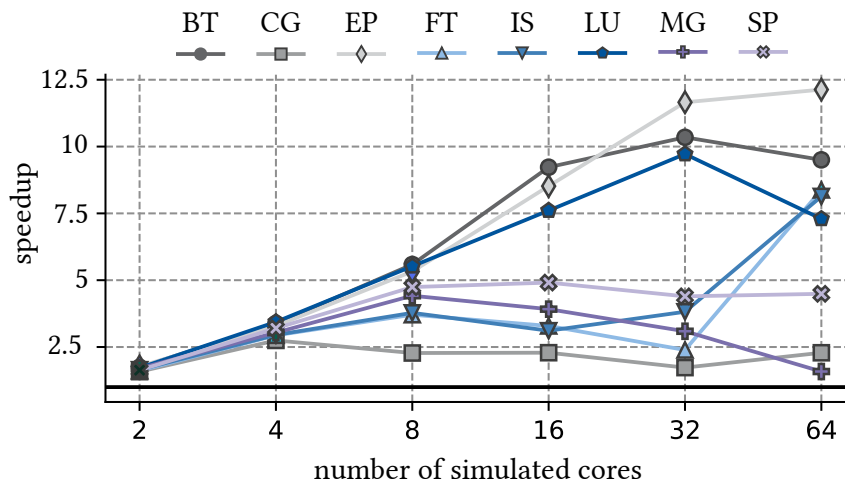


Figure 3.7: Speedup for individual NPB benchmarks.

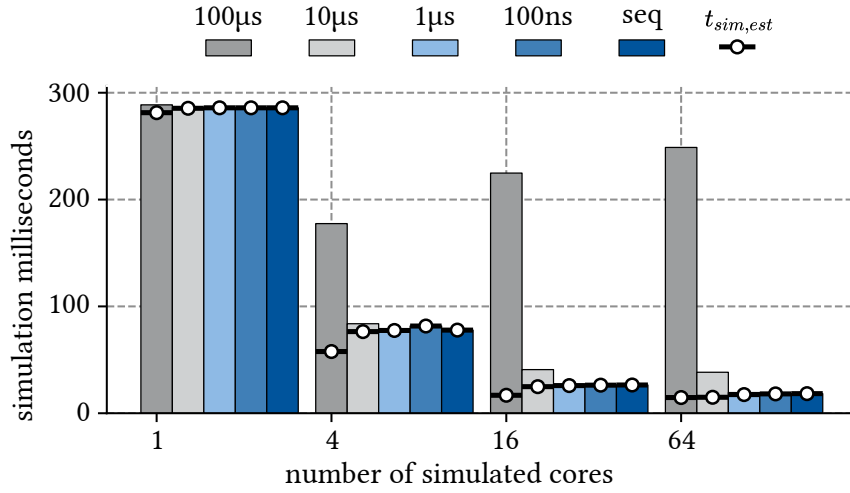


Figure 3.8: Accuracy of simulated time when executing the NPB IS with various quantum sizes.

cache hit rate or memory bandwidth. The deviation of the benchmark report between sequential and parallel simulation is always below 1.53%.

Besides the aforementioned macro statistics, other statistics, e.g., on the performance of the storage system, may also be of interest. Table 3.3 compares the L1 cache miss rates for regular gem5 and par-gem5 when simulating a 4-core system. Here, the inaccuracy of the parallel simulation never exceeds 5.7%. Also other parameters, such as the main memory bandwidth or L2-cache hit rate, exhibit accuracies of more than 99%. The accuracy of the temporal error estimation is exemplarily shown for the IS benchmark in Figure 3.8. Here, larger inaccuracies could reliably be detected (see 4 and 16 cores for 100 μ s quantum) as well. In addition to pure performance measurement, the NPB suite checks the results for correctness. There were no irregularities in any of the benchmarks, which underpins the functional correctness of par-gem5.

Table 3.3: 4-core 1 μ s quantum simulation of NPB IS.

	dcache miss rate			icache miss rate		
CPU	1 μ s	reg	error	1 μ s	reg	error
0	0.040828	0.042419	3.8%	0.000506	0.000497	1.7%
1	0.040065	0.040325	0.6%	0.001457	0.001445	0.8%
2	0.040093	0.042527	5.7%	0.000288	0.000278	3.6%
3	0.040960	0.040621	0.8%	0.000460	0.000446	3.1%

3.3.3 Other Benchmarks

In addition to the aforementioned benchmarks, experiments with other benchmarks like PARSEC, SPEC2017, and STREAM were conducted. Again, a similar picture emerges: arithmetic-intensive applications achieve high speedups (18 $\times$ for PARSEC blackscholes at 128 cores), while benchmarks with a high load on shared resources, such as STREAM, show low to no speedups.

3.4 Conclusion & Outlook

This chapter showed how gem5's atomic mode can be parallelized using a parallel Temporal Decoupling (TD) approach. The idea is to subdivide the simulation into individual threads that run independently for a given time called quantum. When modeling a 128-core system on a 128-core host, speedups of up to $24.7\times$ can be attained for a synthetic best-case benchmark, while retaining more than 99% accuracy for most simulation statistics. For the NPB benchmark, a maximum speedup of $12\times$ can be achieved.

Since communication between individual threads may be postponed by up to a quantum length, accuracy and functionality of the simulator may be affected. However, observations showed that modeling a general-purpose ARM Multiprocessor System on A Chip (MPSoC) running a Linux OS does not lead to any functional flaws for relevant quanta. Yet there is an influence on the accuracy of various simulation parameters.

Ultimately, finding a quantum that allows for high accuracy with high speedup is one of the main challenges. While a larger quantum allows for greater speedups, a smaller quantum leads to greater accuracy in terms of simulation statistics. In order to find an optimal quantum without time-consuming reference simulations, a temporal error estimation was incorporated into the simulator. By tracking communication between individual simulation threads, inaccuracies in simulations can be reliably estimated. An analytical assessment of the relationship between accuracy and speedup is provided in the next chapter (Chapter 4).

While it may not be obvious from the previous subsections, but reworking gem5 to be parallel was a two-year project with multiple engineers involved. Not only needs the simulation engine to be rewritten, but all simulation models need to be reworked as well. Furthermore, when working on the parallel timing mode as shown in [50], some concepts like atomic snoops cannot be transferred to a parallel simulation without further ado. Hence, parallelization in FSSs should be considered from the earliest design stages. Such a process could also be aided by a parallel simulation standard - a necessity the simulation community still lacks.

Chapter 4

Temporal Decoupling

Temporal Decoupling (TD) is a simulation technique in which different segments of the simulation are executed independently for short periods of target time. From fewer context switches to parallelization; TD allows for several optimizations that increase a simulator's performance. The *par-gem5* simulator described in the previous chapter (Chapter 3) also uses TD to achieve speedups of $20\times$ and more through parallelization. It should be noted that TD is often understood as a method that accelerates sequential simulations. This is due to the SystemC TLM-2.0 standard [2] for sequential simulations, which standardized the concept in 2008 and made it accessible to a large audience from industry to academia. It was also this standard that coined the term *temporal decoupling*. However, the actual concept of TD reaches back to the 1960s. In a work by Fuchi et al. [69] from 1969, a method similar to TD is described. Fuchi et al. already use the term *quantum* to refer to the length of an uninterrupted simulation period, yet, the overall concept is referred to as *time slice-based*.

Whilst TD enables the accelerated execution of simulations, changing from a global time to multiple local times can impact the functionality/accuracy of the simulation. As already shown in the previous chapter, the challenge of TD is to find an optimal value for the maximum time difference of the individual simulation segments. This value is referred to as *quantum* $t_{\Delta q}$. Smaller quanta are beneficial for a simulation's accuracy, while larger quanta allow for maximum performance (see Figure 4.1).

Balancing these competing priorities creates the dilemma of finding a quantum with reasonable simulation speed and acceptable accuracy. To date, there is no comprehensive work that thoroughly explains the relationship between accuracy and

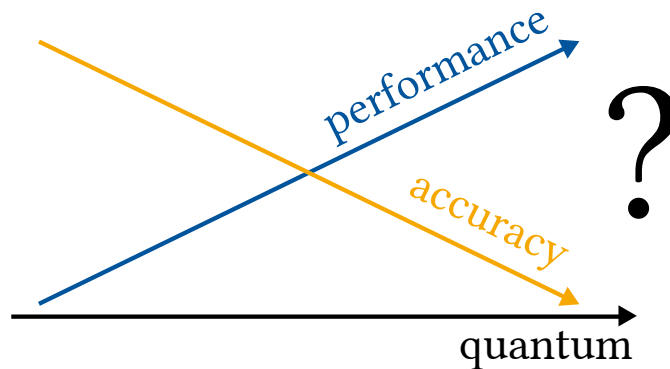


Figure 4.1: Simplified mental model of accuracy and performance used in many works using TD.

speedup. Instead, TD users in industry and academia rely on rules of thumb and empirical observations to determine an optimal quantum:

- Fuchi et al. [69], 1969: “The smaller the time-slice quantum becomes, the more accurate the simulation of the parallelism is.”
- Ryckbosch et al. [163], 2012: “We set the simulation window to 10 ms and the simulation quantum to 100 ms in all of our experiments. We experimentally evaluated different values for the simulation window and quantum, and we found the above values to be effective.”
- J. Engblom [58], 2018: “Time quantum lengths of 10 k to 1 M cycles are needed to maximize VP performance. Most of the time, software functionality and correctness are unaffected by TD, and the default should be to use long time quanta.”
- J. Joy [104], 2020: “Increasing the quantum can cause a thread to run for a longer time, thus reducing the context switching overhead. This increases the simulation speed, but at the cost of accuracy.”
- Jünger et al. [106], 2021: “To increase performance, the quantum should be as large as possible to reduce context switching. However, a large quantum reduces simulation accuracy, as events may be handled too late. Therefore, deploying TD is not trivial.”

This chapter addresses the shortcoming by introducing analytical models for estimating the performance and accuracy of TD-based simulations. These models not only help identify the optimal trade-off between speedup and accuracy but also provide insights into the underlying observations. As discussed in the following sections, increasing the quantum leads to diminishing performance returns while the inaccuracy continues to grow without bound. For excessively large quanta, performance gains may reverse, and the functionality of the simulator can be compromised. This analysis aims to set realistic expectations for simulation users and clarify why a larger quantum does not always result in higher performance. The validity of these models is demonstrated using benchmark examples implemented in SystemC and gem5. In general, this chapter is largely based on the author’s previous works: *The Optimal Quantum of Temporal Decoupling* [213], *The Art of Temporal Decoupling* [205].

4.1 Background

In this section, background information on TD and analytical models are provided.

4.1.1 Temporal Decoupling (TD)

TD enables individual simulation processes to advance in their local time by a duration $t_{\Delta q}$ relative to the rest of the system. This reduces the frequency of context switches and facilitates parallelization, significantly enhancing simulation performance. Figure 4.2 illustrates the TD principle, comparing sequential and TD-based simulations. In Case ①, a standard sequential simulation is shown with two processes (P1, P2) being alternatively executed. This system could represent a dual-core MPSoC with each event corresponding to a single instruction executed in a fixed time (e.g., 1 ns).

For correct chronological order, the simulation kernel switches processes after each instruction/event (Case ①). While executing one target instruction typically requires only a few instructions of the host system, context switching incurs a significant overhead — especially in frameworks like SystemC, where processes are implemented as coroutines. TD reduces the number of context switches by allowing each simulation process to execute without interruption for up to a quantum $t_{\Delta q}$ (Case ②). For instance, in the aforementioned dual-core example with a 1 GHz clock speed, a quantum of $t_{\Delta q} = 3$ ns permits a continuous execution of three instructions. This reduces the number of context switches from six to two, saving a wall-clock time of $t_{\Delta S}$.

As shown in Case ③, TD also facilitates parallel simulations. Here, each process operates in its own thread with an independent local time and synchronizes with other threads at a barrier after every $t_{\Delta q}$. Such a concept was also used in the previous chapter for the implementation of the parallel simulator par-gem5. Parallel TD differs fundamentally from classical PDES, which uses either conservative or optimistic approaches to handle causality errors. Conservative PDES prevents such errors entirely, while optimistic PDES resolves them through rollbacks. Since TD allows causality errors to occur, it does not fit into either PDES category.

Both sequential and parallel TD can result in incorrect chronological execution. However, general-purpose computing systems are highly tolerant of minor time skews. For example, a CPU operating a few cycles ahead or a timer interrupt delayed by a

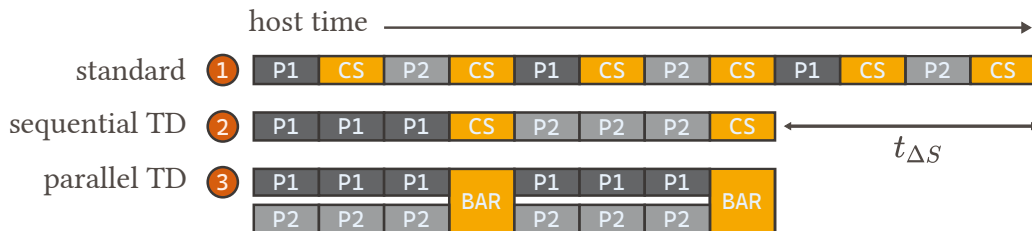


Figure 4.2: Principle of TD.

few microseconds typically does not impact most general-purpose applications. In such cases, the timing accuracy of the simulated system is only marginally affected. Nevertheless, for large quantum sizes (e.g., in the order of milliseconds), functionality may degrade. To mitigate these effects, TD allows early termination of quanta in sequential simulations. For instance, if a TD executes a Wait-For-Interrupt (WFI) instruction, the simulation can predict that no further meaningful tasks will occur until the quantum ends, as the anticipated interrupt will originate from another simulation thread. In such scenarios, the quantum can be terminated early to maintain both accuracy and performance.

4.1.2 Analytical Models

Since the advent of modern microprocessors, computer engineers have sought to abstract complex computing systems using analytical models. These models aim to evaluate key performance metrics such as latency, throughput, or power consumption through mathematical methods. This approach offers both advantages and limitations. On the one hand, analytical models are significantly faster than simulation-based methods, enabling the exploration of entire design spaces within seconds. On the other hand, they often rely on incomplete or oversimplified assumptions, which can compromise their accuracy. As a result, analytical models are frequently criticized for their imprecision. Beyond quantitative evaluation, analytical models also serve to convey fundamental insights. For example, Amdahl's Law [6] cannot precisely predict the speedup of a complex multi-core server workload when increasing thread counts. However, its qualitative insight into diminishing returns with higher core counts remains a foundational concept for computer architects and software engineers. The analytical models introduced in the following sections should be viewed in a similar light. Rather than aiming for perfect accuracy, the proposed parsimonious model provides a framework to help users understand the performance and accuracy boundaries of their simulations.

4.2 Related Work

As discussed in the introduction of this chapter, the origins of TD trace back to a work of Fuchi et al. [69] from 1969. While they referred to the approach as *slice-based*, they already introduced the term *quantum* for the length of a time slice. The modern term *Temporal Decoupling* (TD) emerged nearly 40 years later with the release of the SystemC TLM-2.0 standard [144, 2] in 2008. Despite its practical significance, the determination of the optimal quantum remains a largely unexplored research area.

Several studies have sought to enhance the accuracy of TD-based simulations. For instance, Jung et al. [105] introduced *speculative temporal decoupling*, which achieves error-free simulations by re-executing erroneous sections with tighter synchronization. Similar techniques have been employed by Busnot et al. [30] and Jefferson [101]. Another approach, *predictive temporal decoupling*, was proposed by Glaser et al. [75], who used a Wiener filter to predict future events and dynamically adjust the quantum length to minimize inaccuracies. To address the diminishing performance benefits of dynamic TD, Jünger et al. [107] demonstrated how analyzing the relationships between events can support larger quantum sizes without reducing simulation accuracy. In parallel TD, a common strategy is to set the quantum to the minimum delay of a communication channel [136, 194, 28]. While this approach significantly limits use cases, it allows for error-free simulations. For example, *dist-gem5* [136] simulates distributed systems by using the communication delay of the network interface controller as the quantum. The intersection of analytical models and TD simulations has been explored in only a few works, notably by Falsafi et al. [63] and Over et al. [147].

Falsafi et al. [63] developed an analytical performance model for the *Wisconsin Wind Tunnel Simulator*, aimed at assessing the financial cost-effectiveness of parallel simulations. Their model, based on the Kruskal and Weiss model for fork-join programs [118], focuses exclusively on parallel simulations and employs a static quantum of 100 cycles. Consequently, their performance analysis does not depend on the quantum, and no error model is proposed, as their simulations were designed to be error-free.

Over et al. [147] employed analytical models to compare two parallelization methods in the *Sulima* simulator, which models an UltraSPARC III Cu. Unlike Falsafi et al., their model depends not on the quantum but on the number of cores and the average execution time per quantum. While Over et al. analyzed simulation accuracy, they did not propose an analytical model to explain their observations.

To the best of the author's knowledge, this is the first work (excluding the work on which this chapter is based on) to analytically evaluate the impact of both sequential and parallel TD on simulation performance and accuracy. It aims to bridge the gap between the qualitative insights provided by earlier works and a comprehensive understanding of a TD-based simulation's performance and accuracy limits.

4.3 Methods

In this section, analytical models for the performance and accuracy estimation of temporally-decoupled simulations are introduced.

4.3.1 Analytical Speedup Model

This subsection introduces mathematical models to estimate the expected simulation speed and accuracy as functions of the quantum. Given the distinct principles underlying parallel and sequential TD, separate analytical models are developed to evaluate performance for each approach.

4.3.1.1 Temporally-Decoupled Sequential Simulation

This section presents an analytical model for estimating the performance of sequential TD-based simulations as implemented in the SystemC standard. In sequential TD, the achieved speedup primarily results from a reduction in the number of context switches. Hence, for an ideal simulation without any context-switching cost, the ideal time (T_{ideal}) is the sum of the time of all simulation segments T_i (see Figure 4.3). In practice, a relative overhead O_c is imposed by the context switches:

$$T_{real} = T_{ideal} \cdot (1 + O_c) \quad (4.1)$$

The overhead O_c is inversely proportional to the chosen quantum, $t_{\Delta q}$, since doubling the quantum halves the number of context switches. This proportionality assumes that the quantum is larger than the average event distance, which is typically valid in most real-world scenarios. For example, in a CPU simulation with a clock cycle of 1 ns, using quanta smaller than 1 ns would have no effect.

$$T_{real} = T_{ideal} \cdot \left(1 + \frac{O'_c}{t_{\Delta q}}\right) \quad (4.2)$$

Here, O'_c is a constant overhead factor representing the context-switching overhead, which can be determined using reference simulations. For instance, by comparing two simulations with different quantum sizes $t_{\Delta q1}$ and $t_{\Delta q2}$, O'_c can be derived as:

$$\frac{T_{real}(t_{\Delta q1})}{T_{real}(t_{\Delta q2})} = \frac{1 + \frac{O'_c}{t_{\Delta q1}}}{1 + \frac{O'_c}{t_{\Delta q2}}} \Rightarrow O'_c = \frac{T(t_{\Delta q1}) - T(t_{\Delta q2})}{\frac{T(t_{\Delta q2})}{t_{\Delta q1}} - \frac{T(t_{\Delta q1})}{t_{\Delta q2}}} \quad (4.3)$$

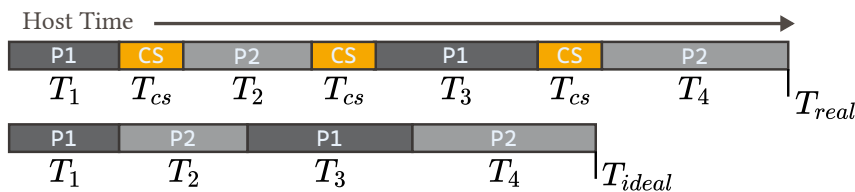


Figure 4.3: Example timings for a sequential temporally-decoupled simulation with (T_{real}) and without (T_{ideal}) context-switching costs.

Alternatively, O'_c can be estimated through multiple reference simulations and curve fitting. Ultimately, the performance achieved by TD can be calculated by dividing the ideal simulation time (T_{ideal}) by the real simulation time (T_{real}), resulting in the slowdown S :

$$S(t_{\Delta q}) = \frac{T_{ideal}}{T_{real}} = \frac{t_{\Delta q}}{t_{\Delta q} + O'_c} = \frac{1}{1 + \frac{O'_c}{t_{\Delta q}}} \quad (4.4)$$

This formula is designed to yield values between 0 and 1 for several reasons. First, TD is fundamentally a method to mitigate the performance degradation caused by context switches, rather than a means of surpassing ideal performance. Thus, $S(t_{\Delta q})$ reflects how closely the simulation approaches the theoretical optimal performance. For instance, if the calculated value is 0.99, it indicates that further increasing the quantum will likely result in negligible performance improvements. This representation makes it straightforward to evaluate the efficiency of TD and determine the point of diminishing returns in terms of quantum size.

4.3.1.2 Temporally-Decoupled Parallel Simulation

The following model describes the speedup of a TD parallel simulation, as for example, used by par-gem5 [214] from Chapter 3. As explained in Section 4.1.1 and illustrated in Figure 4.2 Case ③, this type of simulation leverages the host system's multi-threading capability to execute segments of the simulation in parallel. For instance, each target CPU could be executed in its own thread when simulating an MPSoC. This is a common scenario and forms the basis of the assumptions used in the following analysis. Assuming the target workload and system simulation are perfectly parallelizable with no synchronization barriers, the speedup S is:

$$S(N) = T/(T/N) = N \quad (4.5)$$

Here, N is the number of simulation threads and T is the host time, or wall-clock time, required for a sequential simulation. According to this equation, the speedup scales linearly with the number of simulation threads, representing an upper bound. However, in real-world scenarios, synchronization overhead imposes additional constraints, reducing the achievable speedup. In synchronous parallel simulations, periodic barrier events introduce synchronization overhead. This overhead can be modeled using a relative workload factor $O_b(t_{\Delta q}, N)$, which depends on the length of the quantum $t_{\Delta q}$ and the number of threads N :

$$S(N, t_{\Delta q}) = \frac{N}{1 + O_b(t_{\Delta q})} = \left(\frac{1}{N} + \frac{O_b(t_{\Delta q}, N)}{N} \right)^{-1} \quad (4.6)$$

Similar, to the previous subsection, this equation can be refined by inserting the inverse proportional relationship between quantum and overhead. For instance, dou-

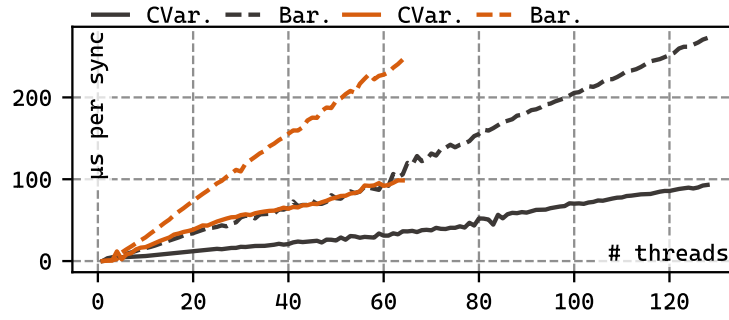


Figure 4.4: Time per sync for condition variables and barriers on Linux (black) and Windows (orange).

bling the quantum size halves the number of barrier events. Thus, the overhead can be refined to $O_b(t_{\Delta q}, N) = O'_b(N)/t_{\Delta q}$. Inserting this into Equation 4.6 yields:

$$S(N, t_{\Delta q}) = \left(\frac{1}{N} + \frac{O'_b(N)}{N \cdot t_{\Delta q}} \right)^{-1} \quad (4.7)$$

On modern OSs, the synchronization overhead imposed by barriers grows linearly with the number of threads. This behavior is consistent across various OSs, hardware platforms, and barrier types (see Figure 4.4). As a result, the overhead can be further refined to $O'_b(N) = O''_b \cdot N$, simplifying the equation to:

$$S(N, t_{\Delta q}) = \left(\frac{1}{N} + \frac{O''_b}{t_{\Delta q}} \right)^{-1} \quad (4.8)$$

This formula accounts for the linear growth of overhead with thread count. Alternative barrier implementations with logarithmic scaling, $O_b(\log(N))$, are theoretically possible [131], but the primary overhead factor O''_b remains as a measure of synchronization cost independent of core count. By rearranging Equation 4.8, O''_b can be determined as:

$$O''_b = t_{\Delta q} \cdot \frac{N - S(N, t_{\Delta q})}{S(N, t_{\Delta q}) \cdot N} = t_{\Delta q} \cdot \frac{t_N(t_{\Delta q}) - t_{seq}/N}{t_{seq}} \quad (4.9)$$

Here, $t_N(t_{\Delta q})$ is the host execution time of a parallel simulation with N cores/threads and a quantum of $t_{\Delta q}$. The host execution time for the sequential reference simulation is given by t_{seq} . Consequently, two simulation runs (one sequential, one parallel) are required to estimate O''_b . The parameter O''_b connects workload characteristics with synchronization overhead and must be determined for each specific workload. In practice, variations in O''_b are minimal across workloads, allowing for reliable measurements from a single measurement. For example, using par-gem5 from the previous chapter, all observed values of O''_b were between 2 ns and 6 ns.

Using Equation 4.8, the results shown in Figure 4.5 were obtained. While the analytical model provides an upper bound for the achievable speedup, it does not

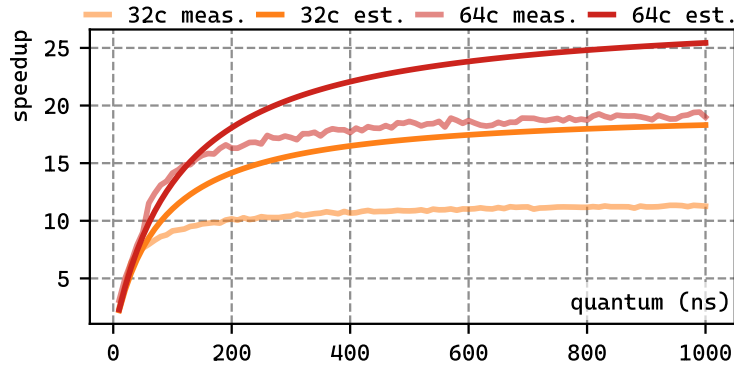


Figure 4.5: Estimated speedup vs. measurement for a bare-metal bubble sort benchmark using the simple, imprecise model as described by Equation 4.8.

precisely predict actual performance. This discrepancy arises because the assumption of 100% parallelism in the simulation does not hold true in practice.

The formula can be refined by accounting for sequential portions of the simulation. For instance, there could be a global state in the simulator, which only allows for serial access due to thread safety. But also the degree of parallelism in the target workload influences speedup. For example, a single-core Dhrystone benchmark will see no performance improvement from parallel execution, as the workload is inherently sequential. To incorporate varying degrees of parallelism, the formula is extended by *Amdahl's Law* [6] to:

$$S = \left(\frac{p}{N} + 1 - p + \frac{O_b''}{t_{\Delta q}} \right)^{-1} \quad (4.10)$$

With Amdahl's Law, a factor $p \in [0, 1]$ indicates to what extent the executed workload is parallelizable. Since p represents an unknown workload characteristic, it must be determined empirically. One approach is to derive p by solving a system of linear equations based on the extended formula (Equation 4.10). In the following, the least error squares method is used to determine p and O_b'' .

4.3.2 Analytical Accuracy Model

While the term “speedup” is clearly defined, “inaccuracy” can be interpreted in various ways. In this thesis, simulation inaccuracy is categorized into two aspects: qualitative and quantitative.

Qualitative inaccuracy refers to effects that cannot be easily expressed as numerical metrics but may impact the functionality of the simulation. For instance, if TD causes a program to crash during simulation, this would be considered qualitative inaccuracy.

In contrast, *quantitative inaccuracy* encompasses measurable deviations caused by TD. Examples include variations in cache hit rates, interrupt timings, or total simulation time. Since simulators like gem5 provide extensive simulation statistics, summarizing quantitative inaccuracy in a clear and concise manner is essential. For this

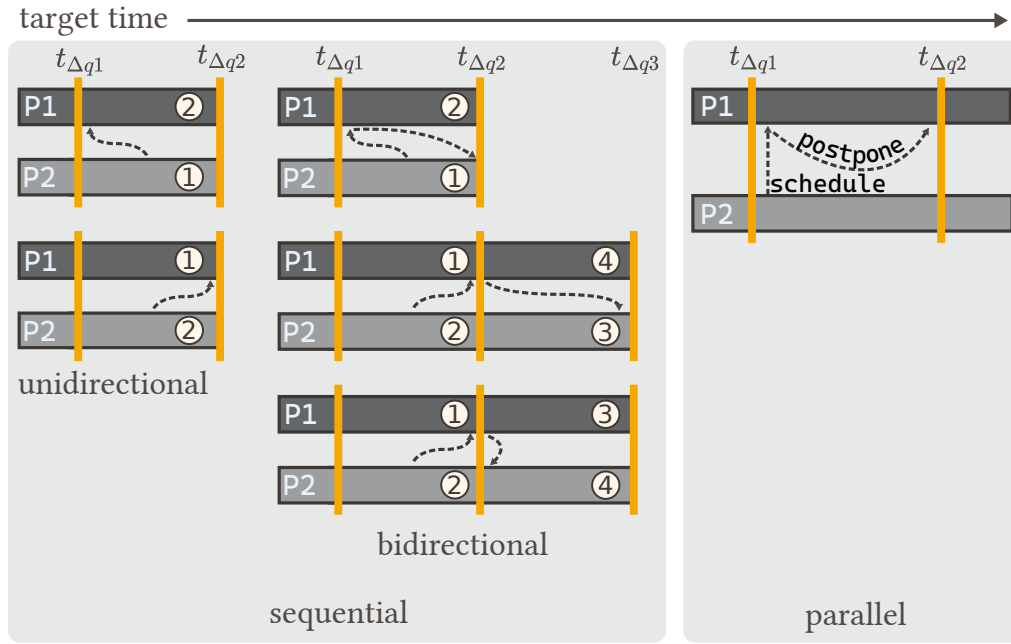


Figure 4.6: Types of timing errors in TD-based simulations.

thesis, target simulation time is chosen as a representative metric for assessing quantitative (in)accuracy. This metric is commonly available across simulators and inherently reflects the influence of other factors, such as interrupt timing deviations. Ultimately, simulation users must determine which specific statistics are most relevant to their use case, as the importance of various metrics may vary depending on the simulation's objectives.

4.3.2.1 Quantitative Inaccuracy: Parallel Simulation

For TD-based simulations, any form of inaccuracy arises from an incorrect execution order of events. This means that, from a process's perspective, events occur either too early or too late. Figure 4.6 illustrates this phenomenon for both sequential and parallel TD simulations. The parallel case specifically refers to TD as implemented in par-gem5 from the previous chapter. In parallel TD, events scheduled across thread boundaries are postponed to the next quantum boundary if their event time occurs before the next synchronization barrier. This postponement ensures correctness, as the target thread may already be waiting at the quantum boundary. However, if the postponed event lies on the simulation's critical path, the target simulation time extends accordingly. To analytically estimate this timing extension, the following assumptions are made:

1. Most events are scheduled without delay, consistent with observations in par-gem5. For instance, cache snoops or SEV instructions are typically scheduled quasi-instantaneously.
2. Cross-scheduled events occur stochastically and independently at a constant rate of r , i.e. $Y \sim \text{Poisson}(r)$.

Under these assumptions, the time until the first cross-scheduled event occurs within a quantum follows an exponential distribution:

$$X \sim \text{Exponential}(r) \implies p(r) = re^{-rt} \quad (4.11)$$

The average simulation time extension t_d per quantum can then be derived as:

$$\begin{aligned} t_d &= t_{\Delta q} - E(X|X \leq t_{\Delta q})P(X < t_{\Delta q}) - t_{\Delta q}P(X > t_{\Delta q}) \\ &= t_{\Delta q} - \int_0^{t_{\Delta q}} rte^{-rt} dt - \int_{t_{\Delta q}}^{\infty} rt_{\Delta q}e^{-rt} dt \\ &= t_{\Delta q} - (1 - e^{-rt_{\Delta q}})/r \end{aligned} \quad (4.12)$$

With the average delay per quantum t_d , the relative timing inaccuracy can be formulated as:

$$I = \frac{t_{\Delta q}}{t_{\Delta q} - t_d} - 1 = \frac{r \cdot t_{\Delta q}}{1 - e^{-rt_{\Delta q}}} - 1 \approx r \cdot t_{\Delta q} - 1 \quad (4.13)$$

As can be seen from Equation 4.13, the relative timing inaccuracy can be described by a hockey stick curve. Initially, the relative inaccuracy grows exponentially for small quanta. As the quantum increases, the growth becomes linear $r \cdot t_{\Delta q}$, which also serves as an upper bound. This linear growth of inaccuracy stands in stark contrast to the saturating speedup model. Unlike speedup, which eventually saturates, timing inaccuracy grows indefinitely with increasing quantum size. This underscores the importance of selecting an optimal quantum to balance speedup and accuracy.

4.3.2.2 Quantitative Inaccuracy: Sequential Simulation

In parallel simulations, such as *par-gem5*, timing inaccuracy primarily arises from the postponement mechanism. In contrast, the situation is more complex for sequential TD-based simulations, such as those implemented in SystemC. Assuming unidirectional communication between two processes (P1 and P2), as illustrated in Figure 4.6, two types of timing errors can occur:

1. Message received too early: If P2 executes first and sends a message to P1 in the middle of the quantum, P1 will receive the message at the beginning of its quantum. This results in the message being received too early by $t_{\Delta q}/2$.
2. Message received too late: If P1 executes first, the message from P2 will not reach P1 until the start of the next quantum, causing a delay of $t_{\Delta q}/2$.

Assuming both cases are equally probable and neglecting the characteristics of the target software, these deviations cancel each other out on average, resulting in no net timing error. The situation differs significantly for bidirectional communication, as shown in Figure 4.6. Here, P2 sends a message to P1, which then responds. For simplicity, it is assumed that the time required for sending and receiving messages is negligible compared to the quantum length. In this scenario, regardless of the execution order, the simulation time is extended. For communication initiated in

the middle of a quantum, the extension can be either $t_{\Delta q}/2$ or $3t_{\Delta q}/2$ depending on the sequence of events. Modeling the frequency of communication attempts using a Poisson distribution also leads to an inaccuracy that linearly depends on the quantum (similar to Equation 4.13).

Hence, it can be concluded that in both the parallel and sequential cases, the relative timing inaccuracy increases linearly with the quantum. Without specifying the exact linear coefficient, the inaccuracy I can be expressed as:

$$I = \alpha \cdot t_{\Delta q} \quad (4.14)$$

The factor α can be determined through two reference simulations or curve fitting. Ultimately, with formulas for performance and inaccuracy, the initial performance/i-naccuracy model of TD can be refined to as follows: The inaccuracy increases linearly with the quantum, while the performance returns eventually diminish (see Figure 4.7).

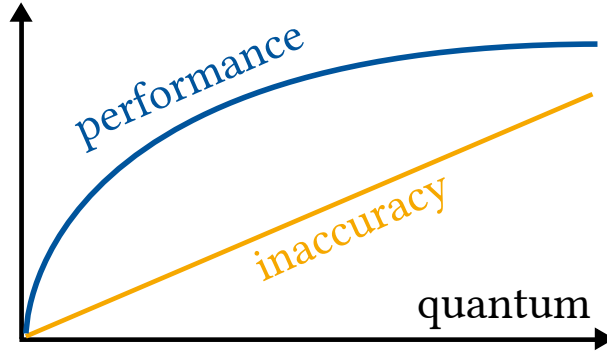


Figure 4.7: Refined accuracy and performance mental model of TD.

4.3.3 Speedup at Large Quanta

As an iconic aphorism in statistics says: All models are wrong, but some are useful. Similarly, also the model of this thesis may encounter situations in which the observations do not match the model's predictions. To rectify the deviations a common practice is to introduce fudge factors. An example, where a fudge factor can be applied, is simulating with large quanta. As later shown in the results, quantitative inaccuracy may begin to negatively impact the simulator's performance at extremely high quanta for some benchmarks. To account for this the previously defined speedup models can be extended by a linear error term $k \cdot t_{\Delta q}$:

$$S_{seq}(t_{\Delta q}) = \frac{1}{1 + \frac{O'_c}{t_{\Delta q}} + k \cdot t_{\Delta q}} \quad S_{par}(t_{\Delta q}) = \left(\frac{p}{N} + 1 - p + \frac{O''_b}{t_{\Delta q}} + k \cdot t_{\Delta q} \right)^{-1} \quad (4.15)$$

Similar to the inaccuracy model, the linear error term is a characteristic of the workload, and has to be obtained by empirical basis. An explanation is later given in the results section.

4.4 Results & Discussion

To validate the proposed analytical model and its accuracy, reference simulations were conducted. All experiments were performed on an *AMD Ryzen 3990x* host system (64 physical cores/128 logical threads) running Ubuntu 21.10

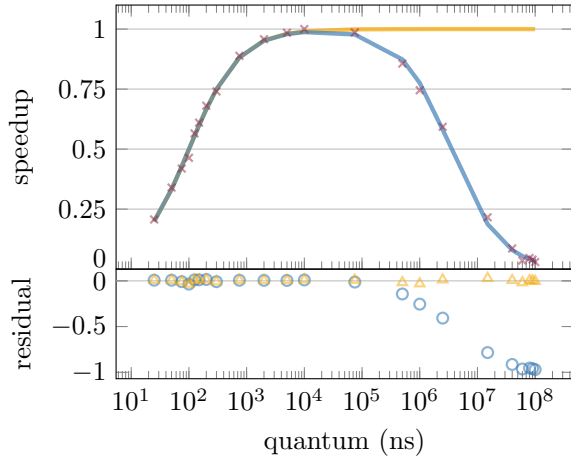
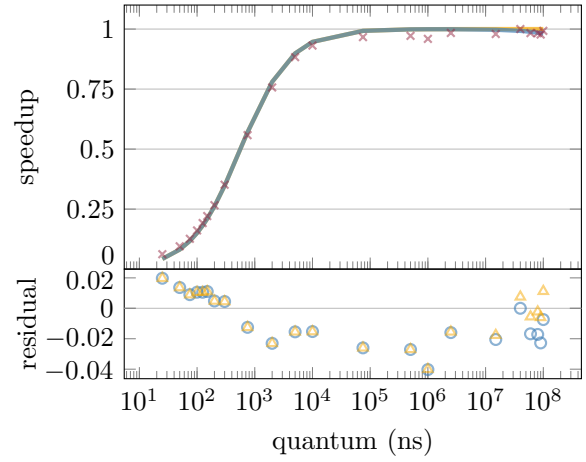
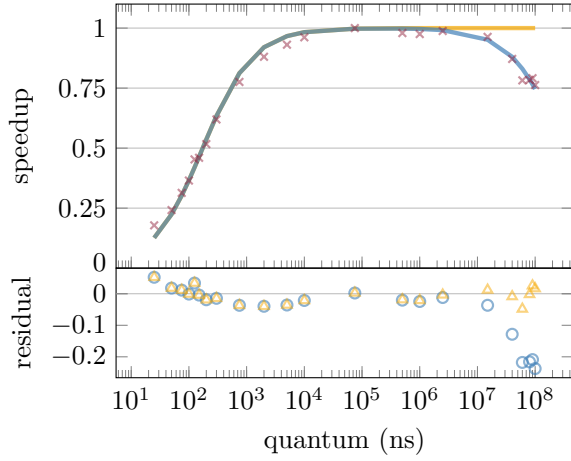
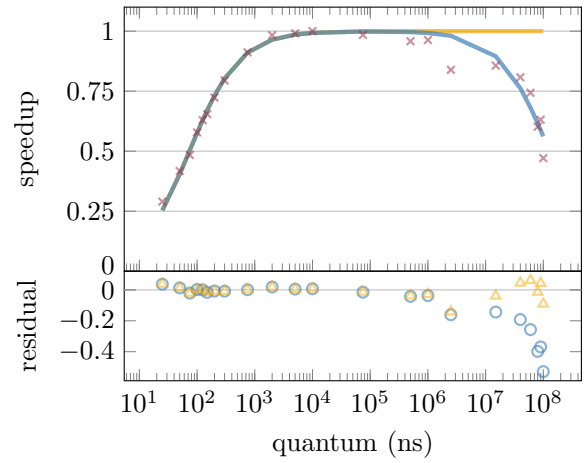
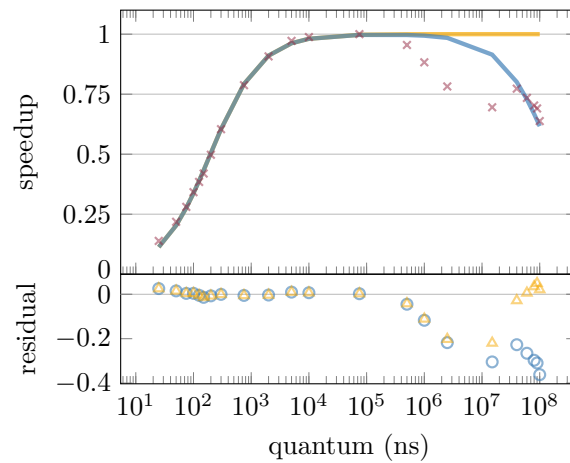
4.4.1 Sequential Temporal Decoupling

As a reference implementation for sequential TD-based simulation, the open-source ARMv8 TLM-2.0-based virtual platform *AVP64* [10] was used. This platform features eight CPUs clocked at 1 GHz. The benchmarks selected for evaluation included Dhrystone, NPB, STREAM, and a Linux operating system boot, all executed on a Buildroot-configured Linux system. The performance results are shown in Figure 4.8, where measurement points are represented by “ $\times$ ” and the model’s prediction are given by the continuous lines. The blue line reflects the model with a linear error term (see Subsection 4.3.3), while the orange line excludes this term.

Across all benchmarks, speedup begins to saturate at quantum values exceeding 10 μ s. As later shown in Subsection 4.4.2, this stands in contrast to the par-gem5 results, where saturation already occurs at 1 μ s. The difference can be attributed to the efficient DBT backend of AVP64 and the relatively high overhead of context switches. This is reflected in the factor O'_c . For example, the Dhrystone benchmark has an overhead of $O'_c = 564$ ns, indicating that executing 564 cycles takes as much time as a single SystemC context switch. These observations highlight why TD has become integral to ESL simulations. For instance, with a 10 μ s quantum, the Dhrystone benchmark requires about 10 minutes of host simulation time. Without TD, the same simulation would take over 2 days.

In general, the proposed analytical model provides sufficiently accurate predictions for speedup, especially at lower quanta. At higher quanta, the linear error term becomes noticeable in 4 out of the 5 benchmarks. Notably, the IS.W and STREAM benchmarks exhibit a peculiar dip pattern, which is addressed in Subsection 4.4.3

The accuracy measurements and predictions for the benchmarks are presented in Figure 4.9. Measurement points ($\times$) and model predictions (continuous lines) show a strong alignment, with accuracy following a linear trend as predicted by the model. For very high quanta, deviations from the model increase slightly. An exception is the Dhrystone benchmark, which exhibited a negative error at higher quanta. This suggests that Dhrystone computations required less target time at larger quanta, potentially due to reduced interference from devices.

(a) SystemC Linux boot ($O'_c = 100$ ns).(b) SystemC Dhrystone ($O'_c = 564$ ns).(c) EP.W ($O'_c = 173$ ns).(d) SystemC IS.W ($O'_c = 74$ ns).(e) SystemC STREAM ($O'_c = 196$ ns).**Figure 4.8:** Sequential temporal decoupling performance results.

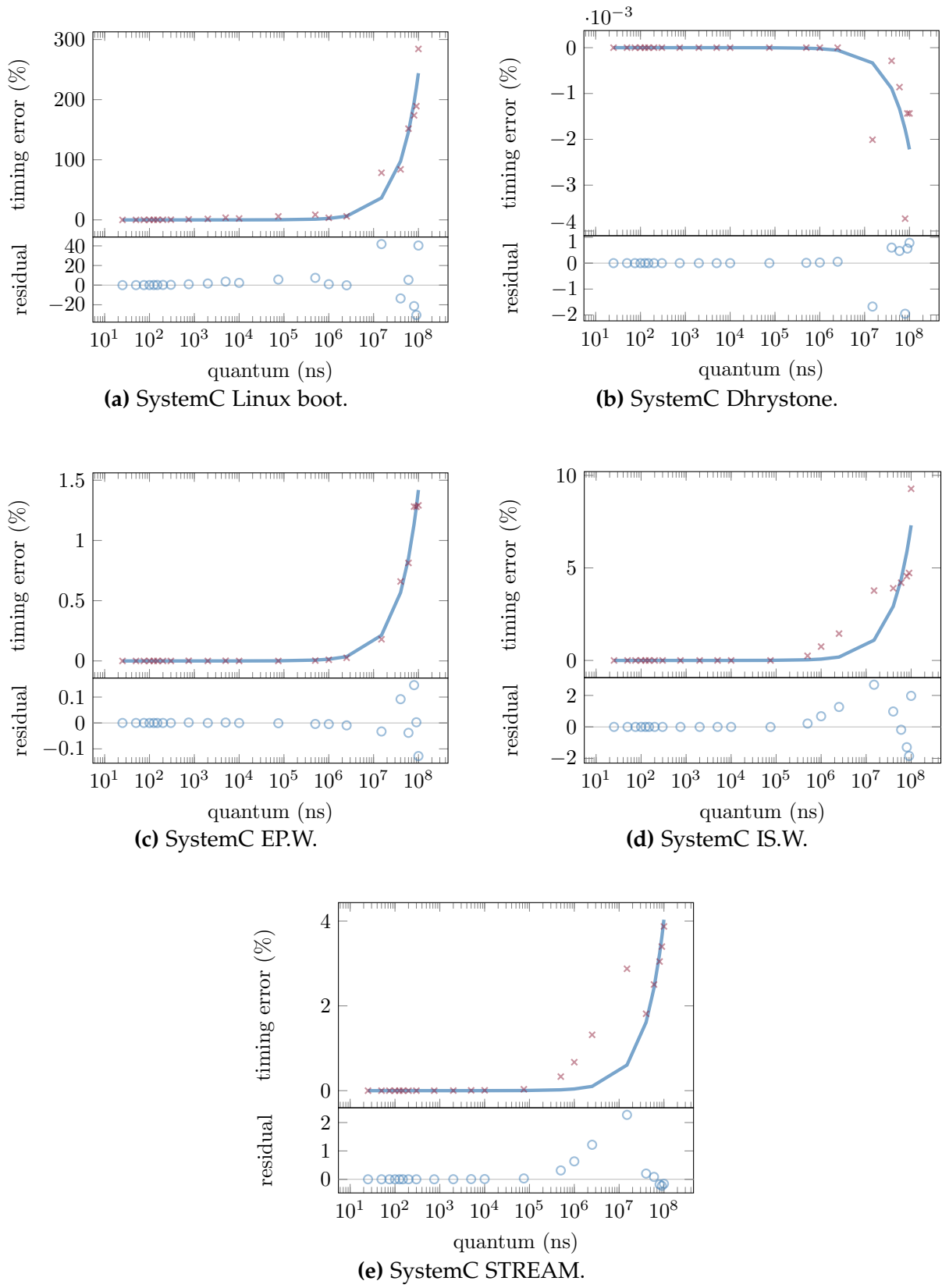


Figure 4.9: Sequential temporal decoupling accuracy results.

4.4.2 Parallel Temporal Decoupling

For parallel TD simulations, the experiments utilized par-gem5, as discussed in Chapter 3. The approach of par-gem5 aligns with the principles and formulas outlined in Section 4.1.1 and Section 4.3. The target system consists of 16, 32, 64-core Simple-CPU ARM configurations, clocked at 1 GHz, with a multi-level cache hierarchy. While the simulation incorporated some microarchitectural details, it can be classified as LT. Using this setup, the following benchmarks were executed: a bare-metal bubble sort application, NPB IS.W [143], and STREAM [127]. These benchmarks, designed for multi-core systems, distribute their workloads across all cores of the target system. Ubuntu served as the target OS for NPB IS.W and STREAM, while the bubble sort application ran in a bare-metal environment. Single-threaded benchmarks were excluded, as sequential workloads in parallel TD-based simulations do not provide speedup. The results are presented in Figure 4.10 and Figure 4.11.

In general, it can be observed that both speedup and accuracy align with the prediction of the analytical model. Achievable speedups range from $1\times$ (STREAM) to $20\times$ (bare-metal bubble sort benchmark), which is reflected in the respective parallelization factor. However, with all benchmarks sharing a factor O_b'' in the same order of magnitude (2 ns - 6 ns), speedups start to saturate at a quantum of 500 ns.

The relative inaccuracy in simulation time exhibited a quasi-linear growth, consistent with the model. The inaccuracy slope between benchmarks varies significantly by a factor of more than $20\times$. Compared to the speedup, inaccuracy showed greater variance around the predicted values (see STREAM). For STREAM, the deviations can be attributed to the periodic behavior of benchmark-internal functions. Unlike the initial assumption of a Poisson process for cross-scheduled events, these events exhibited periodic behavior. While the arithmetic mean still showed linear inaccuracy growth, certain quantum values hit particularly advantageous or disadvantageous sweet spots.

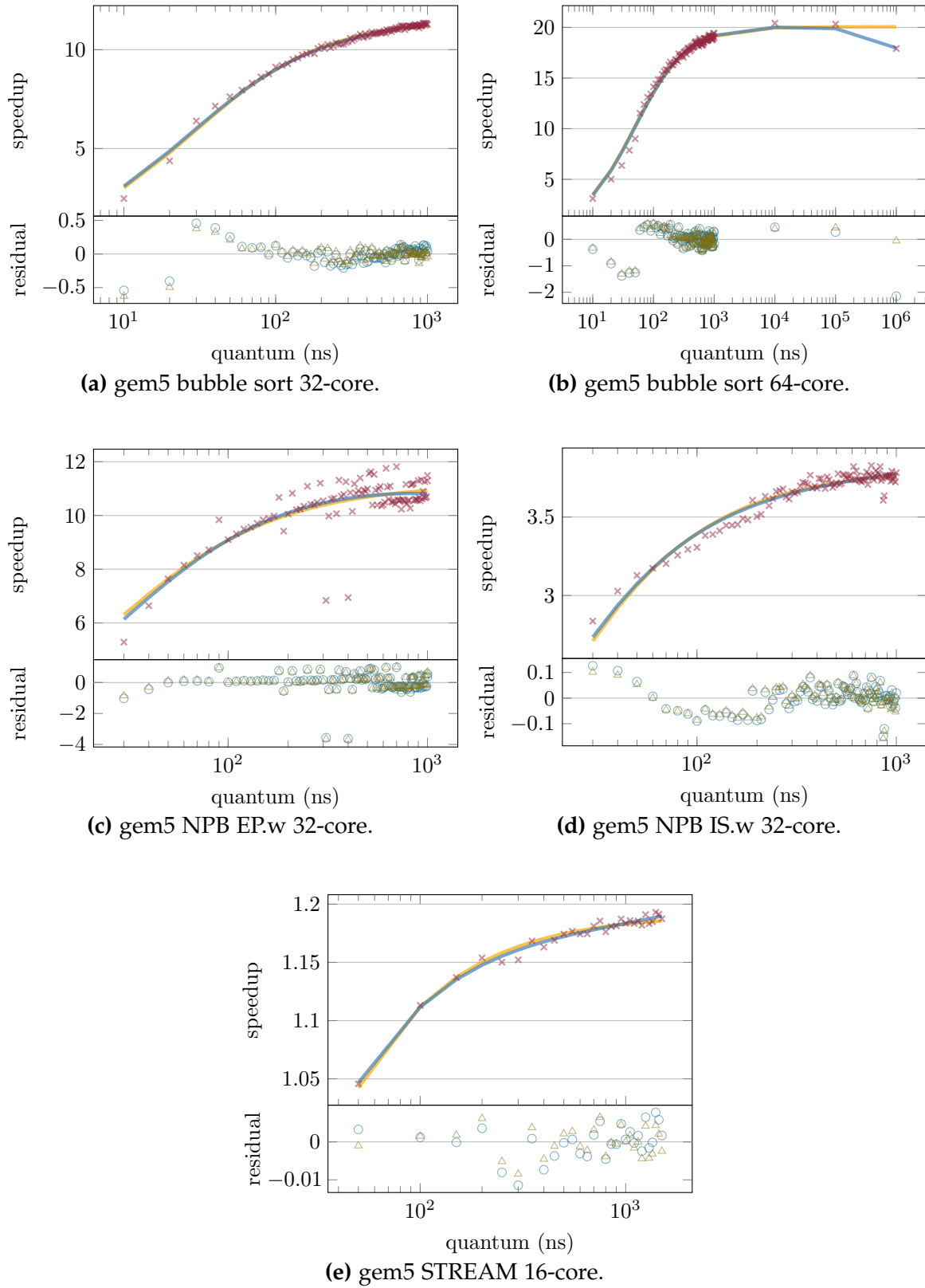


Figure 4.10: Parallel temporal decoupling speedup results.

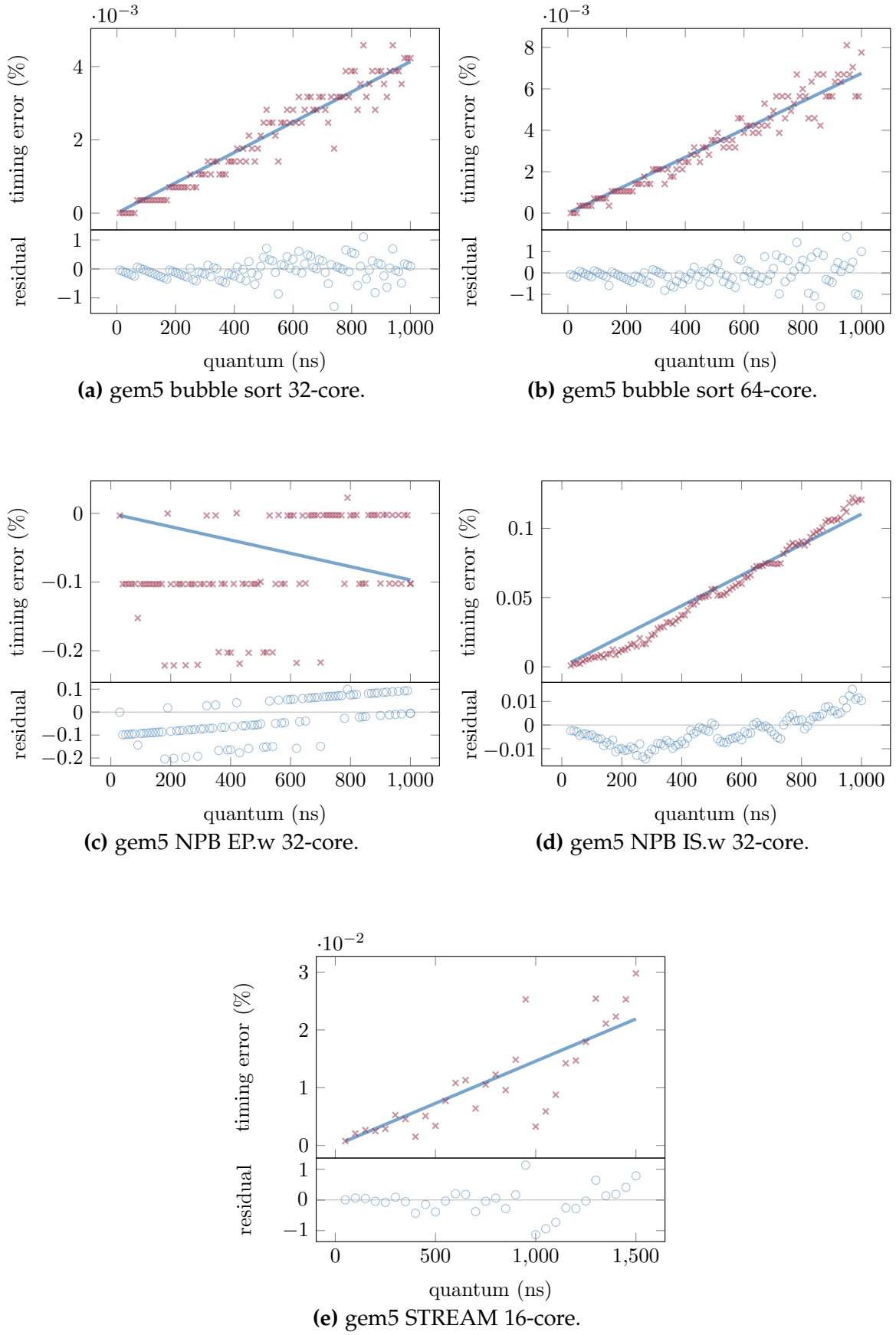


Figure 4.11: Parallel temporal decoupling accuracy results.

4.4.3 Qualitative Inaccuracy

If target software exhibits unexpected behavior during TD-based simulations, qualitative inaccuracies introduced by TD might be the underlying cause. Identifying whether TD is indeed responsible for such anomalies is often not trivial to determine, as the behavior could also stem from faulty simulation models or could simply be a misinterpretation. However, if the issue arises exclusively at very large quanta, it is likely attributable to TD. The following subsection explores incidents of qualitative inaccuracies caused by TD. In all observed cases, qualitative inaccuracies were linked either to accesses from different local time domains or to delays in communication.

The former error manifests in simulation objects that rely on simulation timestamps for functionality. In traditional simulations, observed timestamps increase monotonically, but in TD-based simulations, timestamps can seem to go back in time. If the simulation model is not designed for such cases, issues may appear. One such problem appears in the `gem5` implementation of the ARM virtual counter register `CNTVCT_EL0`, which tracks time in ARM systems. The value of this register is updated based on the time difference between the current and previous access. In TD simulations, if the previous access has a higher timestamp than the current one, the resulting delta becomes negative. Since this delta is stored in an unsigned integer, the negative value is misinterpreted as an extremely large positive number. This anomaly can be observed during a Linux boot, where the timestamps of boot messages, derived from `CNTVCT_EL0`, appear corrupted. Figure 4.12 illustrates this issue by comparing the boot logs of a classical simulation and a TD-based simulation with an improperly modeled `CNTVCT_EL0` register. The problem can be resolved by ensuring that delta updates are constrained to non-negative values. Although this adjustment slightly alters the semantics of the register, it does not appear to impact software workloads. A similar error was also documented by J. Engblom [58], who proposed an identical solution to address this issue.

A case of qualitative inaccuracy due to delayed communication was observed during the execution of the NPB benchmark using AVP64 [10]. In this scenario, the delay in communication caused a corresponding delay in the synchronization of the

TD	default	
[0.000314]	[0.000314]	pid_max: default: 32768 minimum: 301
[0.000385]	[0.000385]	Mount-cache hash table entries: 32768 [...]
[0.000396]	[0.000396]	Mountpoint-cache hash table entries: [...]
[422.828066]	[0.024140]	ASID allocator initialised with 128 entries
[3495.801687]	[0.032140]	Hierarchical SRCU implementation.
[845.656091]	[0.048162]	smp: Bringing up secondary CPUs ...
[5877.941435]	[0.080218]	Detected PIPT-Icache on CPU1
[5877.941435]	[0.080238]	CPU1: Booted secondary processor 0x000[...]
[1691.299953]	[0.080297]	smp: Brought up 1 node, 2 CPUs

Figure 4.12: Comparing linux boot timestamps of a TD and a default simulation in `par-gem5`.

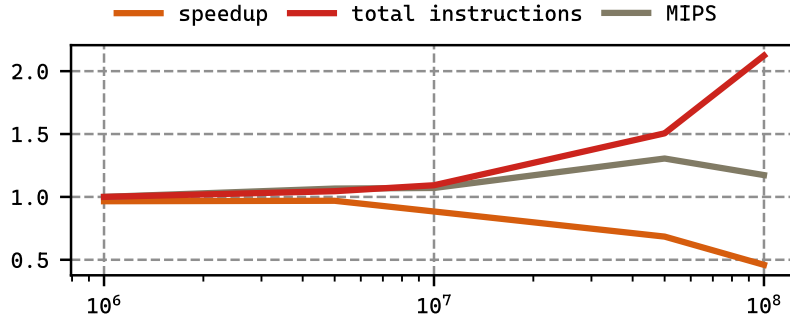


Figure 4.13: Comparing MIPS, speedup, and instructions for the NPB IS benchmark running on the AVP64 SystemC VP. Each metric was normalized to their value at $t_{\Delta q} = 1$ ms.

benchmark’s threads. Typically, synchronization is achieved through instructions that put the CPU into a low-power state, such as WFI. Under normal conditions, larger quanta would only impact the target simulation time. However, due to a bug in AVP64, the WFI instruction was executed as a NOP. According to the ARMv8 reference manual [11] (see “D1.6.2 Wait for Interrupt mechanism”), executing WFI as a NOP is permissible. To address this possibility, WFI instructions are often accompanied by a spin loop to ensure synchronization even when WFI acts as a NOP. In this case, large quanta introduce an interesting effect: the total number of executed instructions increases, leading to a decrease in the speedup measured in host execution time. However, the simulator’s speedup measured in MIPS stagnated or even improved, as NOPs are trivial to simulate. As illustrated in Figure 4.13, these effects begin to appear at $t_{\Delta q} > 1$ ms, and by $t_{\Delta q} > 100$ ms, more than half of the simulation time is spent in synchronization spin loops. This observation highlights the limitations of using MIPS as the sole performance metric for TD simulations. Furthermore, it demonstrates how TD can reveal bugs or questionable implementations that might otherwise remain undetected.

In addition to impacting simulation performance, delayed communication can also impair the throughput or functionality of peripherals. Such effects were observed during the execution of the *iperf3* [61] benchmark on AVP64, with the Virtual Platform (VP) acting as the server and the host system as the client. This benchmark was configured to measure the maximum throughput of a TCP-based connection between the server and the client. Figure 4.14 illustrates the throughput as a function of the quantum size. The results reveal a significant decline in throughput, dropping sharply from 2690 Mbit/s at $t_{\Delta q} = 1 \mu\text{s}$ to just 77 Mbit/s at $t_{\Delta q} = 100 \mu\text{s}$. Counterintuitively, increasing the quantum size led to reduced performance. A detailed investigation identified the implementation of the OpenCores Ethernet device *ETHOC* [145] in VCML [125] as the root cause. This virtual model, used by AVP64 as a network adapter, spawns a thread for sending and another for receiving Ethernet frames. Each thread executes only once per quantum, limiting the model to processing a single Ethernet frame per quantum and thereby capping the maximum achievable throughput. This design choice appears to stem from a failure to adequately account for TD

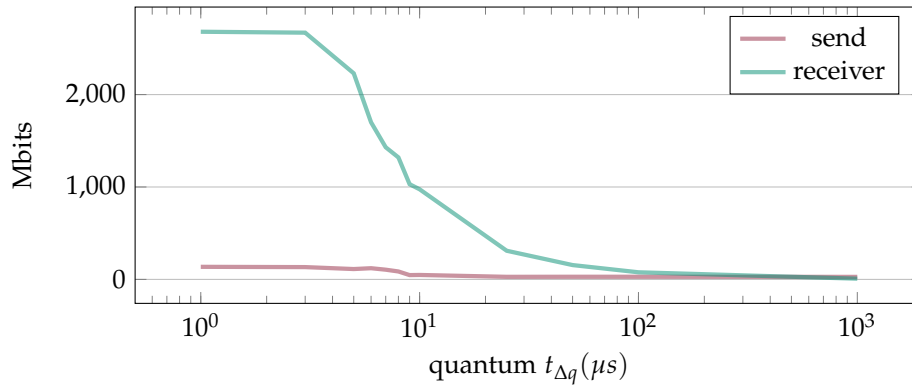


Figure 4.14: Throughput of the iperf3 benchmark as a function of the quantum.

during the device’s implementation. A more appropriate approach would involve calculating the number of packets to process per quantum based on the elapsed time. Since the respective threads are still activated once per quantum, there would be no performance loss. As with the gem5 timer issue, this example underscores the importance of designing virtual peripherals with TD awareness to meet the demands of modern VPs.

4.5 Conclusion & Outlook

This chapter showed how analytical models can be used to estimate the performance and accuracy of parallel and sequential TD simulations. Using these models, TD simulations can be tuned to attain high performance while still keeping TD-induced inaccuracy low. Rather than improving the model’s accuracy by adding complexity, this chapter focused on capturing the gist of TD in a few simple formulas. Nevertheless, as verified with a par-gem5 and SystemC case study, the model can estimate speedup and inaccuracy sufficiently. To conclude, the proposed model helps to answer one of the most important questions with TD: Which quantum should I choose? The general rule of thumb that larger quanta lead to greater speedups while the accuracy decreases could be confirmed by the model and measurements. However, the analytical model allows to quantify this statement further. The speedup of the simulation as a function of the quantum can be described by a saturating sigmoidal curve. The simulation inaccuracy, on the other hand, grows linearly and indefinitely. This means, above a certain point, increasing the quantum leads to an increased inaccuracy while yielding virtually no speedup. Besides analyzing the relation between and speedup inaccuracy, this chapter also investigated how TD changes the semantics of a simulation. As described in Section 4.4.3, TD can result in a plethora of different phenomena. From exploding kernel timestamps to decreasing network throughput; TD with high quanta leads to extreme situations for both the simulator as well as the target software. These situations can expose several bugs or weaknesses that would be hard to uncover without TD.

Chapter 5

Floating Point Simulation

For several decades now, hardware-accelerated Floating Point (FP) arithmetic is a staple of general-purpose CPUs. But today's almost seamless interaction between FP hardware and software was by no means given from the beginning.

Until the 1970s, anarchy among individual FP implementations was widespread, posing immense hurdles to the development of reliable and portable FP software. Because new architectures and implementations continued emerging, a standardization committee under the auspices of the IEEE was created in the mid-1970s [110]. In 1985, this concluded in the publication of the IEEE 754 [94] standard in its first version.

This standard is followed by today's prevalent ISAs, such as ARM, RISC-V, or x64¹. Yet, two revisions of the IEEE 754 standard in 2008 [95] and 2019 [96], and a number of other factors have led to significant disparities, especially between older and newer ISAs, such as x64 and RISC-V.

For example, the maximum of signaling Not a Number (sNaN) and a numeric FP value (e.g., $\max(\text{sNaN}, 5.0)$) yields different results on different platforms. On x64 the result is sNaN, on RISC-V F 2.0 it is quiet Not a Number (qNaN), and on the more recent RISC-V F 2.2 the result is 5.0. Despite different results, all implementations comply with IEEE 754.

This different implementation of seemingly identical instructions poses special challenges for DBT-based cross-platform simulation. DBT simulators achieve near-native execution speeds by mapping target instructions 1-to-1 (or 1-to-few in the worst case) on host instructions. If this is applied to FP instructions, the approach is referred to as *hard float*. But as shown with the maximum example before, different instruction semantics impede a hard float approach. Depending on the ISA, the situation is further complicated by lack of rounding modes, NaN boxing, NaN propagation, FP exceptions flags/traps, and rounding before or after tininess detection.

The impracticability of hard float led to the development of a portable and less constrained alternative named *soft float*. With soft float, FP instructions are modeled purely by integer and bit arithmetic. Typically, functionality is provided by function calls as part of a software library written in a high-level programming language, such as C. This facilitates full control over the implementation, but using a function call including tens if not hundreds of integer instructions to model one FP instruction leaves its mark on the simulator's performance. As shown later in Section 5.6.1, a soft float instruction can be 80× slower than a corresponding hard float instruction. Yet, the ease of use and availability of open-source libraries make it the most popular approach for simulation of FP arithmetic. Examples of simulators incorporating soft

¹ Also referred to as x86-64, AMD64, or Intel 64.

float libraries include Bochs [132, 119], gem5 [20, 123], Spike [159], Uni Bremen RISC-V VP [84, 85], QEMU [17] pre-v4.0.0, and Whisper [40].

One of the goals of this work is to propose a new approach combining the performance of hard float with the flexibility of soft float. Accordingly, this approach is referred to as *floppy float*. The general idea is to leverage the host’s Floating Point Unit (FPU) whenever possible, and handle corner cases in software. Opposed to other works, this thesis covers a large spectrum of FP instructions, and presents new methods for the simulation of Fused Multiply-Add (FMA), division, and square root instructions. Much more, the proposed concepts are also extended to FP vector instructions.

To assess the practical benefits, the proposed methods were integrated into the RISC-V-on-x64 simulator *SIM-V* [124]. Using standard scalar FP benchmarks, floppy float achieves a performance speedup of up to $3\times$ compared to soft float. For non-default rounding modes, the new methods outperform state-of-the-art simulators, like QEMU, by $1.44\times$ in terms of execution time. As part of FP vector instructions, the new method can speed up individual instructions by up to $4\times$. While the case study revolves around RISC-V-on-x64 simulation, the presented methods are applicable to other ISAs as well. In fact, the methods of this chapter were incorporated into a stand-alone library, called *FloppyFloat*, which is able to mimic x64, ARM64, and RISC-V FP arithmetic on virtually all relevant host ISAs (x64, ARM64, RISC-V, PowerPC, MIPS, etc.). The code is available as open-source under <https://github.com/not-chicken/FloppyFloat>.

The rest of this chapter is organized as follows: Section 5.1 provides background information about FP arithmetic in general, and the corresponding implementation in RISC-V and x64. The main challenges of RISC-V-on-x64 FP simulation are described in Section 5.2. This is followed by extensive analysis of FP simulation in other simulators and academic works in Section 5.3. The impact of FP simulation accuracy on the simulator’s functionality is evaluated in Section 5.4. Section 5.5 presents how the floppy float approach overcomes the aforementioned challenges, and how the approach of this thesis improves on existing methods. The performance assessment of the *SIM-V* case study is illustrated and discussed in Section 5.6.

In general, this chapter is an extended version of the author’s following works: *Efficient RISC-V-on-x64 Simulation* [209], *FloppyFloat: An Open Source Floating Point Library for Instruction Set Simulators* [211], *Evaluation of the RISC-V Floating Point Extensions* [206], *Ultra-Fast RISC-V Floating Point Vector Simulation* [210].

5.1 Background

This section provides background information on IEEE 754 Floating Point (FP) arithmetic as well as its implementation in the RISC-V and x64 ISA.

5.1.1 IEEE 754 Floating Point Arithmetic

FP numbers serve as the most prevalent approximation for real numbers in computers. They are featured across various programming languages, denoted as `float` and `double` in C/C++, or `f32` and `f64` in Rust. Given the complex nature of FP arithmetic implementations, adherence to standards is paramount to circumvent fragmentation and software incompatibilities. As a result, the IEEE 754 standard [94] was created in 1985. This standard is followed by most software and hardware FP implementations. The most important number formats defined by IEEE 754 are *binary32* and *binary64*, which are known from C/C++ as `float` and `double`. Using these formats, a FP number is described by a sign, a significand, an exponent, and an implicit bias with the bit representation as depicted in Figure 5.1. The numerical value of an FP number is given by:

$$f = (-1)^{sign} \cdot (1.s_{p-1}s_{p-2}\dots s_1)_2 \cdot 2^{exponent-bias} \quad (5.1)$$

In the formula, s_i refers to the bit at position i in the significand. While most numbers in practice are described by Equation 5.1, the IEEE 754 standard reserves certain bit patterns for exceptional values.

The first case exceptional case is *subnormal numbers*. Whenever $exponent = 0$, the implicit leading 1 is replaced by an implicit leading 0:

$$f = (-1)^{sign} \cdot (0.s_{p-1}s_{p-2}\dots s_1)_2 \cdot 2^{-bias} \quad (5.2)$$

While subnormal numbers facilitate so-called gradual underflows, they are the most controversial aspect of IEEE 754. Implementing subnormal numbers in hardware is considered the most challenging part of FPU design [170]. Moreover, they complicate mathematical proofs and potentially introduce security vulnerabilities through timing side channels [7].

Another special value is infinity. If all bits in the exponent are set and the significand is 0, the value is interpreted as $\pm\infty$.

The last special value is Not a Number (NaN), which is subdivided into quiet Not a Number (qNaN) and signaling Not a Number (sNaN). qNaN is used to represent

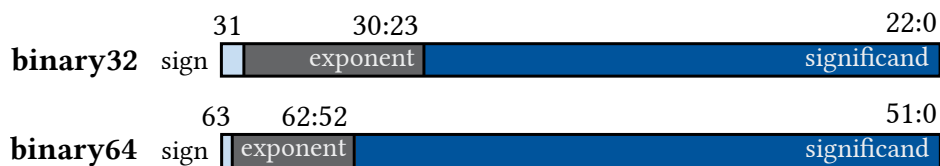


Figure 5.1: Bit representation of IEEE 754 binary32 and binary64.

Table 5.1: Constraints for Equation 5.3

data type	exponent range	precision bits	significand range
binary32	$e_{f,min} = -126 \leq e_f \leq 127 = e_{f,max}$	$p_f = 24$	$ M_f \leq 2^{24} - 1$
binary64	$e_{d,min} = -1022 \leq e_d \leq 1023 = e_{d,max}$	$p_d = 53$	$ M_d \leq 2^{53} - 1$

non-meaningful results (e.g., $\infty - \infty$), while sNaN is intended to be used for uninitialized memory / variables. The bit pattern of a NaN is an exponent with all bits set and a non-zero significand.

While Equation 5.1 is frequently used in literature to introduce the concept of IEEE 754 FP numbers, mathematical proofs are complicated by the $p - 1$ significand bits with an implicit leading 1. Hence, many proofs involving IEEE 754 FP numbers employ Equation 5.3:

$$f = M \cdot 2^{e-p+1}, \quad e = \text{exponent} - \text{bias} \quad (5.3)$$

The idea of Equation 5.3 is to shift the significand by $p - 1$ places, turning it into an integer value. Due to the finite number of bits in binary32 and binary64, the precision p , the significand M , and the exponent e are constrained by the values given in Table 5.1. Whereby the p precision bits include the implicit leading 1. For example, a binary32 value has a precision of 24 bits of which 23 bits are explicitly stored. Hence, the formula is not suitable for the representation of subnormal numbers.

Another important aspect of FP numbers are *rounding errors*. Whenever mathematical operations, such as additions or multiplications, are performed on FP numbers, rounding errors may occur. In this work and other literature, rounding is denoted by the $\circ$ operator. While rounding errors are practically inevitable, *rounding modes* allow to control the sign of the error. The most recent IEEE 754 standard from 2019 [96] defines five rounding modes:

1. roundTiesToEven (mandatory)
2. roundTiesToAway (introduced in 2008, not mandatory)
3. roundTowardPositive (mandatory)
4. roundTowardNegative (mandatory)
5. roundTowardZero (mandatory)

To indicate which rounding mode is used in mathematical representations, an acronym is appended to the $\circ$ operator. For example, $\circ_{RNE32}(a + b)$ corresponds to a 32-bit addition under Round to Nearest, Ties to Even (RNE) rounding mode. The acronyms used in this work follow the RISC-V standard [160]. In the following, if no rounding mode is given, RNE shall be assumed. This is also the default rounding mode of most programming languages. The numerical impact of rounding errors can be assessed by the *standard error model of FP arithmetic* [86]:

$$z = (a \circ b) \cdot (1 + \epsilon) + \eta = \circ(a \circ b) \quad (5.4)$$

$$\epsilon \eta = 0, \quad |\epsilon| \leq \mathbf{u}, \quad \eta \leq |2^{e_{min}}| \cdot \mathbf{u}, \quad \mathbf{u} = 2^{-p}$$

This model can be applied to many arithmetic operations ($+$, $-$, $/$, $\cdot$, $\sqrt{}$) and even accounts for subnormal numbers. For normal numbers, $\eta = 0$ holds true, while for subnormal $\epsilon = 0$ holds true. The relative error ϵ is bounded by the *unit roundoff* error $\mathbf{u}$, which is also known as *machine epsilon*. To account for non-default rounding modes, the roundoff error can be approximated with $2\mathbf{u}$.

To indicate irregularities during a FP operation, the IEEE 754 standard defines five FP exception flags/exceptions. An exception usually sets a corresponding status flag and may entail a trap. Currently, the following flags/exceptions are mandated:

- Invalid: Signaled when the executed instruction was invalid. Example: $\infty - \infty = qNaN$.
- Underflow: Signaled when the obtained result is subnormal. Note that the detection of an underflow may either happen before or after rounding. In both cases, the detection happens before the result is being normalized. Hence, normal results can lead to the underflow flag being set (see Code A.1 in the appendix for an example). The exception only sets the flag if the result is inexact. 32-bit example: $(1.5046328E - 36)^2 = 0$.
- Overflow: Signaled if the rounded result exceeds the format's largest finite value assuming an unbounded exponent range. 32-bit example: $(1.5845633E29)^2 = \infty$.
- Inexact: Signaled if the rounded result differs from the computed result. 32-bit example: $0.00390625 + 65536 = 65536$.
- Division-by-zero: Only relevant for division. Signaled if the divisor is zero and the dividend is a finite, non-zero number. Example $1 / +0 = \infty$.

5.1.2 RISC-V Floating Point

This section provides an overview of how RISC-V implements IEEE 754. All of the following information is based on the 2019 RISC-V ISA manual [160].

To cover a wide range of possible use cases, RISC-V divides its instruction set into so-called *extensions*. Whereby each extension defines a certain set of instructions, registers, and other characteristics. This includes FP arithmetic, which is used in the extensions F, D, Q, V, Zfa, Zfh, Zfhmin, Zfinx, Zhinx, and Zhinxmin. IEEE 754-compliant 32-bit and 64-bit FP arithmetic is provided by the F and D extensions respectively.

All FP extensions are mostly compliant with the latest IEEE 754 standard from 2019 [96]. Consequently, there are five FP exception flags and five rounding modes, which are both represented in the fcsr register (see Figure 5.2). Opposed to other ISAs, FP exceptions do not trigger traps in RISC-V. As explained by A. Waterman [191], the rationale behind this decision was to facilitate the implementation of non-speculative out-of-order execution.

Another unique characteristic of RISC-V is its two-fold way to set rounding modes. The first way is to statically specify the rounding mode in the instruction's encoding.

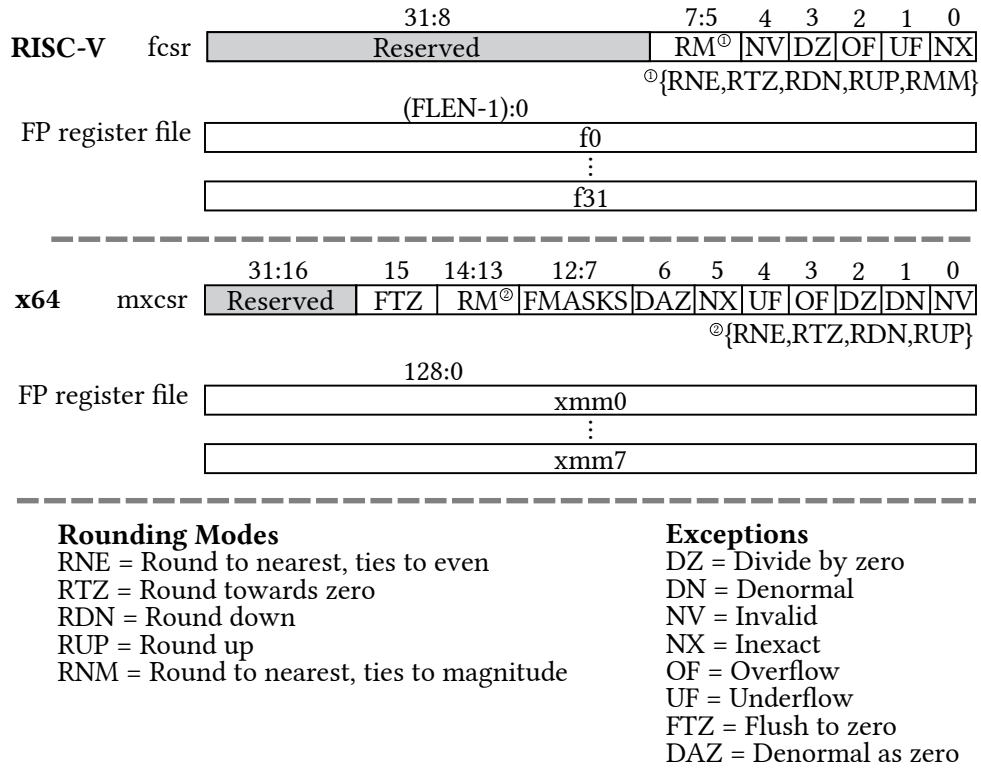


Figure 5.2: Floating point control and status register of x64 and RISC-V.

Alternatively, the instruction’s rounding can encode to “dynamic”. In this case, the dynamic rounding mode from the fcsr register is used. This design was inspired by FP in programming languages [191]. While arithmetic FP instructions often use a dynamic rounding mode (e.g., `std::fesetround` in C++), FP conversion operators, such as typecasts, use a static rounding.

One peculiarity not mentioned in the IEEE standard is RISC-V’s hardware-assisted NaN boxing. It was introduced to rectify a flaw in the RISC-V specification, which was discovered in 2017 [192]. At that time, storing 32-bit FP values residing in 64-bit registers did not have a defined memory layout. The standard only guaranteed that loading the value from the memory reinstantiates the intended value. These undefined memory layouts could create complications in various situations, as highlighted by A. Bradburry in his RFC [25]. For instance, when migrating tasks on a heterogeneous MPSoC, each core might interpret the dump of the FP register file differently. To address this issue, A. Bradburry proposed multiple solutions, which were subsequently discussed within the RISC-V ISA-Dev group [24]. As a result, a unique NaN boxing scheme was chosen. NaN boxing saturates the upper bits if an N-bit FP value is written to an M-bit FP register with $M > N$. For instance, if a 32-bit FP value is stored in a 64-bit register, hardware only considers it valid if the upper 32 bits are set to 1. This means instructions working on 32-bit FP values must check the upper bits when reading the register and set them when writing back the result. As the 64-bit value encodes to a negative qNaN due to its saturated upper bits, hence the name NaN boxing, valid values cannot be created by accident.

While IEEE 754 defines a whole range of patterns for qNaN, it does not specify the so-called *canonical qNaN*. A canonical qNaN is the bit pattern generated by instructions with invalid operands. On RISC-V, the 32-bit canonical qNaN is encoded as 0x7fc00000 for 32-bit registers. The same 32-bit value for 64-bit FP registers results in a NaN-boxed value of 0xffffffff7fc00000.

A feature recommended but not mandated by IEEE 754 is NaN propagation. NaN propagation allows to convey diagnostic information through FP instructions as part of the NaN payload. Due to its small benefits, low popularity, and considerable hardware costs, RISC-V does not use NaN propagation [191].

5.1.3 x64 Floating Point

In this subsection, the details of the x64 FP arithmetic are explained in greater detail, with a focus on the Streaming SIMD Extensions (SSE). The x87 FP extension from 1980 is still supported by modern x64 CPUs, yet it mostly serves the purpose of retaining compatibility with legacy software. Modern compilers, such as gcc or clang, generate SSE instructions by default. In a static code analysis conducted by Akshintala et al. in 2019 [3], SSE instructions appeared $150\times$ more often in Ubuntu 16.05 software packages than x87 instructions.

The SSE extension mostly follows the IEEE 754 standard [94] from 1985, but extends it with non-standard behavior. The following highlights how this behavior and other subtleties compare to RISC-V and the most recent IEEE 754 2019 [96] standard.

A modern IEEE 754 feature, which is absent in x64, is the Round to Maximum Magnitude (RMM) rounding mode. It was introduced and declared mandatory with IEEE 754 2008 [95]. The five FP exceptions (invalid, underflow, overflow, inexact, divide-by-zero) were already defined in IEEE 754 1985, matching RISC-V in this regard. Additionally, x64 defines a non-standardized denormal flag for the detection of subnormal results.

On x64 systems, subnormal results and inputs can be flushed to zero by means of the FTZ and DAZ flags (see Figure 5.2). These modes were primarily designed for 3D applications where performance is a greater concern than accuracy [177]. Because with many microarchitecture implementations, processing subnormal numbers can decrease FP performance by an order of magnitude or more [53]. But if subnormal numbers are flushed to 0, there is no risk of hampering performance.

In contrast to RISC-V, the x64 ISA also allows to control which FP exceptions trigger a trap. The corresponding masking bits are labeled as FMASK in Figure 5.2.

As mentioned in the previous subsection, IEEE 754 does not define a fixed bit pattern for canonical qNaN values. For unknown reasons, x64 uses the same qNaN as RISC-V, but with a negative sign. Hence, a 32-bit qNaN value as a result of an invalid operation is encoded as 0xffc00000.

5.2.2 Instruction Semantics

As already mentioned in Section 5, seemingly identical FP instructions may have different semantics depending on the ISA. This is due to different versions of IEEE 754, as well as implementation-defined behavior. For example, the former reason applies to maximum and minimum instructions. Here, every IEEE 754 version used a different definition, which led to an interesting fragmentation among ISAs:

- x64 (IEEE 754 1985):
 - `maxss 5.f, sNaN = sNaN`
 - `maxss sNaN, 5.f = 5.f`
- RISC-V F/D < v2.2 (IEEE 754 2008):
 - `fmax 5.f, sNaN = qNaN`
 - `fmax sNaN, 5.f = qNaN`
- RISC-V F/D v2.2 (IEEE 754 2019):
 - `fmax 5.f, sNaN = 5.f`
 - `fmax sNaN, 5.f = 5.f`

These different versions are basically a result of missing or erroneous definitions. For example, when x64 FP arithmetic was introduced by Intel in 1999 as part of the SSE extension, the current IEEE 754 standard was still from 1985. However, this standard did not define maximum or minimum instructions. There is no publicly available information on Intel's instruction design, but it seems likely that the engineers resorted to a definition by order relations. For example, using order relations, a C code implementation of the maximum instruction would look as follows: `a > b ? a : b`; . According to IEEE 754 1985 [94], any comparison with NaN is treated as unordered, requiring false to be returned. Thus, `sNaN > 5.f` yields false, as well as `5.f > sNaN`. Ultimately, a maximum/minimum instruction defined by the aforementioned code will always return the second operand (b) if one input is NaN, which is exactly the behavior exhibited by the SSE instruction.

The issue of missing maximum/minimum instructions was finally addressed with IEEE 754 2008 [95]. This standard was also adopted by RISC-V for the `fmax/fmin` instructions, and kept until FP extensions version 2.2. According to the standard, maximum/minimum should return a `qNaN` when one of the operands is a `sNaN`. If only one of the operands is a `qNaN`, the number must be returned. However, following this definition leads to problems with the instruction's associativity. The following example shows how a different association of maximum operations leads to different results:

$$\begin{aligned} \max(6.f, \max(5.f, sNaN)) &= \max(6.f, qNaN) = 6.f \\ \max(\max(6.f, 5.f), sNaN) &= \max(6.f, sNaN) = qNaN \end{aligned} \tag{5.5}$$

David G. Hough later confirmed in his article *The IEEE Standard 754: One for the History Books* [89] that the aspect of flawed associativity was simply overseen. Hence, the problem was rectified with IEEE 754 2019 [96], which is followed by the present-day RISC-V ISA.

Besides rectifications in the standard, some instructions simply differ due to being implementation-defined. This concerns the FMA instruction and float-to-integer conversions. Code A.2 in the appendix provides an example program, which yields different results depending on the executing ISA.

Moreover, some instructions may not even be available on some systems. For instance, the `fclass` instruction of RISC-V, which is used to classify a FP into different categories (i.e., normal, subnormal, etc.), is not available on x64. Consequently, simulating `fclass` on x64 has to be achieved by means of other instructions.

5.2.3 Rounding Modes

As previously mentioned in Section 5.1, x64 misses the rounding mode “roundTiesToAway”, which was introduced in the IEEE 754 2008 [95] standard. Hence, using this rounding for RISC-V-on-x64 simulation poses a challenge. Yet, this is a corner case, since most applications and programming languages use RNE as the default rounding mode.

5.2.4 NaN Boxing

Since NaN boxing is a non-IEEE feature that is unique to RISC-V, simulating it poses a significant challenge. Because if NaN boxing would be ignored, FP instructions may yield wrong results.

An example of such a situation is depicted in Figure 5.3. In the given example, a register file holds two FP values in two 64-bit registers. If only the Least Significant Bitss (LSBs) are interpreted as 32-bit FP values, `r1` holds 2.0, while `r2` holds 1.0. When

Register	Value
r1	$\overbrace{0x0000000040000000}^{=2.0}$ incorrect NaN box
r2	$\overbrace{0xfffffffff3f800000}^{=1.0}$ correct NaN box

	32-bit add : $r1 = r1 + r2$	32-bit add : $r2 = r2 + r2$
RISC-V	$r1 = 0xfffffffff7c00000$ $\overbrace{\hspace{1.5cm}}^{qNaN}$	$r2 = 0xfffffffff4000000$ $\overbrace{\hspace{1.5cm}}^{=2.0}$
ARM64	$r1 = 0x0000000040400000$ $\overbrace{\hspace{1.5cm}}^{=3.0}$	$r2 = 0xfffffffff4000000$ $\overbrace{\hspace{1.5cm}}^{=2.0}$

Figure 5.3: Comparing RISC-V NaN boxing to ARM64.

adding values as 32-bit, e.g., $r1 = r1 + r2$, some ISAs, such as ARM64 or x64, ignore the upper bits of the 64-bit registers for reading and writing. Hence, $r1$ is interpreted as 2.0, while $r2$ is interpreted as 1.0. Furthermore, the zeros in the upper half of $r1$ and the ones in the upper half of $r2$ are retained when performing a 32-bit write into the registers.

With RISC-V NaN boxing, the operands are first checked for correct NaN boxes. That means, even though the instruction uses 32-bit operands, all bits of the register are checked for a correct value. Since $r1$ does not have a valid NaN box (all 1 in the non-used part), $r1$ is interpreted as qNaN, which subsequently leads to qNaN as a result for the addition. Moreover, the 32-bit write into the register sets all upper bits to 1 in order to create a valid NaN box. A test program showcasing the effects of NaN boxing is given in Code A.3.

5.2.5 NaN Propagation

A feature recommended, but not mandated by IEEE 754, is NaN propagation. The idea is to propagate NaN payloads through instructions to retain diagnostic information. It is supported by both x64 and ARM. The RISC-V developers decided against strictly mandating it due to the additional hardware cost and its low popularity [191]. Figure 5.4 highlights how NaN propagation leads to different results on x64 and RISC-V. In the example, a qNaN with a payload of 0x123 is used as an input operand for a FP addition. On x64 systems, the payload is included in the output NaN, while RISC-V omits any information and just returns the canonical qNaN. Note that in the presence of two qNaNs, x64 always returns the former NaN. This is in accordance with the recommendations of IEEE standard, but has been criticized due to flawing the commutativity of many operations [66]. A C program of NaN propagation in practice is shown in Code A.4 in the appendix.

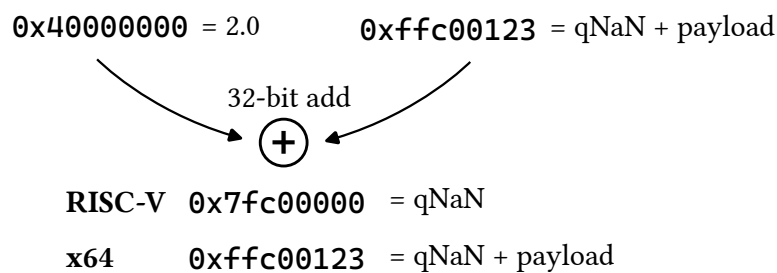


Figure 5.4: Comparing NaN propagation on RISC-V and x64.

5.2.6 Exception Flags

As already explained in the background section, IEEE 754 mandates five so-called FP exception flags. The first IEEE 754 standard already defined these flags, which leads to RISC-V being identical to x64 in that regard. Consequently, the simplest solution would be to load/store the target's FP exception flags before/after an instruction. More specifically, this *FPU guard* approach involves the following steps: Save host

```

1  int __fegetenv (fenv_t *envp) {
2      // x87 state
3      __asm__ ("fstenv_0\n" : "=m" (*envp));
4      __asm__ ("fldenv_0\n" : "=m" (*envp));
5
6      // SSE state
7      __asm__ ("stmcsr_0\n" : "=m" (envp->__mxcsr));
8
9      return 0;
10 }

```

Code 5.1: Simplified version of `fegetenv` as implemented in `glibc/sysdeps/x86_64/fpu/fegetenv.c`.

FPU state, load target FPU state, execute target FP instruction(s), save target FPU state, load host FPU state. In fact, this naïve method was also considered in the works of Cota et al. [48], Guo et al. [76], Sarrazin et al. [165], and You et al. [199]. However, in three of the works, the approach was ultimately discarded due to its poor performance. According to Guo et al. [76], the FPU guard approach is even outperformed by a soft float implementation. Also, the experiments of this thesis confirmed that more than 2-4 MIPS are hardly achievable.

To better understand the performance bottleneck of this seemingly trivial approach, a deeper analysis of the underlying code was conducted. When using `glibc`, the FPU state can be read by a function called `fegetenv`. Note that a similar function also exists for writing the FPU state (`fesetenv`). The simplified code of `fegetenv` is given in Code 5.1.

As can be seen in the code, `fegetenv` only comprises three instructions. Two of them are responsible for the x87 part, while only one is needed to fetch the `mxcsr` register of the SSE extension. In a profiling run on an AMD Threadripper 3990X, the x87 part took about 90% of the total execution time of the function. Similar results

μArch	Latency (cycles)		Throughput	
	LDMXCSR	STMXCSR	LDMXCSR	STMXCSR
AMD Zen2	-	-	1/17	1/16
AMD Zen3	13	13	1/20	1/15
AMD Zen4	13	13	1/21	1/15
Intel Coffee Lake	5	4	1/3	1
Intel Cannon Lake	5	4	1/3	1
Intel Ice Lake	6	5	1/3	1

Table 5.3: Latencies and throughput of instructions for loading and storing the `mxcsr` register according to [65]. The throughput is the maximum number of instructions of the same kind that can be executed per clock cycle assuming that operands are independent of preceding instructions. The latency of an instruction is the delay of an instruction in a dependency chain.

were also obtained for writing the FPU state. This is an overproportionate share, considering that x87 is an obsolete extension for which compilers, with a few exceptions, no longer generate code. So, a second profiling run was conducted with the x87 part excluded. But even with this modification, the attainable performance was in the order of a few MIPS. This boils down any possible performance bottleneck to two instructions: `STMXCSR` for reading the FPU state, and `LDMXCSR` for writing it. A. Fog [65] conducted an exhaustive study of the performance of individual x64 instructions on different microarchitectures. This includes `STMXCSR` and `LDMXCSR` which are shown in Table 5.3.

As can be seen in the table, executing these instructions is relatively costly, especially on the AMD Zen microarchitecture. This microarchitecture is implemented in the AMD Threadripper 3990X, which was used to conduct the performance analysis in this work. Ultimately, choosing an approach that works fast on all microarchitectures is desirable, which is why other works, including this thesis, focus on other methods.

5.2.7 Other Challenges

While many of the aforementioned challenges also apply to other target-on-host combinations, this subsection briefly mentions rather niche challenges of other ISAs. Since many systems follow IEEE 754, the supported FP formats usually boil down to binary32 (a.k.a float) and binary64 (a.k.a. double). One more recent systems, this also includes binary16. Yet some ISA offer their own, non-standard-compliant formats, which may be hard to simulate if the formats are not supported by the host. For example, the x87 extension of x86 supports 80-bit FP numbers. Also, the PlayStation2's FP format is rather exotic as it does not support NaN or infinite values. This simplifies FPU hardware design at the cost of compatibility and ease of simulation. Another FP format which emerged in recent years is the posit format [77]. Compared to IEEE 754 binary formats, it promises a higher accuracy and a higher range of representable values.

As already mentioned in Subsection 5.1.3, some ISAs, such as x64, allow to trap on FP exceptions. That means, whenever a FP exception is raised, a corresponding trap handler can be executed. If traps have to be simulated on non-trapping systems, such as RISC-V, additional modeling effort is required.

Furthermore, the IEEE 754 standard allows for tininess detection after and before rounding. RISC-V as well as x64 detect tininess after rounding, while ARM64 detects it before rounding. An overview of ISAs and their differences is given in Table 5.4.

Table 5.4: Differences of FP-related characteristics among ISAs.

ISA	32-bit qNaN	Class	Exception Flags	FTZ	NaN Boxing	NaN Propagation	Rounding Modes	Tininess	$fma(0, \infty, qNaN)$
ARM [11]	0x7fc00000	-	✓	✓	-	✓ ⁸	4	BR ⁹	invalid ¹⁰
LoongArch [122]	0x7fc00000	FCLASS	✓	-	-	✓	4	AR	undefined
MIPS64 [135]	0x7fc00000 ¹	CLASS	✓	✓	-	✓	4	AR	undefined
OpenRISC 1000 [146]	undefined	CLASS ³	✓	-	✓	-	4	BR	undefined
PowerPC [93]	0x7fc00000	FPSCR_FPRF ⁴	✓ ⁵	✓ ⁷	-	✓	4	BR	invalid
RISC-V [160]	0x7fc00000 ²	FCLASS	✓	-	✓	-	5	AR	invalid
x64 SSE [99]	0xffc00000	-	✓ ⁶	✓	-	✓	4	AR	valid

AR = After Rounding, BR = Before Rounding

- 1: With $FCSR_{NAN2008}=0$ the canonical qNaN can be changed to 0x7fbfffff.
- 2: The canonical qNaN of RISC-V was 0x7fffffff before version 2.1 of F/D.
- 3: The occurrence of sNaN and qNaN values can also be detected by reading FPSCR.
- 4: The $FPSCR_{FPRF}$ register contains a classification of the result after a FP instruction.
- 5: PowerPC has additional non-sticky exception flags in the FPSCR register.
- 6: x64 SSE has an additional denormal operand exception.
- 7: Only enabled if the NI flag in FPSCR is set.
- 8: NaN propagation can be toggling a bit in FPCR.DN.
- 9: If FEAT_AFP is supported, FPCR.AH allows to change tininess detection to after rounding.
- 10: IF FEAT_AFP is supported, FPCR.AH allows to change the validity of the instruction.

5.3 Related Work & Other Simulators

In this section, the FP implementation of popular simulators is analyzed. This is complemented by an extensive analysis of academic works in this field.

5.3.1 Soft Float

Many open-source simulators, such as gem5 [20, 123], QEMU [17] pre-v4.0.0, Spike [159], UniBremem RISC-V VP [84, 85], Bochs [119, 132], and Whisper [40], employ a method called *soft float* to simulate FP arithmetic.

The fundamental concept behind the soft float approach involves utilizing integer arithmetic and logical operations to replicate arbitrary FP calculations. The availability of diverse open-source libraries and the ease of use make it the most popular FP simulation approach. For example, all simulators listed above use the open-source library *Berkeley Softfloat* by J. Hauser [83], which is based on the IEEE 754 1985 standard [94]. Soft float libraries that implement the more recent IEEE 754 2008 [94] standard include *SoftFP* by F. Bellard [18] and *FLIP* by C.-P. Jeannerod et al. [19]. Additionally, specialized soft float libraries optimized for specific processor architectures are available. For instance, *RVfplib* [154] is tailored for RISC-V systems that lack the F or D extension.

While soft float is often favored for its versatility and simplicity, its major drawback is the significant computational overhead. Using tens or hundreds of integer instructions to simulate one FP instruction can potentially reduce the performance by that same factor. Exact slowdown factors are provided later in Section 5.6.

Comprehensive background knowledge on implementing soft float instructions is provided in the *Handbook of Floating Point Arithmetic* by Muller et al. [141].

5.3.2 rv8

rv8 [44, 45] is a DBT-based, open-source RISC-V-on-x64 simulator. With *rv8*, the RISC-V target exception flags and rounding mode are mapped 1-to-1 to the x64 host. As explained in Subsection 5.2.6, RISC-V and x64 are equivalent with respect to their exception flags. Thus, checking and setting the exception flags is simply achieved by accessing the host's `mxcsr` register. Yet mapping the rounding modes poses a problem, as x64 is not endowed with a mode for RMM. The solution of *rv8* for this problem can be found in Code 5.2 in Line 9.

In the function `fenv_setrm(int rm)`, the RISC-V rounding mode is loaded into the host FPU. The missing rounding mode RMM of x64 is incorrectly mapped to RNE. This trades accuracy for simplicity/performance and leads to *rv8* not being compliant with the official RISC-V standard.

Other problems, such as NaN boxing or semantically different instructions, are solved by rectifications in software. Furthermore, FP instructions are not directly translated, but use an interpreter. This interpreter falls back to standard C++ opera-

```

1  inline void fenv_setrm(int rm) {
2      int x86_mxcsr_val = __builtin_ia32_stmxcsr();
3      x86_mxcsr_val &= ~x86_mxcsr_RC_RZ;
4      switch (rm) {
5          case rv_rm_rne: x86_mxcsr_val |= x86_mxcsr_RC_RN; break;
6          case rv_rm_rtz: x86_mxcsr_val |= x86_mxcsr_RC_RZ; break;
7          case rv_rm_rdn: x86_mxcsr_val |= x86_mxcsr_RC_DN; break;
8          case rv_rm_rup: x86_mxcsr_val |= x86_mxcsr_RC_UP; break;
9          case rv_rm_rmm: x86_mxcsr_val |= x86_mxcsr_RC_RN; break;
10     }
11     __builtin_ia32_ldmxcsr(x86_mxcsr_val);
12 }
13
14 P::ux exec_inst_rv32(T &dec, P &proc, P::ux pc_offset) {
15     // ...
16     switch (dec.op) {
17         case rv_op_fadd_s:
18             if (rvf) {
19                 fenv_setrm((fcsr >> 5) & 0b111);
20                 freg[dec.rd] = freg[dec.rs1] + freg[dec.rs2];
21             }
22             break;
23         case rv_op_fmax_s:
24             if (rvf) {
25                 freg[dec.rd] = (freg[dec.rs1] > freg[dec.rs2])
26                             || isnan(freg[dec.rs2])
27                             ? freg[dec.rs1] : freg[dec.rs2];
28             }
29             break;
30         // ...
31     }
32 }

```

Code 5.2: Rounding modes in rv8.

tors to implement RISC-V instructions. For example, Code 5.2 shows the implementation of the FADD.S and FMAX.S instructions.

5.3.3 QEMU post-v4.0.0

In an attempt to increase the simulator's FP performance, QEMU switched from soft float to the method of Guo et al. [76] in version 4.0.0. The initial idea of Guo et al. was to calculate the result of a FP instruction on the host FPU, while the FP exception flags are simulated in software. However, their approach exhibited a significant overhead of the calculation of the inexact exception, leading to no speedup compared to soft float. They only could find a fast solution for additions, which is explained in greater detail in Subsection 5.3.5. To avoid the high performance penalties for the inexact calculation, Guo et al. introduced a quick check before each FP operation,

whether the exception flag must be calculated at all, because with most ISAs, like x64 or RISC-V, the inexact exception flag is sticky. That means, once the inexact flag was set, it remains in that state even if subsequent instructions generate exact results. In practice, most applications do not clear the inexact exception flag and tend to frequently calculate inexact results. Thus, it can be concluded that in the majority of the operations, the inexact exception flag is already set, and that no expensive recalculation is needed. An example of QEMU's square root instruction using the method of Guo et al. is shown in Code 5.3.

The implementation of the square root instruction starts with a call to `can_use_fpu` in Line 13. This corresponds to the aforementioned check whether the calculation of the inexact exception flag can be skipped. Moreover, the host FPU can only be used if the host rounding mode and target rounding mode match. The host rounding mode is assumed to be always the default C rounding mode of RNE. If the macro `QEMU_NO_HARDFLOAT` is defined, the host FPU is never used. This is required for target ISAs, such as PowerPC, which have an additional set of non-sticky FP exception flags. If `can_use_fpu` returns false, the code resorts to a soft float implementation, as can be seen in Line 14.

To also avoid setting the underflow and invalid exception, the soft float fallback is also called if the input is subnormal or negative (see Line 17-18). The idea of checking both underflow and invalid flags was proposed in a work of Cota et al.

```

1  static inline bool can_use_fpu(const float_status *s) {
2      if (QEMU_NO_HARDFLOAT)
3          return false;
4
5      return likely(s->f_excep_flags & f_flag_inexact
6                  && s->f_round_mode == f_round_near_even);
7  }
8
9  float32 float32_sqrt(float32 xa, float_status *s) {
10     union_float32 ua, ur;
11
12     ua.s = xa;
13     if (unlikely(!can_use_fpu(s)))
14         goto soft;
15
16     float32_input_flush1(&ua.s, s);
17     if (unlikely(!is_zero_or_normal(ua.s) || is_neg(ua.s)))
18         goto soft;
19
20     ur.h = sqrtf(ua.h);
21     return ur.s;
22
23     soft: return soft_f32_sqrt(ua.s, s);
24 }

```

Code 5.3: Implementation of a square root instruction in QEMU.

[48], who committed their idea to QEMU in 2018. Note that the contributors seem to have overlooked the fact, that checking for underflows is not necessary as square root instructions can never result in an underflow. Eventually, if all checks were positive, the function `sqrtof` is called in Line 20. This will result in a `sqrtps` FP instruction in case a x64 host system is used.

With the methods of Cota et al. and Guo et al., the performance of simulated FP instructions could be increased by a factor of more than $2\times$ compared to the former soft float implementation. However, this speedup can only be attained if an inexact exception occurs at some point and if the RNE rounding mode is used.

5.3.4 Rosetta 2

In 2020, the company *Apple* transitioned their formerly x64-based PC and laptop segment to ARM64-based silicon. To retain compatibility with x64-based legacy software, the x64-on-ARM64 user-mode simulator *Rosetta 2* was introduced. Since x64-on-ARM64 simulation faces similar FP challenges, a look into an industry giant's solution is of great interest. However, since Apple does not disclose technical characteristics of their products, detailed information about Rosetta 2 is sparse.

According to a blog post of J. Dougall [102], Apple introduced an alternate FP mode to solve the problem of FP simulation in hardware. By setting a certain bit in the ARM FP control register, x64 FP behavior can be mimicked. Hence, there is virtually no overhead for simulating FP instructions. This is also reflected in performance results of Rosetta 2. In a performance study [68] of Rosetta 2 on the Apple M1 processor, the simulator attained near-native speeds ($0.57\times$ - $0.82\times$) for the SPEC 2017 FP benchmark suite.

While this method approaches the theoretical maximum performance, full control of the ISA and the silicon is required. Since the ISA in particular is in the hands of ARM, it is likely that Apple has influenced the development of the architecture. Shortly after Apple's release of the M1 processor [8], the first physical implementation of the alternate FP mode, ARM officially included this mode in the ARMv8 ISA. More specifically, it was introduced with the ARMv8.7 architecture extension from January 2021 [11] and is technically referenced as FEAT_AFP. With FEAT_AFP now being part of the official ARM ISA, the alternate FP mode will likely find its way in the silicon of other manufacturers.

5.3.5 You et al.

As mentioned in Section 5.3.3, Guo et al. [76] attempted to implement software-based calculations for the inexact exception, but only managed to find a fast solution for additions/subtractions. Specifically, the proposed solution looks as follows:

$$inexact = ((a + b) - a) < b \quad (5.6)$$

Despite not being mentioned in the paper, this method resembles the so-called *Fast2Sum* algorithm, which was introduced by T. J. Dekker [51] in 1971. According

```

1 // Fast2Sum for RUP
2 float c = a + b; // Result.
3 float x = fabs(a) > fabs(b) ? a : b;
4 float y = fabs(a) > fabs(b) ? b : a;
5 float r = y - (c - x); // Rounding error.
6 if (r != 0) {
7     inexact = true;
8     if (r > 0) {
9         c = nextup(c); // Next greater FP value.
10        overflow = is_inf(c) ? true : overflow;
11    }
12 }

```

Code 5.4: C++ code of the Fast2Sum algorithm simulating RUP rounding.

to this algorithm, the result of a rounded addition can be described by the sum of its exact value and a residual:

$$\begin{aligned}
 a + b - r &= s = \circ(a + b) \\
 r &= \circ(b - \circ(s - a)) \quad \text{with } |a| > |b|
 \end{aligned}
 \tag{5.7}$$

The residual r holds the exact rounding error of the addition of the variables a and b . Consequently, if the residual r is unequal to 0, the FP addition was inexact. Furthermore, the rounding direction of the preceding addition $\circ(a + b)$ is also determined by the residual r . For values less than 0, the result of the addition was rounded up; for values greater than 0, the result was rounded down. This fact was not used by Guo et al. [76], but by a work of You et al. [199] in 2019. A C++ implementation to simulate a Round Up (RUP) target rounding mode using RNE on the host is given in Code 5.4. While Guo et al. and You et al. show how to perform quick inexact checks with rounding adjustments for additions/subtractions, other arithmetic instructions remain unnoted. They propose an inexact check for FMA instructions based on integer-based intermediate results, but their experiments show no performance advantage over a soft float implementation.

5.3.6 Sarrazin et al.

In 2016 Sarrazin et al. [165] published a work about simulating 32-bit FMA instructions on host systems without hardware support for said instruction. Although the focus is not on determining the inexactness of FMA, Subsection 5.5.5 of this thesis shows how the algorithm can be rearranged to quickly calculate the inexact exception flag. Remarkably, their work was published in 2016, predating the unsuccessful

attempt of You et al. [199] in 2019. The proposed idea of Sarrazin et al. extends the 2Sum algorithm to get the following set of equations:

$$\begin{aligned}
 A &= \circ_{64}(a), B = \circ_{64}(b), C = \circ_{64}(c), d = \circ_{32}(a \cdot b + c) \\
 M &= \circ_{64}(A \cdot B) \\
 S &= \circ_{64}(M + C) \\
 \tilde{d} &= \circ_{32}(S) \\
 R &= 2Sum(M, C) \\
 E &= ||S - \tilde{d}|| \\
 d &= \begin{cases} Round(\tilde{d}, R, S) & \text{if } (E = 2^{e_{\tilde{d}}-p}) \wedge (R \neq 0) \\ \tilde{d} & \text{else} \end{cases} \\
 Round(\tilde{d}, R, S) &= \begin{cases} RoundToMagnitude(\tilde{d}) & \text{if } sgn(S) = sgn(R) \\ RoundToZero(\tilde{d}) & \text{else} \end{cases}
 \end{aligned} \tag{5.8}$$

The algorithm starts by upcasting all 32-bit FP values to 64-bit FP values. Using a 64-bit multiplication, the product of a and b can be exactly calculated ($M = a \cdot b$). Subsection 5.5.2 will later provide a proof for this relation. Next, M and C are added to obtain the result S of a 64-bit FMA with 32-bit operands. This result is then cast to a 32-bit result $\tilde{d}$. At this point $\tilde{d}$ and d only differ by one unit in the last place in the worst case. Statistically speaking, the worst-case scenario is rather unlikely, but if S is exactly between two 32-bit FP values, and if the residual of $\circ_{64}(M + C)$ is non-zero, the casting operation $\circ_{32}(S)$ may have rounded the result into an undesired direction. Hence, the 2Sum algorithm is used to determine the residual R of adding C and M . Note that the output of the 2Sum algorithm is identical to the Fast2Sum algorithm, which was presented in Subsection 5.3.5. A more detailed discussion about the differences and performance implications is provided later in Subsection 5.5.1. Next, it is determined if S lies between two 32-bit FP values. This can be achieved by calculating $E = ||S - \tilde{d}||$, a subtraction that is always exact due to Sterbenz' Lemma [175]², and comparing the result with $2^{e_{\tilde{d}}-p}$. So, if E is equal to $2^{e_{\tilde{d}}-p}$ and if the residual R is non-zero, subsequent rounding has to be applied. More specifically, if the signs of R and S are equal, the approximated result $\tilde{d}$ has to be rounded towards larger magnitudes. Otherwise, the approximated result $\tilde{d}$ result has to be rounded towards zero.

One major disadvantage of the method by Sarrazin et al. is the dependence on larger data types. If the residual of a 32-bit FMA instruction is computed, at least 64-bit FP precision is required. Or more precisely, the larger data type needs at least $2p$ significand bits. Consequently, this algorithm does not work for double precision values on x64 systems. The 80-bit precision provided by x87 FPU cannot be used, as it has less than $2p$ significand bits.

² if $a/2 \leq b \leq 2a$ then $\circ(b - a) = b - a$

5.4 Impact of Accuracy

As can be seen from the previous sections, considerable effort is required to correctly simulate FP arithmetic in all its aspects. Remarkably, the effort required to calculate the actual result of an instruction is almost negligible. The complexity lies much more in the modeling of corner cases and side effects, such as NaNs, FP exception flags, and so forth. However, these secondary factors hardly ever play a role in practice. The question therefore arises as to whether neglecting these parts leads to problems at all.

In fact, an early version of the RISC-V simulator SIM-V [108] neglected all these aspects. As long as the rounding was executed correctly, no problems were ever witnessed in practice. Linux systems can be booted, and all 78 benchmarks listed in Figure A.1 executed flawlessly. At that time, SIM-V was already deployed in industry, and there were also no reports of an erroneous FP implementation.

The only application where problems occurred is the RISC-V Architectural Test Framework (RISCOF) [73]. This test framework by RISC-V International tests multiple architectural features to guarantee a system's RISC-V compatibility. It is important to note that passing the tests is a prerequisite for licensing the RISC-V trademark. Moreover, a system may only be marketed as "RISC-V compatible" if the tests pass. While passing RISCOF is certainly of paramount importance for many manufacturers of RISC-V systems, the artificial test cases bear little practical relevance.

One domain in which inaccurate FP simulation has led to practical issues is game simulation. Here, simulators put a strong focus on performance and therefore often neglect aspects that are deemed unimportant. Coupled with FP-intensive games, which are often customized to the target hardware, the perfect breeding ground for FP bugs is created. For example, when using the GameCube/Wii simulator *Dolphin* on x64, the game *Dragon Ball: Revenge of King Piccolo* requires accurate NaN simulation [64]. As shown previously in Table 5.2, the PowerPC ISA, which is used by the Wii, generates qNaN with a positive sign, while x64 generates a negative qNaN. If the wrong qNaN is generated, two incoming enemies in the aforementioned game fall off the stage, which leads to a subsequent softlock. A visual representation of this bug is depicted in Figure 5.5. Since simulating qNaN correctly would generally impede the



Figure 5.5: Missing enemies due to wrong NaN simulation in *Dragon Ball: Revenge of King Piccolo*.

```

1  float t;
2  float y {-3065.f};
3  ...
4  y -= std::sin(t) * 0.58;
5  t += 0x100;

```

Code 5.5: Possible C++ implementation of the oscillating platform.

simulator's performance, Dolphin has a special mode only for *Dragon Ball: Revenge of King Piccolo*.

In addition to wrong qNaN simulation, missing FP exception traps caused crashes when simulating *True Crime: New York City* with Dolphin. As described in the Dolphin progress report [103], the game's GameCube version is actually a PlayStation 2 port. Porting the game was probably a non-trivial undertaking, as the PlayStation 2 did not adhere to IEEE 754, while the GameCube (PowerPC) does. More specifically, a divide-by-zero generates the largest possible number on the PlayStation 2, while it is ∞ for the GameCube. Hence, a calculation like $(1/0) - (1/0)$ results in 0 for the PlayStation 2, and qNaN for the GameCube. To rectify this difference, the GameCube's port relies on the divide-by-zero FP exception trap. Using these traps all infinite values can be mapped to the largest, finite number. If these traps are not modeled in the simulator, qNaN values will eventually lead to the game's crash.

Still in the context of game simulation, also the proprietary N64-on-Wii simulator *Virtual Console* does not accurately model FP arithmetic. More specifically, the Virtual Console erroneously does not use RNE for 64-bit to 32-bit FP conversions, but RTZ. Directed rounding can lead to accumulating rounding errors, which can be observed when simulating *Super Mario 64* in Virtual Console (see Figure 5.6). As explained in [179], the game features a level, in which a platform keeps oscillating in a sinusoidal fashion using 32-bit FP variables. A possible implementation is shown in Code 5.5. As shown in the code, the software developers probably forgot to add an *f* suffix to the constant 0.58, which leads to the subsequent multiplication being performed in double precision. The result is later converted back to single precision and stored in variable *y*. As Virtual Console erroneously rounds the conversion to zero, the position of the platform progressively drifts from its vertical starting point at -3065 to 0. Due to rounding errors being rather small in comparison to the rounded number, position changes only become noticeable after multiple hours of simulation.



Figure 5.6: Upwards moving platform in *Super Mario 64* due to rounding errors caused by Round to Zero (RTZ). Footage from [180].

5.5 Methods

The main contribution of this thesis to the field of fast FP simulation is presented in this section. Thereby, the focus is on fast calculation of FP exception flags and rounding modes. By introducing methods for addition/subtraction, multiplication, division, square root, and FMA, all relevant arithmetic FP instructions are covered. Besides using mathematical means to check the validity of the proposed methods, all instructions were implemented in the RISC-V simulator SIM-V [108] and verified using the RISC-V Architecture Tests (RISCOF) [73]. Furthermore, a C++ implementation of all presented methods, including a comprehensive test suite, is available as open-source under: <https://github.com/not-chicken/FloppyFloat>. Note that in the following a positive residual is used (e.g., $c_{exact} + r = \circ(a + b)$). Hence, if $r > 0$, the result was rounded up, and if $r < 0$, the result was rounded down.

5.5.1 Fast Addition/Subtraction

As explained in Subsection 5.3.5, the Fast2Sum algorithm was used by You et. al [199] for the calculation of the residual r . The algorithm only requires two arithmetic operations, but the operands must be sorted by absolute value. Depending on the host ISA, sorting requires branching instructions, which can lead to performance penalties. As an alternative without sorted operands, the so-called 2Sum algorithm was introduced by O. Møller [137] in 1965. Similar to Dekker’s Fast2Sum algorithm, the 2Sum was initially conceived to increase accuracy in FP calculations. However, opposed to the Fast2Sum algorithm, 2Sum requires more arithmetic instructions. The details are provided in Equation 5.9.

$$\begin{aligned} c_{exact} + r &= c = \circ(a + b) \\ a' &= \circ(c - b) \quad b' = \circ(c - a') \\ \delta_a &= \circ(a' - a) \quad \delta_b = \circ(b' - b) \quad r = \circ(\delta_a + \delta_b) \end{aligned} \tag{5.9}$$

In benchmark experiments using a x64 host system, the 2Sum algorithm was 10% faster than the Fast2Sum algorithm when working on randomized data. If the input data is predictable, hence favorable to a branch predictor, both algorithms achieve a similar performance. A C++ implementation using the 2Sum for the fast calculation 32-bit FP additions for RUP rounding is given in Code 5.6.

While the 2Sum algorithm was initially chosen for this work, extensive tests revealed severe problems with overflows. For example, for two 16-bit FP values, assume an addition of 65504 (largest, finite 16-bit FP number)³, and -48. The result of the addition is inexact with a residual of 16:

$$\begin{aligned} c_{exact} + r &= c = \circ(a + b) \\ 65456 + 16 &= 65472 = \circ_{16}(65504 - 48) \end{aligned} \tag{5.10}$$

³ $2^{15} \cdot 1.111111111_2 = 65504$

```

1  // RUP case
2  float c = a + b; // Result.
3  float ad = c - b;
4  float bd = c - ad;
5  float da = ad - a;
6  float db = bd - b;
7  float r = da + db; // Residual.
8  if (r != 0.f) {
9      inexact = true;
10     if (r < 0.f) {
11         c = nextup(c);
12         overflow = (c == infinity) ? true : overflow;
13     }
14 }

```

Code 5.6: C++ code of a 32-bit addition using the 2Sum algorithm.

In the intermediate calculations of the 2Sum algorithm, the rounded-up value of c leads to infinite values:

$$\begin{aligned} a' &= \circ(c - b) \\ \infty &= \circ_{16}(65472 + 48) \end{aligned} \quad (5.11)$$

Ultimately, this leads to the residual being a qNaN:

$$\begin{aligned} r &= \circ(\delta_a + \delta_b) \\ qNaN &= \circ_{16}(\infty - \infty) \end{aligned} \quad (5.12)$$

Due to this flaw, SIM-V as well as the derived open-source library use the Fast2Sum algorithm.

5.5.2 Fast 32-bit Multiplication

For the fast calculation and rounding of multiplications, an inherent characteristic of IEEE FP numbers can be exploited: multiplying two 32-bit FP values as 64-bit is always exact. Similar to the 2Sum algorithm, this property allows to calculate a residual, which can be used for rounding and calculation of the inexact exception flag. This approach is presented in the following and referred to as the *UpMul* algorithm. In general, the following set of equations are required to calculate the residual:

$$\begin{aligned} c_{exact} + r &= c = a \cdot b + r = \circ_{32}(a \cdot b) \\ r &= a \cdot b + r - a \cdot b = \circ_{64}(\circ_{32}(a \cdot b) - \circ_{64}(a \cdot b)) \end{aligned} \quad (5.13)$$

The first step of deriving the UpMul algorithm is to show that the multiplication of the 32-bit values as 64-bit values is always exact. Using Equation 5.3 a multiplication can be expressed as:

$$a \cdot b = M_a \cdot M_b \cdot 2^{e_a + e_b - 2p_f + 2} = c = M_c \cdot 2^{e_c - p_d + 1} \quad (5.14)$$

As stated in Subsection 5.1.1, this model does not regard subnormal numbers. Yet, it can be used in the given case as casting 32-bit FP values to 64 bit never leads to subnormal results. And even the subsequent multiplication cannot generate a subnormal result. This is due to smallest subnormal 32-bit FP number being $2^{e_{f,min}-p_f+1} = 2^{-149}$. Multiplying the smallest subnormal 32-bit FP number with itself results in $2^{2 \cdot (-149)} = 2^{-298}$. Such a value is still outside the 64-bit subnormal range, which begins at $2^{e_{d,min}} = 2^{-1022}$.

Next, the maximum ranges of M_c and e_c are determined:

$$\begin{aligned} |M_c| &= |M_a \cdot M_b| \leq (2^{p_f} - 1)^2 \leq (2^{24} - 1)^2 \leq 2^{48} - 1 \leq 2^{p_d} - 1 \leq 2^{53} - 1 \\ |e_c| &= |e_a + e_b - 2p_f + p_d + 1| \leq 260 \leq |e_{d,min}| \end{aligned} \quad (5.15)$$

Since both M_c and e_c fit into the range of a double-precision value, the result of the multiplication is exact. As a last step, the exactness of the subtraction in Formula 5.13 needs to be shown. For this purpose, Sterbenz' Lemma [175] can be employed. According to the Lemma, the subtraction of two nearby FP numbers is always exact:

$$\begin{aligned} &\text{if } a/2 \leq b \leq 2a \\ &\text{then } \circ(b - a) = b - a \end{aligned} \quad (5.16)$$

Excluding infinity, the values of $\circ_{32}(a \cdot b)$ and $\circ_{64}(a \cdot b)$ can never have a difference that exceeds the machine epsilon ($\epsilon < 2^{-p}$). Hence, their subtraction is always exact.

A C/C++ implementation for a 32-bit multiplication using a RUP rounding mode is provided in Code 5.7. As presented in the code, the inexact flags is set if $r \neq 0$. Subsequently, the result is rectified in case the host FPU rounded it down. Increasing the result could lead to an overflow, which is why the result is checked for infinity. According to the RISC-V ISA, tininess is detected after rounding, necessitating an underflow check after rectification. Note that IEEE 754 defines an underflow as the non-normalized result being both subnormal and inexact. As presented in Subsection 5.1.1, there are cases in which a normal result can lead to an underflow. This can only happen for the smallest normal number and requires a soft float fallback.

```

1 // RUP case
2 float c = a * b;
3 double r = (double)c - (double)a * (double)b;
4 if (r != 0.) {
5     inexact = true;
6     if (r < 0.) {
7         c = nextup(c); // Next greater FP value.
8         overflow = is_inf(c) ? true : overflow;
9     }
10    underflow = (is_subn(c) || is_zero(c)) ? true : underflow;
11    underflow = is_min(c) ? SoftUf(a, b) : underflow;
12 }

```

Code 5.7: C++ code of a 32-bit multiplication using the UpMul algorithm.

5.5.3 Fast 32-bit Division

For the simulation of fast 32-bit FP divisions, a new method called *UpDiv* was developed in this thesis. Similar to the *UpMul* method from Subsection 5.5.2, both operands must be 32-bit FP values, and the goal is to compute the residual r . However, in this case, the exact determination of the residual of a division is overambitious, as many rational numbers cannot be represented with a finite number of significand bits. Nevertheless, the exact value of the residual is not crucial for determining inexactness and rounding directions. Rather, it suffices to know whether there was a rounding error, and if this error is positive or negative. In mathematical terms, an approximation of the residual $\tilde{r}$ is sought, for which $\text{sgn}(\tilde{r}) = \text{sgn}(r)$ is satisfied. Such an approximation is obtained by the following formula:

$$\begin{aligned} a/b + r &= c_{\text{exact}} + r = c = \circ_{32}(a/b) \\ \tilde{r} &= \circ_{64}(\circ_{64}(\circ_{32}(a/b) \cdot b) - a) \cdot \text{sgn}(b) \end{aligned} \quad (5.17)$$

This residual can then be used to simulate 32-bit FP instructions as shown in Code 5.8. The *UpDiv* equation can be derived by using the standard model of FP arithmetic extended for subnormals (see Equation 5.4). According to the model, the error of a FP division, including underflow and overflow, can be represented as follows:

$$\frac{a}{b} \cdot (1 + \epsilon_1) + \eta_1 = \circ_{32}(a/b) = a/b + r \quad (5.18)$$

If the result of the division is upcasted to 64-bit and multiplied by the value of b , which is also upcasted to 64-bit, the result must be exact (see Subsection 5.5.2 for the proof). This allows to calculate an approximation $\tilde{a}$ as follows:

$$\tilde{a} = a + a\epsilon_1 + b\eta_1 = \circ_{64}(b \cdot \circ_{32}(a/b)) \quad (5.19)$$

```

1 // RUP case
2 float c = a / b;
3 double r = (double)c * (double)b - (double)a;
4 r = signbit(b) ? -r : r;
5 if (r != 0.) {
6     inexact = true;
7     if (r < 0.) {
8         c = nextup(c); // Next greater FP value.
9         overflow = is_inf(c) ? true : overflow;
10    }
11    underflow = (is_subn(c) || is_zero(c)) ? true : underflow;
12    underflow = is_min(c) ? SoftUf(a, b) : underflow;
13 }
```

Code 5.8: C++ code of a 32-bit FP division using the *UpDiv* algorithm.

Subtracting a from $\tilde{a}$ yields Equation 5.20:

$$z = \circ_{64}(\tilde{a} - a) = \circ_{64}(a - \circ_{64}(b \cdot \circ_{32}(a/b))) = (a\epsilon_1 + b\eta_1)(1 + \epsilon_2)$$

$$z = \begin{cases} a\epsilon_1(1 + \epsilon_2) & \text{if } z \text{ is normal} \\ b\eta_1(1 + \epsilon_2) & \text{if } z \text{ is subnormal} \end{cases} \quad (5.20)$$

Although this addition can be inexact, which is described by ϵ_2 , a result of 0 can only be obtained if the preceding division was exact ($\epsilon_1 = \eta_1 = 0$). Otherwise, the sign of z is directly determined by $a\epsilon_1$ or $b\eta_1$. Next, Equation 5.20 is rearranged to:

$$\epsilon_1 = \frac{z}{a \cdot (1 + \epsilon_2)} \quad \eta_1 = \frac{z}{b \cdot (1 + \epsilon_2)} \quad (5.21)$$

Inserting Equation 5.21 into Equation 5.18 results for both cases in the following residual:

$$r = \frac{z}{b \cdot (1 + \epsilon_2)} = \frac{\circ_{64}(a - \circ_{64}(b \cdot \circ_{32}(a/b)))}{b \cdot (1 + \epsilon_2)} \quad (5.22)$$

Thus, the residual can only be 0 if z is 0 as well. Likewise, the sign of r is directly determined by z and b . Therefore, $\text{sgn}(\tilde{r}) = \text{sgn}(r)$ can be concluded.

5.5.4 Fast 32-bit Square Root

Following the same principle as the UpDiv algorithm from Subsection 5.5.3, an algorithm for 32-bit FP square root can be derived. The residual for this algorithm can be calculated as:

$$\sqrt{a} + r = b_{\text{exact}} + r = b = \circ_{32}(\sqrt{a})$$

$$\tilde{r} = \circ_{64}(\circ_{64}(\circ_{32}(\sqrt{a})^2) - a) \quad (5.23)$$

The proof of the algorithm is similar to the proof of the UpDiv algorithm. Again, an approximation $\tilde{r}$ for the residual r with $\text{sgn}(r) = \text{sgn}(\tilde{r})$ is sought. According to the standard error model of FP, the 64-bit multiplication of the 32-bit square root a results in:

$$\circ_{64}(\circ_{32}(\sqrt{a})^2) = \circ_{32}(\sqrt{a})^2 = (a \cdot (1 + \epsilon_1))^2 \quad (5.24)$$

Note that a square root can never produce a subnormal result (thus $\eta = 0$) and that a 64-bit multiplication of 32-bit values is always exact. Next, a is subtracted:

$$\tilde{r} = \circ_{64}(\circ_{32}(\sqrt{a})^2 - a) = ((a \cdot (1 + \epsilon_1))^2 - a) \cdot (1 + \epsilon_2) \quad (5.25)$$

This formula can be rearranged to:

$$\epsilon_1 = \sqrt{\frac{\tilde{r}}{(1 + \epsilon_2) \cdot a}} + 1 - 1 \quad (5.26)$$

Inserting ϵ_1 into $\sqrt{a} \cdot \epsilon_1 = r$ results in:

$$r = \sqrt{\frac{\tilde{r}}{1 + \epsilon_2} + a} - \sqrt{a} \quad (5.27)$$

Since the sign of r is only dependent on $\tilde{r}$, $\text{sgn}(r) = \text{sgn}(\tilde{r})$ holds. This residual can be used to calculate 32-bit square root FP instructions as shown in Code 5.9:

```

1 // RUP case
2 float b = sqrt(a);
3 double r = (double)b * (double)b - (double)a;
4 if (r != 0.) {
5     inexact = true;
6     if (r < 0.) {
7         b = nextup(b);
8     }
9 }

```

Code 5.9: C++ code of a 32-bit FP square root instruction using the UpSqrt algorithm.

5.5.5 Fast 32-bit Fused Multiply-Add

For the calculation of 32-bit FP FMA instructions a method based on the work of Sarrazin et al. [165] was employed. The key idea is to first calculate the exact multiplication of a and b using a larger data type. Subsequently, the residual of the summation of $a \cdot b$ and c is calculated using the Fast2Sum algorithm. But even if this summation was exact ($r_1 = 0$), the final result might not be representable as 32-bit FP value. Therefore, another residual r_2 is calculated to determine the 64-bit to 32-bit rounding error. The calculation of the residuals is shown in Equation 5.28.

$$\begin{aligned}
 d_{\text{exact}} + r &= d = \circ_{32}(a \cdot b + c) \\
 r_1 &= \text{Fast2Sum}(\circ_{64}(a \cdot b), c) \\
 r_2 &= \circ_{64}(d - \circ_{64}(\circ_{64}(a \cdot b) + c))
 \end{aligned} \quad (5.28)$$

Note that r_2 is always exact due to Sterbenz' Lemma [175]. Ultimately, an approximation of the rounding error $\tilde{r}$ can be calculated, as shown in Equation 5.29:

$$\tilde{r} = r_1 + r_2 \quad (5.29)$$

Although the addition of r_1 and r_2 is not exact per se, it satisfies $\text{sgn}(\tilde{r}) = \text{sgn}(r)$. This is facilitated by gradual underflows of IEEE 754 FP numbers, due to which the following property holds for two arbitrary 32-bit FP numbers: $\text{sgn}(a + b) = \text{sgn}(\circ_{32}(a + b))$.

Using the approximated residual, a C++ implementation for 32-bit FMA instructions is presented in Code 5.10.

```

1  // RUP case
2  float d = std::fma(a, b, c);
3  double p = (double)a * (double)b;
4  double dd = p + (double)c;
5  double r1 = fast2sum<double>(p, (double)c, dd);
6  double r2 = (double)d - dd;
7  double r = r1 + r2;
8  if (r != 0.) {
9      inexact = true;
10     if (r < 0.) {
11         d = nextup(d);
12         overflow = is_inf(d) ? true : overflow;
13     }
14     underflow = (is_subn(d) || is_zero(d)) ? true : underflow;
15     underflow = is_min(d) ? SoftUf(a, b, c) : underflow;
16 }
17 return d;

```

Code 5.10: C++ code of a simulated 32-bit FMA instruction.

5.5.6 Fast 64-bit Operations

The previous upcast algorithms UpMul, UpDiv, UpSqrt, and also the method for FMA instructions, are all based on the availability of larger data type that can perform multiplications exactly. As mentioned earlier, these algorithms reach their limitations for the simulation of 64-bit instructions on x64 and many other systems. However, most modern ISAs provide a 64-bit FMA instruction that can be used to circumvent these limitations to a certain degree. FMA instructions allow to calculate a combined addition and multiplication with infinite precision. Only the result is later rounded to the chosen data type. Mathematically speaking, a 64-bit FMA calculates:

$$d = \circ_{64}(a \cdot b + c) \quad (5.30)$$

This infinite precision is advantageous for many of the algorithms mentioned in the previous subsections. For example, using FMA, the residual of the UpMul algorithm can be calculated as follows:

$$r' = \circ_{64}(a \cdot b - \circ_{64}(a \cdot b)) = \circ_{64}(a \cdot b - c) = \circ_{64}(c_{exact} - c) \quad (5.31)$$

However, the rounding step at the end of a FMA instruction poses a problem. Although an FMA instruction calculates all intermediate results with infinite precision, the result is eventually rounded. Consequently, it is possible that r' is not representable with a 64-bit precision. If the FMA instruction is used without further ado, one could wrongly assume a residual of 0, although the value is actually different from 0. Hence, $r' = r$ does not hold in all cases.

The effects of this problem can be limited by determining bounds for which r' is no longer representable. Since r' is the direct result of the subtraction of c and c' , the smallest distance between these numbers, excluding 0, has to be determined. This

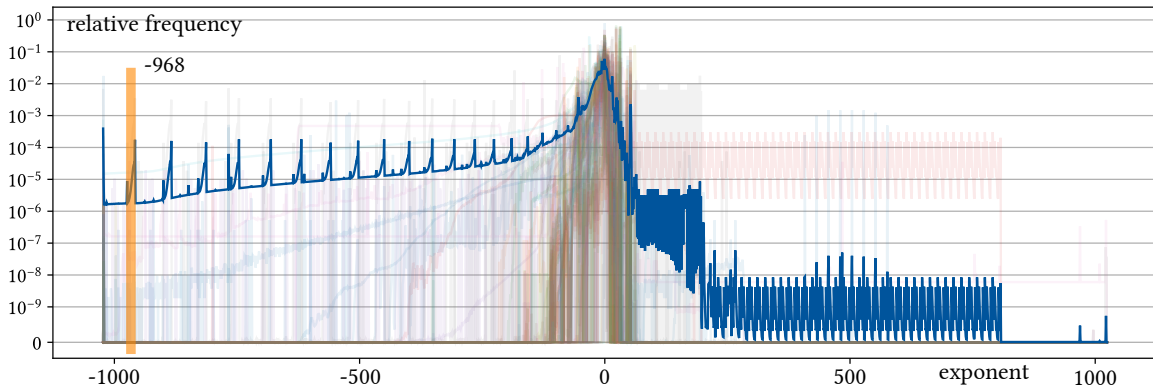


Figure 5.7: Relative frequency of 64-bit FP exponents in 78 different FP benchmarks from Figure A.1. The solid, blue line represents the average while individual benchmarks are represented in faint tones.

distance is $|d| \geq 2^{e_c - 2p_d}$. The number of double significand bits $2p_d$ follows from the exact intermediate results of the FMA instruction. As explained previously, $2p$ significand bits are needed for the exact representation of a p -bit multiplication. In order to represent r' as a 64-bit FP value, $e_c - 2p_d \geq e_{d,min} - p_d + 1$ must hold. A simple rearrangement leads to the following inequality:

$$e_c \geq e_{d,min} + p_d + 1 = -1022 + 53 + 1 = -968 \quad (5.32)$$

If $|c|$ is less than 2^{-968} , the FMA method cannot be used, and the instruction has to be calculated using soft float. However, the range below 2^{-968} represents less than 3% of all 64-bit FP values. In practice, even less values are below this threshold, as most FP values are centered around 1. To validate this statement, the 78 FP benchmarks from Figure A.1 in the appendix were executed in a simulation environment. This allowed to track the in- and output exponents of all 64-bit arithmetic FP instructions. The results are depicted in Figure 5.7. The experiment revealed, that on average less than 0.1% of all practically occurring values have an exponent less than 2^{-968} . A C++ example of a simulated 64-bit square division instruction is given in Code 5.11.

```

1  if (abs(a) < 4.008336720017946e-292)
2      return soft::div(a, b);
3
4  double r = std::fma(c, b, -a);
5  if (r != 0.) {
6      inexact = true;
7  }
```

Code 5.11: C++ code of a simulated 64-bit division instruction. Note that $2^{-968} = 4.008336720017946e - 292$.

5.5.7 Fast Vector Instructions

Due to the growing interest in machine learning, executing FP instructions in a vectorized fashion has become increasingly important. Currently, even state-of-the-art simulators, like QEMU, handle vector FP instructions as scalars. That means the simulator translates vector instructions into a for-loop comprising the scalar helper calls from the previous subsection. As these helper calls are rather complex, it is unlikely that a compiler-based autovectorization will change the situation. Ultimately, any parallel notion of the original target instruction is lost, which leads to poor simulator performance. However, as the presented approaches of this thesis are mostly based on arithmetic FP algorithms with only little control-flow, they can be extended to vector instructions with reasonable effort.

An example is provided in Code 5.12, which highlights the implementation of a 32-bit FP vectorized addition. As shown in the code, the implementation is based on the backend-independent library `std::experimental::simd`. The library adheres to ISO/IEC TS 19570:2018 Section 9 “Data-Parallel Types” and is currently under proposal to enter the non-experimental standard library with C++26 [117]. Thus, the presented approach can be used in other C++ simulators without requiring third party libraries. Similar to QEMU’s scalar implementation, the function starts with a rounding mode check. As shown in Line 4, the rounding mode of the target has to match with the C++ default rounding mode of RNE. Otherwise, the function falls back to a scalar soft float implementation. Note that other rounding modes can also be simulated by the approaches from the previous subsections. After the rounding mode check, the function’s core loop iterates over the FP vector (see Line 10). The loop divides the array into subsets, which are constrained by the host system’s native vector width. For example, the x64 AMD Ryzen Threadripper 3990x, which is used throughout this thesis, provides 8-element 32-bit FP vectors through AVX. In this case, `std::native_simd<f32>::size()` evaluates to 8. After calculating the arithmetic kernel of the instruction in Line 15, the challenges of Section 5.2 need to be solved. The handling of overflows and NaN values is shown in Line 17-28. Since corner cases tend to be infrequent, a preliminary check with the function `is_inf_or_nan` is performed in Line 17. Here, one can exploit specific IEEE 754 FP characteristics to quickly assess whether the result is $\pm\infty$ or NaN. More specifically, as shown in Code 5.13, the inequality $(a - a) \neq (a - a)$ only holds true for $\pm\infty$ and NaN. Only for these cases, special treatment may be required, which is shown in Line 17-Line 28. Next, Line 30 checks if the inexact exception flag must be determined. Similar to the scalar implementation, the Fast2Sum algorithm [137] can be used. As shown in Line 41, the addition was inexact if at least one calculated residual in the vector is non-zero. Ultimately, any tailing vector elements are calculated in Line 49-50, using the aforementioned scalar implementations. Besides additions, vector implementations of most other arithmetic FP instructions were developed as well. However, with `std::experimental::simd` still being worked on, vectorized versions of C++ math functions, such as `std::fma` and `std::sqrt` are still missing. Hence, the thesis relies on the compiler’s autovectorization for these cases.

```

1  namespace stdx = std::experimental;
2
3  void simd::add(f32* pa, f32* pb, f32* pc, size_t len) {
4      if (round != ROUND_NEAR) [[unlikely]] {
5          soft::add(pa, pb, pc, len);
6          return;
7      }
8
9      size_t i = 0;
10     while (i + stdx::native_simd<f32>::size() <= len) {
11         stdx::native_simd<f32> a, b, c;
12         a.copy_from(&pa[i], stdx::element_aligned);
13         b.copy_from(&pb[i], stdx::element_aligned);
14
15         c = a + b;
16
17         if (stdx::any_of(is_inf_or_nan(c)) [[unlikely]] {
18             if (stdx::any_of(is_inf(c)
19                             && !is_inf(a) && !is_inf(b))) {
20                 overflow = true;
21                 inexact = true;
22             }
23             if (stdx::any_of(is_nan(c) && is_inf(a) && is_inf(b)))
24                 invalid = true;
25             if (stdx::any_of(is_snan(a) || is_snan(b)))
26                 invalid = true;
27             stdx::where(c != c, c) = vqnan32;
28         }
29
30         if (!inexact) [[unlikely]] {
31             stdx::native_simd<f32> a_abs, b_abs, x, y, r;
32             a_abs = a;
33             b_abs = b;
34             stdx::where(a < 0, a_abs) = -a_abs;
35             stdx::where(b < 0, b_abs) = -b_abs;
36             x = a;
37             y = b;
38             stdx::where(a_abs < b_abs, x) = b;
39             stdx::where(a_abs < b_abs, y) = a;
40             r = (c - x) - y;
41             if (stdx::any_of(is_non_zero(r)))
42                 inexact = true;
43         }
44
45         c.copy_to(&pc[i], stdx::element_aligned);
46         i += stdx::native_simd<f32>::size();
47     }
48
49     for (; i < len; ++i)
50         pc[i] = fpu::add(pa[i], pb[i]);
51 }

```

Code 5.12: C++ code for a 32-bit vectorized FP addition.

```
1  template <typename T>
2  constexpr auto is_nan(T& a) {
3      return a != a;
4  }
5
6  template <typename T>
7  constexpr auto is_inf_or_nan(T& a) {
8      return (a - a) != (a - a);
9  }
10
11 template <typename T>
12 constexpr auto is_inf(T& a) {
13     return ((a - a) != (a - a)) && (a == a);
14 }
15
16 template <typename T>
17 constexpr auto is_non_zero(T& a) {
18     return (a != -a) && (a == a); // "false" for NaN.
19 }
```

Code 5.13: Using FP IEEE 754 characteristics for quick classification.

5.6 Results & Discussion

In the following subsections, the performance results of the proposed floppy float approach are presented. All experiments were executed on a AMD Ryzen Threadripper 3990x running Ubuntu 21.10. The machine can achieve a boost clock of 4.3 GHz and supports 256-bit FP vectors through Advanced Vector Extensions (AVX).

5.6.1 Cleanroom Benchmarks

In this subsection, instructions were isolated and benchmarked to assess their individual maximum performance in an environment without interference. This includes implementations as soft float and floppy float. Furthermore, instructions were executed natively (hard float) to see how close the aforementioned methods come to the practical optimum. To attain maximum performance, values of inputs and outputs are never subnormal, data dependencies between the instructions are minimal, standard rounding is used, and there is no DBT overhead. Also, the input data was designed to favor optimistic paths in the control-flow-heavy soft float implementation. The results of the experiments are depicted in Figure 5.8.

As shown in the figure, executing FP instructions natively (hard float) achieves up to 8500 MIPS for instructions that can be executed in one cycle (max, min, add, sub, etc.). These results can be explained by the FP pipeline of the AMD Ryzen Threadripper 3990X host processor. Most FP instructions can use two of the four FP slots provided by the Zen 2 microarchitecture, leading to $8500 \text{ MIPS} \approx 2 \cdot 4.3 \text{ GHz}$. Instructions, such as division, square root, or 64-bit multiplication, require multiple cycles, which results in a lower performance. Nevertheless, hard float is faster than soft and floppy float in all cases. In some cases, such as `fmadd32`, hard float is more than $80\times$ faster than a soft float implementation. The performance of the floppy float approach is in the range of 300-600 MIPS, and is faster than soft float by up to $5\times$ in some operations, such as square root. For lightweight operations, such as max or min, there is no significant difference between soft- and floppy float.

5.6.2 System Benchmarks

Since the approach of this thesis is intended to increase FP performance in DBT-based simulators, a practical performance assessment is indispensable. For this purpose, the method was integrated into the RISC-V-on-x64 simulator SIM-V [108]. Moreover, the method by Cota et al. [48] (QEMU's method), and Bellard's SoftFP [18] were also incorporated for comparison reasons. Subsequently, a performance analysis was conducted using a set of popular FP benchmarks such as linpack, NPB, and SPEC CPU 2017. The results are depicted in Figure 5.9 and Figure 5.10. Note that this work abstained from directly comparing the simulators SIM-V and QEMU as they differ in many performance-critical regards.

In the figure, the speedups of the individual benchmarks are depicted, whereby the soft float method was used as a reference. That means, a speedup of 1 would

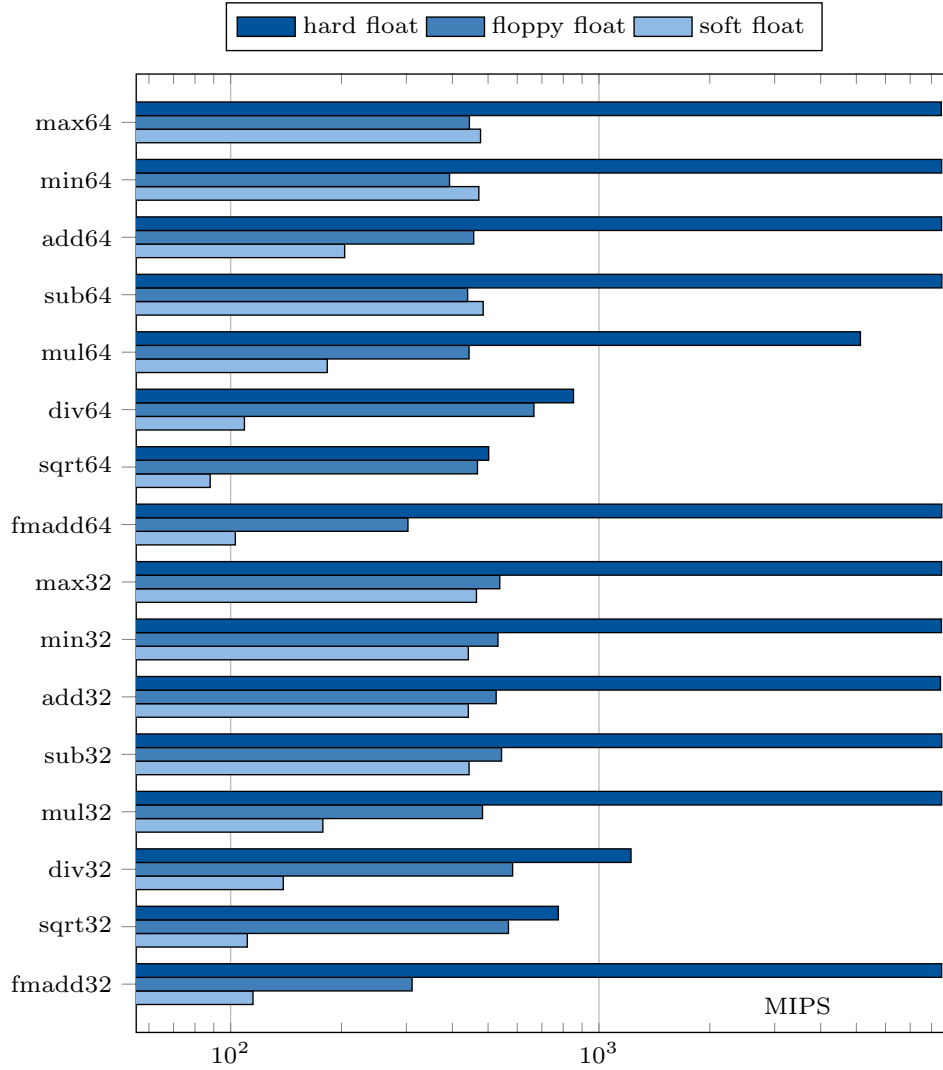


Figure 5.8: Performance results of soft, floppy, and hard float for individual instructions. Floppy float refers to the approach presented in this thesis, while soft float is based on SoftFP [18].

achieve no performance benefit over soft float implementation. All benchmarks in Figure 5.9 were executed with the default RNE rounding, while Figure 5.10 represents the same benchmarks under RUP rounding. As can be seen in Figure 5.9, QEMU’s method and the approach of this thesis achieve a speedup of $3\times$ in a best-case scenario (see NPB/ft.A and 508.namd). Also, in most cases, the performance of the proposed approach is equal to the performance of QEMU’s approach when RNE rounding is used. As explained previously, the approach of this thesis is only faster when underflows occur and no inexact flags are set, or when a non-default rounding mode is not used. Since most applications set an inexact exception flag after a few executed instructions, the speedup gained from an accelerated inexact calculation is marginal. Also, underflows are seldom. When executing the 78 FP benchmarks from

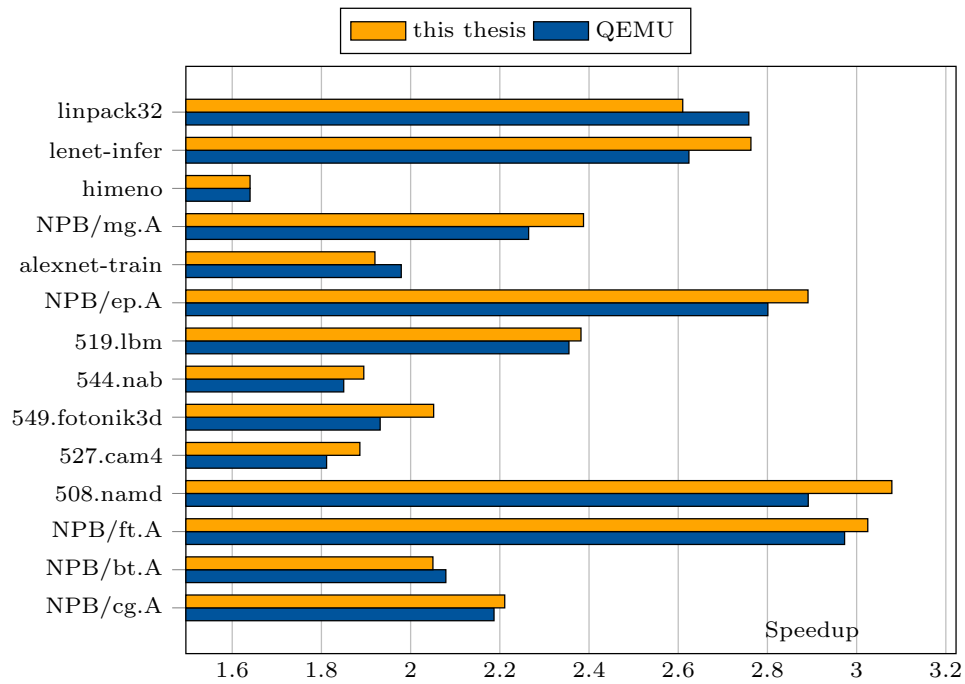


Figure 5.9: Comparing the method of this thesis against QEMU’s method with RNE rounding. Soft float was used as the baseline.

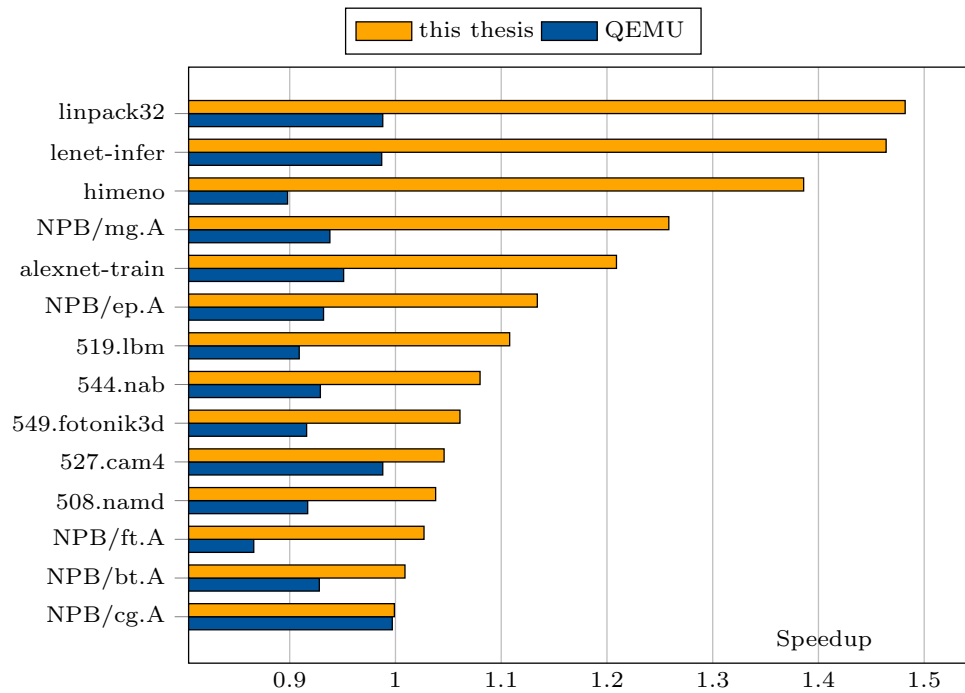


Figure 5.10: Comparing the method of this thesis against QEMU’s method with RUP rounding. Soft float was used as the baseline.

Figure A.1, underflows occurred once every 7992 arithmetic FP instructions. 59 of the 78 benchmarks did not raise a single underflow exception.

To demonstrate the advantages of the proposed method, all benchmarks were executed again under RUP rounding which is depicted in Figure 5.10. Here, it can be seen that QEMU is slower than soft float in all cases. This can be attributed to the fact that QEMU first checks the rounding mode before resorting to soft float. The method of this thesis, however, can rectify the result for most instructions and set the exception flags without using soft float. Thus, speedups of $1.48\times$ over QEMU are achieved for benchmarks like `linpack32`. Since the speedup of the method depends on the executed instructions, a heterogeneous picture of results emerges. Moreover, the speedups under RNE cannot be used to infer the speedups under RUP. As described in the previous subsections, there is currently no fast 64-bit FMA instruction, and all presented approaches require fewer checks when working on 32-bit data. Hence, single precision benchmarks, such as `linpack32` or machine learning applications (`lenet`, `alexnet`), achieve higher speedups in non-default rounding modes. Applications that comprise many 64-bit FMA instructions achieve low to no speedups (see `NPB/bt.A` and `NPB/cg.A`). Since FMA is one of the most frequent instructions (see Figure 5.11), there is some potential left for future optimizations. The work of Boldo et al. [21] proposed an algorithm to estimate the rounding error for FMA instructions, but the significant computational overhead renders it impractical for fast simulations.

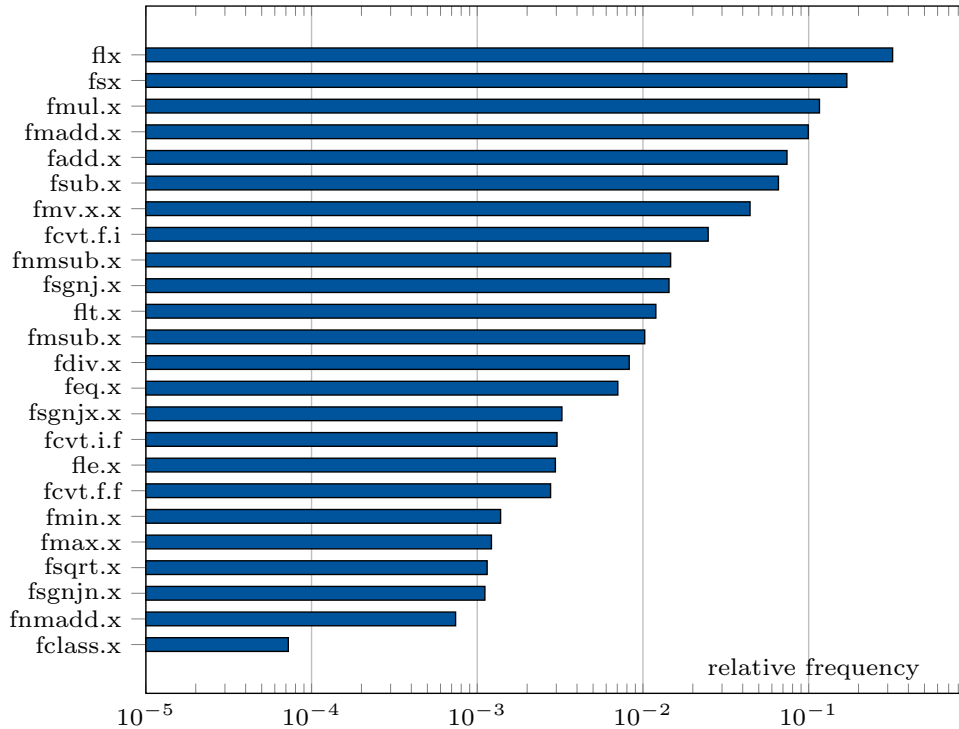


Figure 5.11: FP instruction distribution of the 78 benchmarks from Figure A.1.

5.6.3 Vector Benchmarks

As shown in Subsection 5.5.7, the approach of this thesis can be extended to vector instructions. To assess the maximum attainable speedup, the implementations of this thesis were benchmarked against QEMU's implementation in an isolated fashion.

This is similar to the cleanroom benchmarks from the previous subsection. The results are depicted in Figure 5.12. For vector instructions with 32-bit elements, the approach of this thesis attains speedups of up to $3.5\times$. The only exception is `vfsqrt`, which only saw a speedup of $1.39\times$. By closer examining the generated code, it became apparent that both gcc and clang did not autovectorize `std::sqrt` for unknown reasons. As mentioned before in the Subsection 5.5.7, FMA and square root operations for vectors are not yet supported by C++. Consequently, one has to rely on the compiler's autovectorization for these cases, which may or may not succeed. For 64-bit instructions, the AVX vector length is halved to four, which leads to smaller speedups compared to the 32-bit counterparts. Nevertheless, with speedups of around $2\times$, the vector method is still significantly faster than QEMU's method.

To see how the individual speedups translate into system benchmarks, the RiVEC benchmark suite [158] was executed using SIM-V with and without the approach of this thesis. Note that SIM-V's default method is akin to QEMU's method. The RiVEC benchmarks are particularly designed to benchmark vector instructions on RISC-V systems. Besides RiVEC, also scalar FP benchmarks were compiled with gcc autovectorization. However, besides `linpack`, the compiler was in no case able to find a significant number of vector instructions (see Figure 5.13). The performance increase of a vectorized approach is depicted in Figure 5.14. As can be seen, the highest speedup

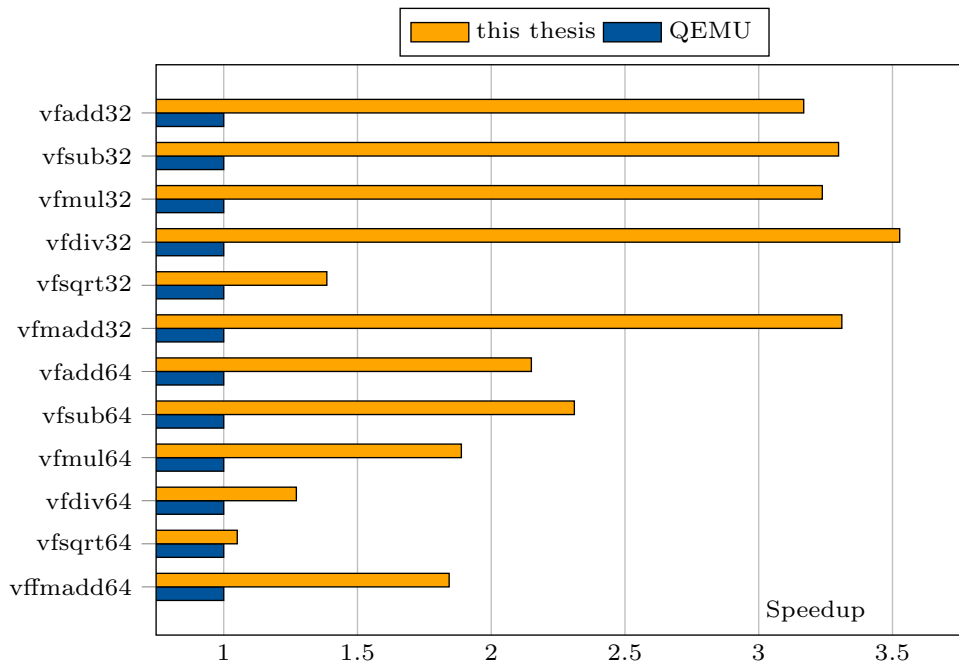


Figure 5.12: Comparing the performance of RISC-V FP vector instructions.

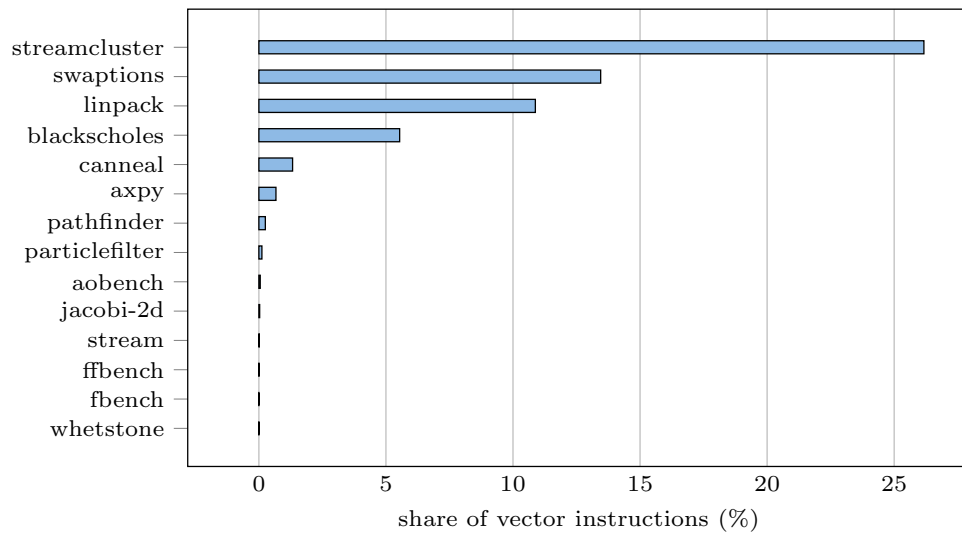


Figure 5.13: Overall share of vector instructions per benchmark.

of $1.24\times$ is attained for blackscholes. Other benchmarks only achieve speedups in the range of $1.0\times - 1.1\times$. These speedups are rather low compared to the results of the cleanroom benchmarks. A closer examination of the benchmarks revealed, that most of them comprise a relatively large number of load and store instructions. Hence, the increased arithmetic performance of the simulator only entails relatively small benefits for these memory-bound benchmarks.

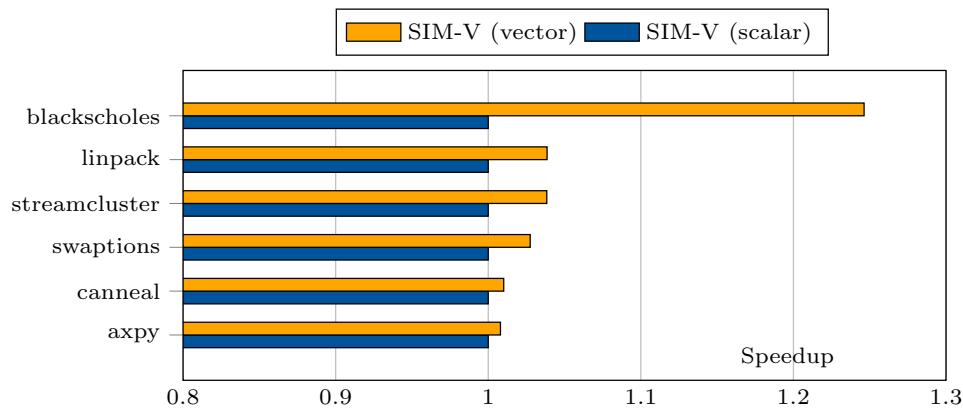


Figure 5.14: Comparing the performance of RISC-V vector benchmarks.

5.7 Conclusion & Outlook

This chapter presented new approaches for fast simulation of FP instructions. As a first step, existing work was analyzed to create the most complete picture of this topic to date. Subsequently, the findings were complemented with new approaches for the fast computation of inexact/underflow exceptions and the simulation of rounding modes. Furthermore, with the presented approaches being simple in terms of control flow, this thesis showed how fast vector instructions can be implemented as well. To emphasize the practical relevance, the approach of this thesis was integrated into the RISC-V simulator SIM-V [108] and compared to the state-of-the-art methods of QEMU [17]. This was achieved by comparing the attained performance of popular scalar FP benchmarks, such as SPEC 2007, NPB, and linpack. As shown in the results, the advantage of the new approach becomes evident when a non-default rounding mode is used. In these cases, benchmarks execute up to 50% faster than QEMU. For default rounding modes, the approach is on par with QEMU's method achieving speedups of up to $3\times$ compared to soft float. Since the speedup is based on a quick calculation of exception flags and rounding modes, architectures with non-sticky exception flags, such as PowerPC, also benefit for default rounding modes. In terms of FP vector instructions, the approach of this thesis yields speedups of up to $3.5\times$ for individual vector instructions and speedups of up to $1.24\times$ for system benchmarks. All methods of this chapter were incorporated into an open-source library called *Floppy Float* which is available under <https://github.com/not-chicken/FloppyFloat>. FloppyFloat is rigorously tested and validated. In fact, comprehensive comparisons against Berkeley Softfloat revealed bugs in said library [208].

While this thesis pushes the boundaries of fast FP simulation, some challenges remain unresolved. For instance, fast algorithms for 64-bit FMA instructions were still untouched. Since 64-bit FMA instructions occur relatively frequently, their fast simulation remains an interesting subject for future investigations. Also, the efficient simulation of narrower or exotic data types, such as float16 or bfloat16, might get more relevant due to the growing interest in machine learning.

Ultimately, the biggest challenge remains the performance gap between soft/floppy float and hard float. Even though floppy float is up to $3\times$ faster than soft float, hard float is still an order of magnitude faster. Although floppy float remains an interesting and mostly uncharted field of research, it seems unlikely that further improvement of this method will yield any significant performance gains. Hence, the last resort remains an implementation in hardware. ARM's FEAT_AFP in conjunction with Apple's Rosetta 2 simulator already showcased, how hardware-assisted FP simulation can approach near-native performance. This thesis therefore advises ISA designers to consider dedicated extensions for the simulation of FP instructions.

Chapter 6

Register Allocation for Dynamic Binary Translators

Similar to classical compilers, also simulators based on Dynamic Binary Translators (DBTs) face the ubiquitous challenge of Register Allocation (RA). In classical compiler engineering, RA refers to the mapping of an unlimited number of virtual variables to a limited number of physical registers. For DBTs, the unlimited number of virtual variables is simply the set of target registers which have to be mapped to the host register file. Although RA in classical compiler engineering is a thoroughly-explored field of research, established global methods may fail to meet expectations in DBT-based simulations. This is mainly due to the dynamic nature of DBT, which entails a constantly changing Control Flow Graph (CFG). If global methods are applied, every change to the CFG may require a subsequent global register allocation, which hampers the simulator's performance. Thus, state-of-the-art DBT-based simulators, such as QEMU [17] or SIM-V [108], employ local RA. This achieves an optimal solution per basic block, but requires loading and storing target registers at the beginning and end of each basic block. An analysis conducted in Section 6.3 showed, that basic blocks usually do not comprise more than 10 instructions. Consequently, additional loads and stores can have a significant impact on the simulation performance.

To remedy the flaw of local RA, this chapter proposes a new global RA method based on static mappings. The key idea is to exploit spatial locality of register usages in compiled programs. Experiments on an exhaustive set of RISC-V benchmarks showed that most applications only work on a small set of registers.

By keeping these frequently used target registers in static host registers, the number of additional loads and stores can be reduced by a significant amount. Since the registers are kept statically, changes in the CFG do not require reallocations. Even though the approach blocks parts of the register file, modern ISAs with larger register files, such as ARM64 or RISC-V, can compensate for that. To demonstrate the practical benefits of the proposed approach, the DBT-based RISC-V-on-ARM user-mode simulator RISE-SIM was developed for this thesis. For RISE-SIM, the static approach attains performance improvements of up to $1.4\times$ compared to a basic-block-local RA. In general, this chapter is an extended version of the author's work *Static Global Register Allocation For Dynamic Binary Translators* [212].

6.1 Background

As mentioned in this chapter's introduction, DBTs share many similarities with classical compilers, including the problem of RA. For classical compilers, the input of the RA phase is usually code that is close to assembly. But instead of registers, the individual instructions use a possibly unlimited number of automatic/virtual variables as operands. The key challenge of RA, is to map these variables to a limited number of physical registers on the host.

When the allocator cannot fit all active (also called "live") variables into the available registers, memory spills must be inserted. This involves storing variables in main memory and retrieving them when needed again.

The main goal of most allocators is to reduce the number of memory spills to produce efficient code. However, finding an optimal solution in a reasonable time is complex, as the underlying problem is NP-complete [35]. This has led to the development of numerous algorithms and heuristics over the past few decades. While these methods can be adapted for use in DBTs, they may not yield expected performance results. This is due to classical compilers and DBTs differing in the following aspects:

Number of Variables: In classical compiler engineering, there is no limit to the number of variables that need to be assigned to the host registers. However, with DBTs, the number of variables (i.e., target registers) is limited. For instance, when simulating non-embedded RISC-V, there are 32 general-purpose registers that must be mapped to the host register file.

Compilation Time: With DBTs, generating executable code is part of the runtime. Thus, if too much time is spent on RA or other optimizations, any resulting performance gains could be compromised. In classical compiler engineering, this issue only occurs with Just-In-Time (JIT) compilers, which are often employed for the compilation of dynamic programming languages, such as Python or Javascript.

Control Flow Graph Visibility: For traditional compilers, entire translation units are known at compile time. In contrast, DBTs operate under more uncertain conditions. For example, jump addresses of indirect branches can only be determined at runtime. This is why DBTs require alternating phases of execution and translation to progressively explore the CFG. Additionally, evictions of translated basic blocks and self-modifying code may also alter the CFG during runtime.

Preprocessed Target Code: For DBTs, the executed target code comprises registers, which were likely determined by a register allocator of a compiler. These allocators create inherent characteristics, which can be exploited by a DBT register allocator.

6.1.1 RISC-V/ARM64 Registers

This subsection provides greater detail about the calling conventions and register file structure of RISC-V and ARM64, as these ISAs serve as examples for the case study conducted in Section 6.4. Table 6.1 provides an overview of general-purpose registers and their roles in both ISAs. As can be seen in Table 6.1, registers are classified into callee-saved and caller-saved registers. Callee-saved registers, also known as call-preserved registers, retain their values before and after a function call. This means that if a callee-saved register is used within a function, a store instruction must be added in the function's prologue and a load instruction in the epilogue. In contrast, caller-saved registers, also called call-clobbered registers, may have their values altered by the function. If the caller needs to preserve the value, a store must be inserted before the function call and a load afterward. For RISC-V, the registers x0, x3 and x4 play a special role, as they cannot be used by the register allocator. While x3 and x4 can be overwritten in theory, the value of x0 is hardwired to zero. In addition to the classification into callee-saved and caller-saved registers, registers also have individual semantics. These semantics are defined in the Application Binary Interfaces (ABIs) of the ISAs and ensure interoperability between individual software components. For example, both RISC-V and ARM64, include eight argument registers for passing arguments to functions.

Name	ABI Mnemonic	Meaning	Call-preserved?
ARM64			
r0 - r7	-	Argument	No
r8	-	Indirect result	No
r9 - r15	-	Temporary	No
r16	IP0	Temporary, IP	No
r17	IP1	Temporary, IP	No
r18		Temporary, platform	No
r19 - r28		Callee-saved	Yes
r29	FP	Frame pointer	Yes
r30	LR	Return address	Yes
SP		Stack pointer	Yes
RISC-V			
x0	zero	Zero	– (Immutable)
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	– (Unallocatable)
x4	tp	Thread pointer	– (Unallocatable)
x5 - x7	t0 - t2	Temporary	No
x8 - x9	s0 - s1	Callee-saved	Yes
x10 - x17	a0 - a7	Argument	No
x18 - x27	s2 - s11	Callee-saved	Yes
x28 - x31	t3 - t6	Temporary	No

Table 6.1: General-purpose registers and semantics for RISC-V [161] and ARM64 according to AAPCS64 [12].

6.2 Related Work

While RA has been extensively explored in traditional compiler engineering, there are only a handful of publications of RA in DBTs.

One of the first works in this domain was written by Zhu et al. [201] in 1999. Parts of their work include a comparison between different register allocation methods (greedy, fixed, static). While the results differ up to $9\times$ in performance, the work provides very few details about the implementation.

The work of Zhu et al. [201] is also mentioned in a survey-like paper of M. Probst from 2002 [156] about DBTs. M. Probst also recognizes that register allocation for DBTs is relatively uncharted field of research.

A work about improving QEMU's RA was written by Hu et al. [91] in 2009. Instead of using the instruction-local RA of QEMU v0.9.0, the authors propose a basic-block-local method. Their method attains an average speedup of $1.22\times$ for Loongson-on-x86 translation, with some benchmarks reaching a speedup of $1.75\times$. These results are in line with the observations made in this thesis, as later shown in Subsection 6.4. Also in 2009, the group of Hu et al. use a modified version of QEMU for legacy-software support on their newly developed Godson-3 CPU. Without mentioning any details, such as methods or results, they employed a "fixed register allocation" [90]. Still in 2009, Cai et al. [31] compared several allocation methods for the their MIPS-on-x86 Crossbit DBT. This includes an instruction-local RA, a live-variable-based basic-block-local RA, a global RA based on dynamic usage frequencies of registers, and a simplified global graph-coloring RA. The global dynamic RA bears similarities to the method of this thesis, however, in their approach only one variable is statically mapped leading to no significant improvement. The low number of statically mapped variables is a result of IA-32's limited register file which only comprises 8 registers. Ultimately, the best performance is achieved by a simplified graph-coloring RA. Yet, the differences among the approaches are marginal. For instance, the performance difference between the worst method (instruction-local RA) and the best method never exceeds $1.11\times$. Similar performance observations were also made in a work by Wang et al. [190] in 2018 using a x86-on-Sunway DBT. For the sake of completeness, the work of Liang et al. [198] shall also be mentioned.

6.3 Methods

In this section, RA methods for DBTs are introduced. This includes the methods used by QEMU [17] and SIM-V [124], as well as the strategy of this thesis.

6.3.1 Instruction-Local Register Allocation

This subsection presents an instruction-local RA. The presented method is simple and easy to implement, but creates code with a maximum number of spills. However, due to this characteristic, the method is later used to assess impact of RA on DBT performance. As its name suggests, instruction-local RA handles RA on a per-instruction basis. This means that all source operands are loaded before executing the translated instruction, and all destination operands are stored afterwards. Code 6.1 provides an example for RISC-V on ARM64 DBT, showcasing the translation of two arithmetic RISC-V instructions (add and mul). Since both ARM64 and RISC-V implement addition and multiplication similarly, a RISC-V add/mul can be directly mapped to an ARM64 add/mul. However, before executing these instructions, the operands from the target register file must be loaded. The target register file is typically represented by a contiguous data structure in the DBT code, as shown by struct RegisterFile in Code 6.1. Note that the static assertion ensures a C-style memory layout of the struct. If a pointer to the target register file is stored in a host register, target registers can

```

1  // C++ implementation of the register file.
2  struct RegisterFile {
3      std::array<u64, 32> x;
4      std::array<u64, 32> f;
5      u64 pc;
6  } rv_regfile;
7
8  static_assert(std::is_standard_layout_v<RegisterFile>);
9
10 // Target code: RISC-V
11 add x1, x2, x3 // x1 = x2 + x3
12 mul x5, x1, x4 // x5 = x1 * x4
13
14 // Host code: ARM64
15 ldr r20, [r27, #16] // Load x2 into r20
16 ldr r21, [r27, #24] // Load x3 into r21
17 add r22, r20, r21 // Perform add
18 str r22, [r27, #8] // Store x1 into struct
19 ldr r20, [r27, #8] // Load x1 into r20
20 ldr r21, [r27, #32] // Load x4 into r21
21 mul r22, r20, r21 // Perform mul
22 str r22, [r27, #40] // Store x5 into struct

```

Code 6.1: Example of instruction-local RA.

be accessed using loads with specific offsets. In the provided example, r27 holds this pointer, which is initialized during the DBT's prologue. Hence, an instruction like `ldr r20, [r27, #16]` loads the third element of `std::array<u64, 32> x` into r20. Once the operands are loaded, the instruction is executed, and the result is stored back into the target register file. This load-and-store process is performed for each instruction, even if subsequent instructions use the same operands. For instance, in the example, x1 is reloaded for the `mul` instruction, even though the values are already available in r22. This method introduces multiple loads and stores for each translated instruction, potentially impacting overall performance. Despite this inefficiency, early versions of QEMU (up to version 0.9.0) adopted a similar approach due to its simplicity.

6.3.2 Basic-Block-Local Register Allocation

A more sophisticated RA on basic-block level is employed by QEMU [17] and SIM-V [108]. SIM-V's approach involves a linear scan of the basic block and assigning registers in a greedy fashion. An example using the same target code as in the previous subsection (Subsection 6.3.1) is presented in Code 6.2. Similar to the former method, operands used for the first time must be loaded from the target register file. However, during the scan, the register allocator tracks which target registers are already loaded. Therefore, the second instruction, `mul x5, x1, x4`, does not reload register x1; only x4 is loaded and assigned the next available register. At the end of the basic block, all modified target registers are written back.

A limitation of SIM-V's greedy approach is the lack of foresight. The allocator does not know which registers will be reused in future instructions. For example, after the addition, the values of r20/x2 and r21/x3 are not used again, meaning those registers could be reassigned. As a result, r20 and r21 are unnecessarily blocked.

QEMU's register allocation addresses this issue. Like SIM-V, it performs a linear scan and assigns registers greedily, but it also conducts a live-variable analysis beforehand. This analysis identifies which target registers are "live" (i.e., needed in future instructions), preventing unnecessary blocking of host registers and reducing the likelihood of memory spills. However, this improvement comes with the added

```

1 // Target code: RISC-V
2 add x1, x2, x3 // x1 = x2 + x3
3 mul x5, x1, x4 // x5 = x1 * x4
4
5 // Host code: ARM64
6 ldr r20, [r27, #16] // Load x2 into r20
7 ldr r21, [r27, #24] // Load x3 into r21
8 add r22, r20, r21 // Perform add
9 ldr r23, [r27, #32] // Load x4 into r23
10 mul r24, r20, r23 // Perform mul
11 str r22, [r27, #8] // Store x1 into struct
12 str r24, [r27, #40] // Store x5 into struct

```

Code 6.2: Example of basic-block-local RA.

cost of increased translation time due to the extra overhead of a live-variable analysis. Additionally, with this approach, instruction selection and register allocation must be handled as two distinct phases.

6.3.3 Global Register Allocation for DBT

In this section the contribution of this thesis is presented: a global RA for DBT. The problem of basic-block-local RA is its narrow scope, requiring loads and stores per basic block. Specifically, the number of target registers used (read or write) within a basic block sets a minimum for the number of additional loads and stores. For instance, the target code of the previous subsection in Code 6.2 uses five different target registers, requiring three loads and two stores. In the worst-case scenario, memory spills can increase the number of loads and stores even further. Given that basic blocks tend to be short, it can be assumed that the additional loads and stores of basic-block-local RA degrade DBT performance. This statement is based on an analysis of RISC-V benchmarks (see Figure 6.1). For the given benchmarks, the length of the average basic block is 12.2 instructions.

To address the inherent limitations of local procedures, *global RA* approaches the problem on an inter-basic-block level. An well-established and widely used method is graph coloring allocation, originally proposed by Chaitin et al. [35] in 1981. This method typically produces quality code, but the quadratic runtime complexity makes it impractical for DBTs and JIT compilers. A more efficient global RA method, so-called *linear scan*, was introduced by Poletto et al. [155] in 1999. Due to its linear runtime, linear scan and variants of it are often used in JIT compilers, such as the v8 Javascript engine [56].

While all conventional global RA can be applied to DBT, the frequently changing CFG poses a major challenge. New basic blocks can be found by indirect branches,

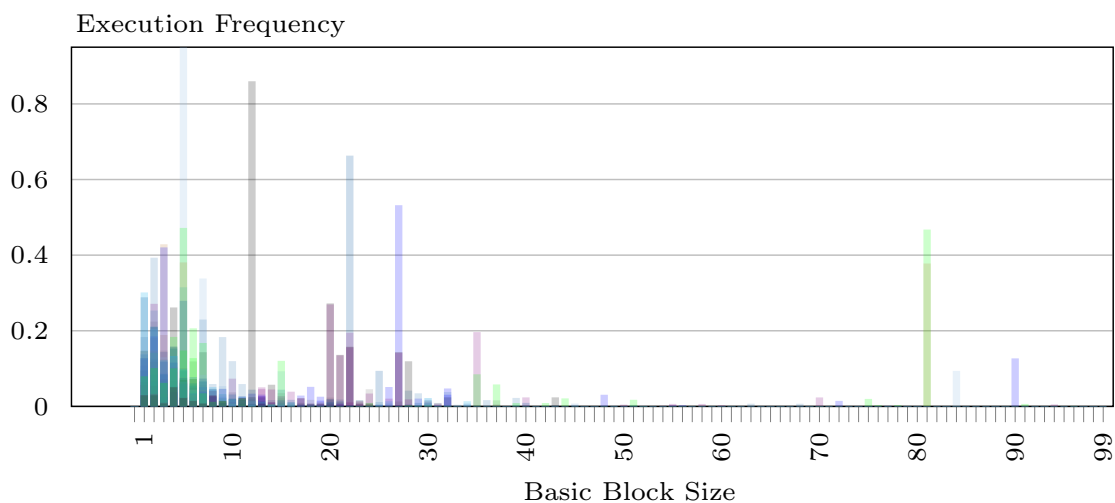


Figure 6.1: Size distribution of executed basic blocks in individual RISC-V binaries (applications from Figure A.1). On average, a basic block comprises 12.2 instructions.

and old basic blocks may either be evicted from the basic block cache or invalidated by self-modifying code. For that reason, this thesis proposes a static global RA, which requires no reevaluations even if the CFG changes.

The core concept is to store frequently used target registers in static host registers, while less frequently used registers are handled with basic-block-local allocation. This idea was motivated by a statement of A. Waterman, one of the RISC-V co-inventors. In his thesis *Design of the RISC-V Instruction Set Architecture*, he states that “instructions express great spatial locality of register reference” [191]. His statement was based on a dynamic and static analysis of the SPEC CPU 2006 benchmarks. To verify his assertion and build upon it, this thesis analyzed more than 114 different benchmarks compiled for RISC-V (see Figure A.1). These benchmarks span a wide range of scenarios from embedded software to high performance computing, and cover multiple different programming languages. Figure 6.2 depicts the results of the register utilization analysis. As shown in the figure, there is indeed a high degree of spatial locality, which also remains relatively consistent across all benchmarks. For instance, on average 60.1% of all register accesses are covered by the eight argument registers (a0-a7), which only constitute 25% of the register file. Slight differences between benchmarks are not only caused by workload characteristics, but also depend on the used compiler. For instance, the highest peak at register a5 can be explained by gcc’s register allocator which was used for most benchmarks. In case of gcc, argument register are used from a5 to a0. This stands in contrast to clang, which assigns registers from a0 to a5.

A downside of the static approach is a mandatory initialization of the static registers in the DBT prologue. Even if registers are not used in the target code being executed, the corresponding values have to be loaded. Furthermore, the approach of this thesis prefers to load statically mapped registers in callee-saved registers to avoid frequent reloading due to function calls from the generated code. As shown in

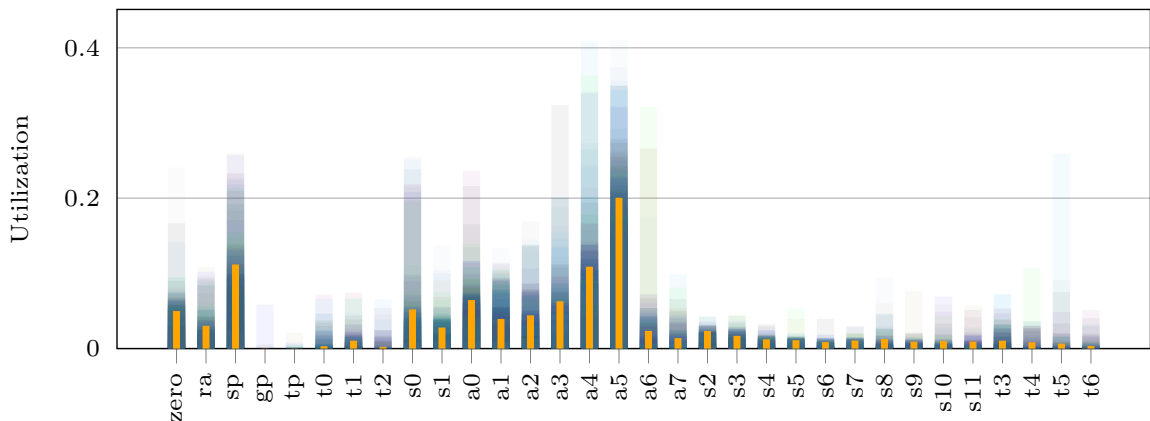


Figure 6.2: Utilization of RISC-V general-purpose registers in 114 benchmarks from Figure A.2. In total 169,111,077,904,832 register accesses were performed. Shaded bars represent individual benchmarks. The average is given by the orange bar. Most frequently used registers in descending order are: a5, sp, a4, a0, a3, s0, zero, a2, a1, ra, s1, a6, s2, s3, a7, s8, s4, s5, s7, t3, t1, s10, s11, s9, s6, t4, t5, t6, t0, t2, gp, tp.

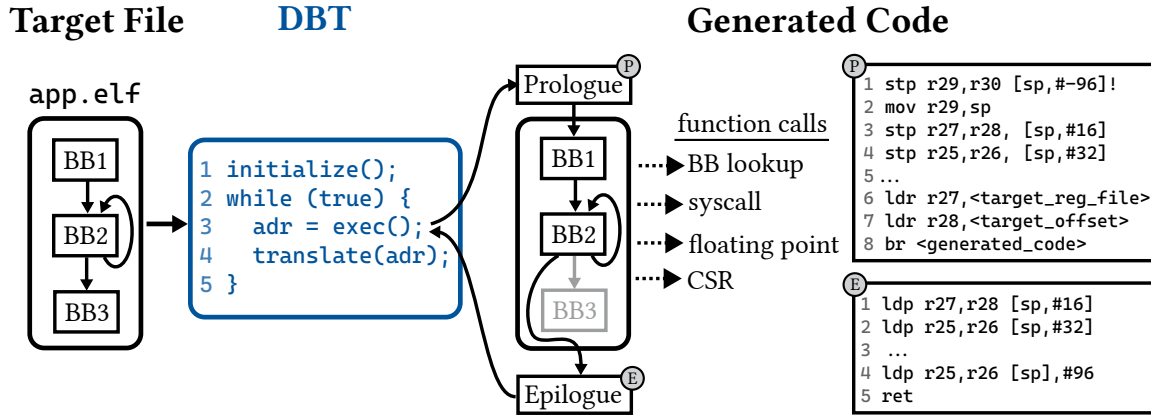


Figure 6.3: Extended principles of a DBT.

“function calls” in Figure 6.3, function calls include basic block lookups due to indirect branches, system calls, Floating Point (FP) instructions, and reads and writes of Control/Status Registers (CSRs).

As these function calls can be frequent, especially in FP benchmarks, DBTs, such as QEMU and RISE-SIM, prefer to store target registers in callee-saved host registers. With the approach of this thesis, parts of the ten callee-saved ARM64 registers are already reserved for statically mapped registers. Therefore, striking a balance between static and dynamic register mapping is crucial. As later shown in the results section, an optimal configuration was obtained by mapping eight RISC-V registers statically: a5, a4, a3, a0, ra, sp, x0, and a2. These registers cover 69.5% of all register accesses.

6.4 Results & Discussion

In this section, the performance results of running benchmarks with different RA methods are presented.

6.4.1 Optimal Number of Statically Mapped Registers

In the previous subsection, the exact layout of the static register mapping is determined by empirical means. To do so, the static RA method was incorporated into the RISC-V-on-ARM64 user-mode simulator *RISE-SIM*, which was developed for this thesis. Subsequently, a comprehensive set of benchmarks was executed, whereby each benchmark was run with different numbers of statically allocated registers. Note that for a number of zero statically mapped registers, the method of this thesis essentially becomes a basic-block-local method. The results are depicted in Figure 6.4. As shown in the figure, two different patterns of speedup behavior emerge: benchmarks that attain greater speedups with increasing number of statically allocated registers, and benchmarks that attain lower speedups with increasing number of statically allocated registers. The latter pattern is especially exhibited by FP benchmarks. With these benchmarks, every FP instruction results in a function call from the DBT-generated

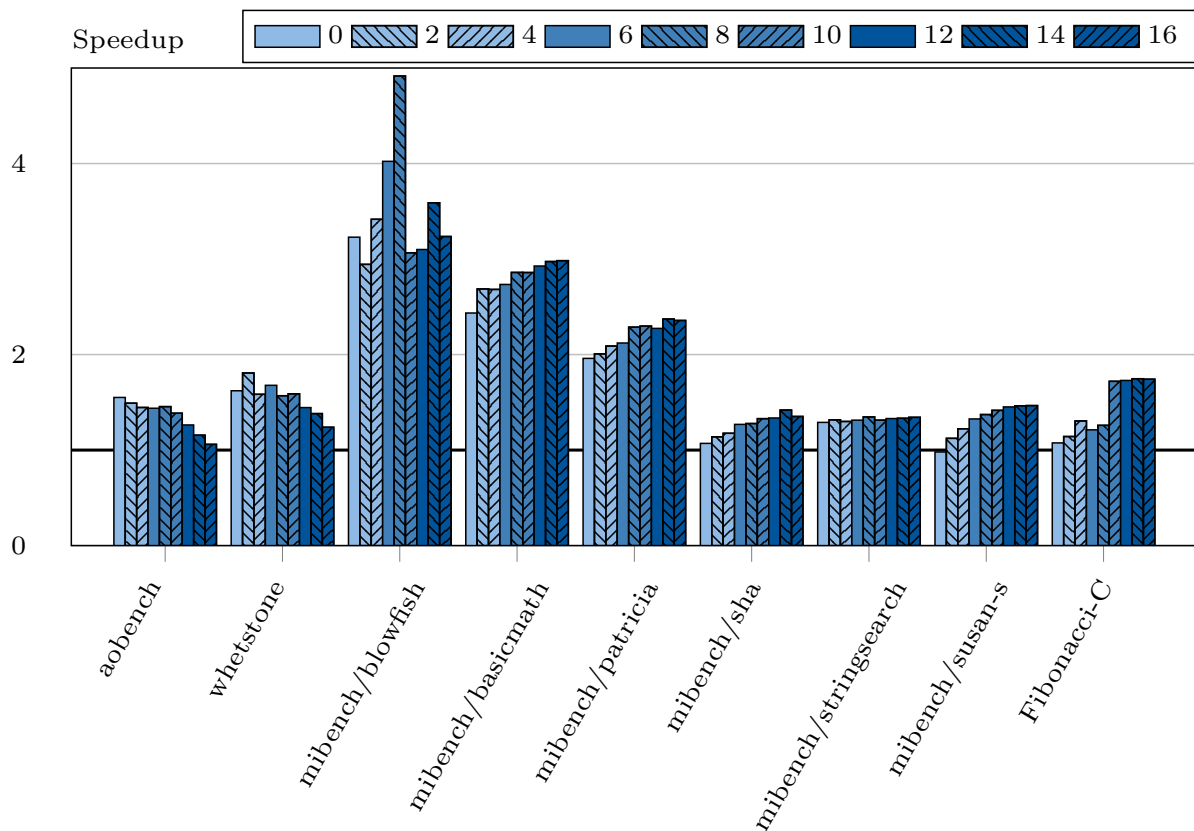


Figure 6.4: Speedup of the static RA approach using different numbers of statically mapped registers. Each setup maps the $N \in \{0, 2, 4, 6, 8, 10, 12, 14, 16\}$ most frequently used target registers. QEMU v7.2.0 was used as a reference for the speedup.

code. As the static register allocator prefers to put static registers in callee-saved registers, the risk of putting live registers in caller-saved increases. At ten statically mapped registers, also statically mapped registers must reside in caller-saved registers. Hence, every function call involves storing and reloading registers, which leads to the simulation performance almost linearly decreasing. Opposed to FP benchmarks, most integer- and control-flow-based benchmarks show greater speedups with more statically mapped registers. However, the returns in terms of performance diminish with an increasing number of statically mapped registers. In some cases, e.g., mibench/blowfish or mibench/sha, the performance even degrades.

To conclude, the layout of statically mapped registers heavily depends on the executed workload. In the following subsections, a setup of eight statically mapped RISC-V registers (a5, a4, a3, a0, ra, sp, x0, and a2) is used as this number provides significant for integer benchmarks, while still not affecting FP benchmarks too severely. All statically mapped registers were mapped to callee-saved registers on the ARM host (r19-r26).

6.4.2 Benchmark Results

In this subsection, the performance of the static RA method is compared against other methods. Similar to the previous subsection, a comprehensive set of benchmarks was executed, whereby each benchmark was run under four different setups:

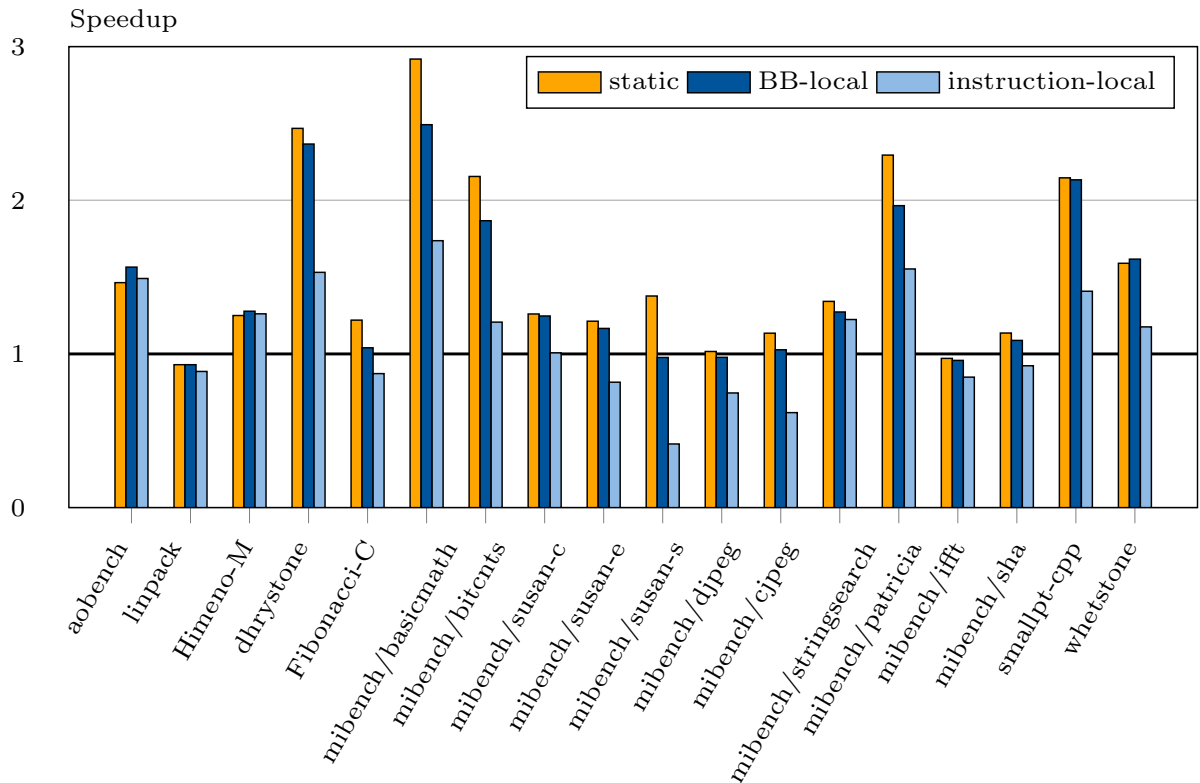


Figure 6.5: Performance of different RA methods in RISE-SIM. User-mode QEMU v7.2.0 was used as a performance reference.

1. RISE-SIM with instruction-local RA: The instruction-local RA from Subsection 6.3.1 uses stores and loads per translated instruction. Hence, it should yield the worst-case performance.
2. RISE-SIM with basic-block-local RA: This setup uses basic-block-local RA approach without live variable analysis (see Subsection 6.3.2).
3. RISE-SIM with static RA: The approach of this thesis as explained in Subsection 6.3.3. A setup with eight statically mapped registers (a5, a4, a3, a0, ra, sp, x0, and a2) was used. The number is motivated by the results in Figure 6.4.
4. QEMU v7.2.0: A basic-block-local RA with live variable analysis is used by the open-source simulator QEMU.

Figure 6.5 depicts the results of the benchmark study. As shown in the Figure, the highest performance is attained by static RA, followed by basic-block-local and instruction-local approaches. The speedup difference between instruction-local RA and static RA can exceed $3\times$ (see “mibench/susan-s”). This underscores the role of efficient RA for optimizing the performance of user-mode DBTs. On average, RA also outperforms the basic-block-local method by $1.1\times$, with some benchmarks, such as “mibench/susan-s”, reaching performance differences of up to $1.40\times$. Yet, as also shown and explained in previous subsection, the basic-block-local RA can slightly outperform the static approach for FP benchmarks. In general, RISE-SIM outperforms QEMU in almost all cases. However, RISE-SIM and QEMU differ in many aspects, such as FP simulation, basic block caching, and code optimization.

6.4.3 Result Analysis

To better understand the observed performance results from the previous subsection, the code generated by RISE-SIM and QEMU was analyzed in greater detail. Due to its simplicity and high locality, the Fibonacci microbenchmark is an ideal starting point.

The main loop of the Fibonacci benchmark including code generated QEMU and RISE-SIM with the static allocator is depicted in Figure 6.6. Starting from the benchmark’s C code, the kernel of the Fibonacci benchmark is compiled to the RISC-V machine code in Figure 6.6 b). For the compilation, RISC-V gcc 12.2.2 with maximum optimizations (O3) and standard extensions was used. The generated ARM host code of QEMU and RISE-SIM is shown in Figure 6.6 c).

As shown in the generated code, both RISE-SIM and QEMU start by fetching target register a1 from the main memory, which is needed for the subsequent `mv a3, a1`. Note that target register a1 is not included in the eight statically mapped registers of RISE-SIM. After fetching the only read operand, the translated target instruction is executed in Line 2. This also marks the first and last time, at which the value of register a3 changes in the basic block. Hence, the RA of QEMU writes the value back to the main memory as shown in Line 3. Since RISE-SIM keeps the target register a3 in static host registers, no write-back needs to be performed. A similar situation also arises for other target registers (Line 4, 6, 7).

Opposed to RISE-SIM, code optimizations are also performed by QEMU, which becomes evident in Line 10 and Line 11. The target instruction `mv a4, a3` is skipped by QEMU as target register `a3` is directly written into the memory location of `a4`. Code optimizations in JIT compilers and DBTs should always be carefully considered, as spending too much time on optimizations may outweigh the benefits of improved code. For this reason, RISE-SIM does not perform any optimizations.

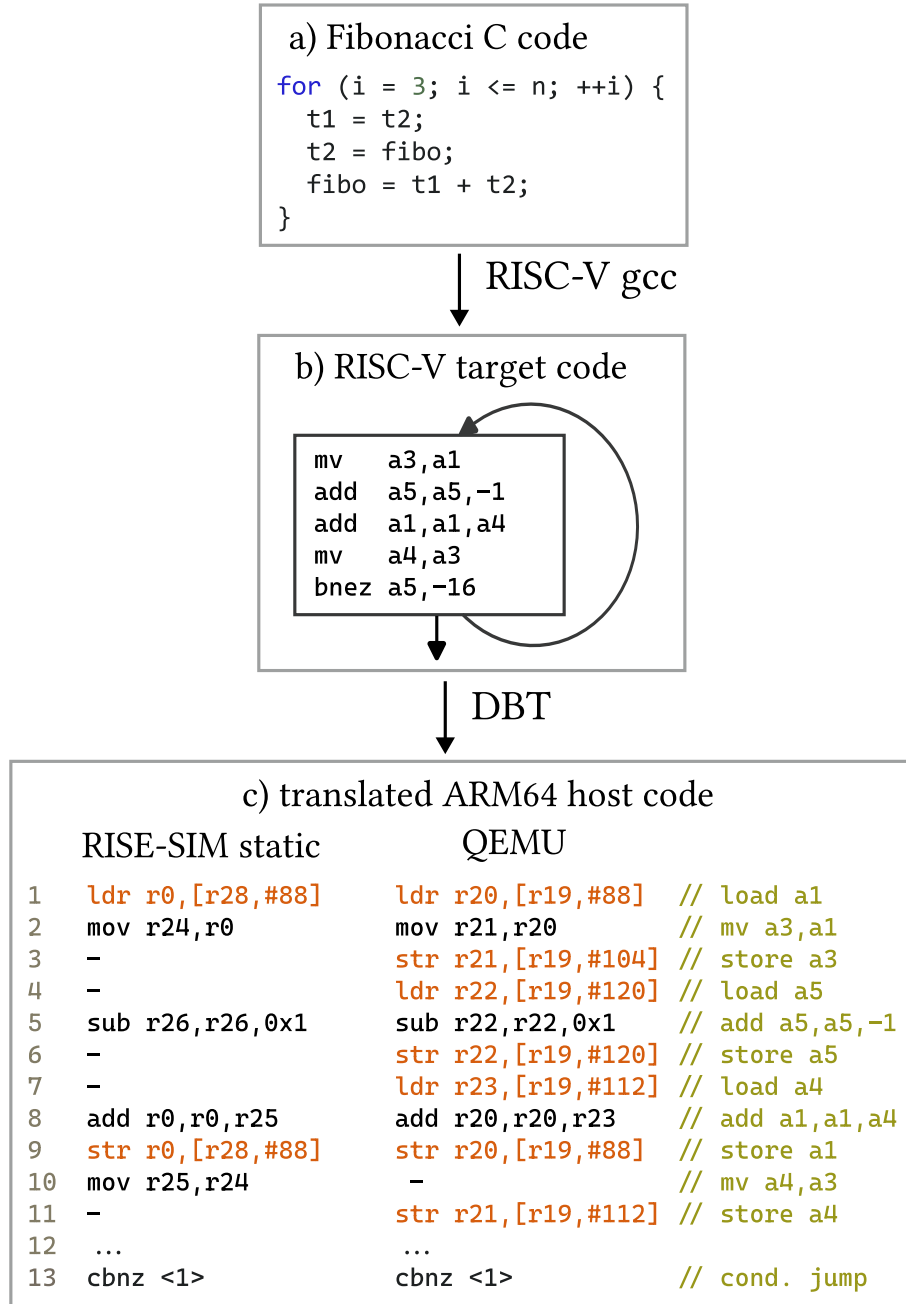


Figure 6.6: Depicting DBT-generated ARM64 code for the Fibonacci microbenchmark. RISE-SIM static uses eight statically mapped registers.

Ultimately, the branch instruction in Line 13 marks the end of the basic block. Depending on the Fibonacci number to be generated, the control flow either jumps back to the first instruction or continues with the subsequent, non-depicted instruction. Since the branch address can be determined during translation time, both RISE-SIM and QEMU apply a technique called basic block chaining. That means instead of resorting to expensive address lookups, a conditional branch instruction (`cbnz`) is directly patched into the generated code.

Overall, the Fibonacci benchmark example showcases why optimal RA is essential for fast DBTs simulations. While the machine code of the target only consists of five instructions, QEMU requires as much as eleven instructions. Furthermore, the additional instructions are loads and stores which can severely impact the simulator's performance. The static RA of RISE-SIM attains the same functionality with only seven instructions. In fact, if the static register allocator with ten registers is used, RISE-SIM only requires five instructions, as the load and store in Line 1 and Line 9 can be omitted. This also explains the sudden speedup jump of the Fibonacci benchmark in Figure 6.4 when the number of static registers is increased from eight to ten.

6.5 Conclusion & Outlook

This chapter showed how a static global RA can significantly improve the performance of RISC-on-RISC DBT-based user-mode simulators. Using the RISC-V-on-ARM64 simulator RISE-SIM, an average $1.1\times$ speedup could be achieved over commonly used basic-block-local methods.

The key idea of the approach of this thesis is to only statically map a frequently used subset of registers. Although RISC-V offers 32 general-purpose registers, most applications utilize only a few of them. As shown in the benchmark study in Section 6.3.3, eight registers cover 69.5% of all register accesses. By statically mapping these registers to host registers, corresponding load and store instructions can be moved from a basic block's entry/exit to the simulation entry/exit. This reduces the overall number of memory instructions leading to an improved performance on average. Opposed to other global RA methods, the static nature of the approach does not require expensive reruns in case the CFG changes. This characteristic is important for DBTs, where code may change due to newly discovered control paths or translation cache invalidations.

However, with the global approach, target registers are loaded/stored without consideration for actual usage. This may become a problem for applications which frequently leave the simulated code due to system calls, FP instructions, or basic block lookups. As shown in Section 6.4, the performance of FP benchmarks can decrease linearly with respect to the number of statically registers. This underscores the importance of mapping only a selected subset and highlights the need for a sufficient number of host registers. In the case of this thesis, a setup of eight statically mapped RISC-V registers (a5, a4, a3, a0, ra, sp, x0, and a2) achieved balanced results.

If FP instructions could be handled without leaving the simulated code, higher speedups could be achievable. This connects to a proposal of hardware extensions for FP simulation made in the conclusion of Chapter 5. With said extension, FP instructions do not need to be handled by expensive function calls, but simply use the instructions of the host system. Not only would the extension accelerate the simulation of FP instructions, but also the static RA allocation approach would benefit from less function calls.

In conclusion, the method presented in this thesis presents a step forward in the quest for achieving near-native execution speeds in DBT-based simulations.

Chapter 7

Summary & Future Work

This thesis pursued the overarching goal of making Full-System Simulators (FSSs) faster with contributions in parallelization, Temporal Decoupling (TD), Floating Point (FP) simulation, and Register Allocation (RA).

Chapter 3 showed how a parallelization of the FSS gem5 increased the simulator's performance by more than $20\times$ when simulating a 64-core system on a 128-core host system. With ever larger multi-core systems to be developed, the impact of parallelization will further increase, underpinning the need for parallel simulations. While this thesis as well as other works show how parallel FSS is feasible, one of the last missing milestones is a parallel standard. Such a standard would facilitate the seamless interaction between different simulators and hardware models. Although there are discussions about extending SystemC to parallel simulations, there have not yet been any substantial results.

Besides parallelization, this thesis introduced new and faster methods for the simulation of FP instructions in Chapter 5. By using the host FPU for primary calculations, while doing software rectifications in corner cases, individual instructions see speedups of up to $5\times$ compared to soft float. The implementation of this work is available as open-source ¹. Although the methods and results presented here represent a significant progress, a purely native implementation is still an order of magnitude faster. Hence, as shown with the extensions in ARMv8.7 and LoongArch, this problem can only be solved efficiently in hardware.

As another major, performance-critical factor of DBT-based simulators, this thesis covered the topic of RA. Opposed to basic-block-local RA methods, as used by QEMU, this thesis proposes a global, static approach. The approach exploits the spatial locality of register file accesses, by mapping a certain subset of the target registers statically to the host registers. This requires an ample number of registers, which is provided by all modern RISC Instruction Set Architectures (ISAs) (e.g., RISC-V and ARM64). With x64 now extending its register file to 32 registers, the here presented approach may also become feasible for x64 in near future. The static mapping allows for speedups of up to $1.4\times$ compared to QEMU's basic-block-local method.

As a last point, the author of this thesis would like to raise awareness for considering FSSs when designing an ISA or a standard. These are often conceived with certain criteria in mind, like impact on the hardware's performance, the associated chip cost, or security implications. A question that rarely arises is: How can a certain feature be simulated? Especially the chapter of FP simulation highlighted how various decisions of IEEE 754 and the RISC-V community severely complicate simulations.

¹<https://github.com/not-chicken/FloppyFloat>

Appendix A

Appendix

SPEC CPU 2017 [174] (1) 503.bwaves (2) 507.cactuBSSN (3) 508.namd (4) 510.parest (5) 511.povray (6) 519.lbm (7) 527.cam4 (8) 538.imagick (9) 544.nab (10) 549.fotonik3d (11) 554.roms	glmark2 [67] (18) buffer (19) build (20) bump (21) clear (22) conditionals (23) desktop (24) effect2d (25) function (26) ideas (27) jellyfish (28) loop (29) pulsar (30) refract (31) shading (32) shadow (33) terrain (34) texture	NPB [143] (12) NPB.bt.A (13) NPB.cg.A (14) NPB.ep.A (15) NPB.ft.A (16) NPB.mg.A (17) NPB.sp.A	OpenNN [13] (35) iris_plant (36) breast_cancer (37) simple_approx (38) simple_class (39) logical_operations (40) airfoil (41) mnist (42) outlier_detection
FinanceBench [33] (43) Black Scholes (44) Bonds (45) Monte Carlo	NumPy [82] (48) linalg (49) scalar	Other (66) fbenc [187] (67) ffbenc [188] (68) linpack32 [14] (69) linpack64 [14] (70) whetstone [148] (71) stream [128] (72) lenet-infer (73) alexnet-train (74) cray [178] (75) aobench [72] (76) glxgears (77) himeno [87] (78) SciMark 2.0 [133]	CoreMark-PRO [57] (50) loops-all-mid-10k (51) linear_alg-mid-100x100 (52) nnet_test (53) radix2-big-64k
mibench [78] (60) basicmath (61) susan (62) qsort (63) lame (64) rsynth (65) fft	Octane 2.0 [181] (46) raytrace (47) navierstoke		smallpt [139] (54) smallpt-c (55) smallpt-cpp (56) smallpt-java (57) smallpt-erlang (58) smallpt-numpy (59) smallpt-python3

Figure A.1: All 78 benchmarks and their corresponding index for this work.

SPEC CPU 2017 [174] 500.perlbench 503.bwaves 505.mcf 507.cactuBSSN 508.namd 510.parest 511.povray 519.lbm 520.omnetpp 523.xalancbmk 525.x264 527.cam4 531.deepsjeng 538.imagick 541.leela 544.nab 548.exchange2 549.fotonik3d 554.roms 557.xz	glmark2 [67] buffer build bump clear conditionals desktop effect2d function ideas jellyfish loop pulsar refract shading shadow terrain texture	NPB [143] NPB.bt.A NPB.cg.A NPB.ep.A NPB.ft.A NPB.mg.A NPB.sp.A	OpenNN [13] iris_plant breast_cancer simple_approx simple_class logical_operations airfoil mnist outlier_detection
FinanceBench [33] Black Scholes Bonds Monte Carlo	Octane 2.0 [181] all box2d codeload crypto deltablue earleyboyer mandreel navierstoke raytrace regex splay	Other alexnet-train aobench[72] buildroot-boot cray[178] cryptsetup dhrystone-clang dhrystone-gcc fbench[187] ffbench[188] glxgears himeno[87] lenet-infer linpack32[14] linpack64[14] SciMark 2.0[133] stream[128] whetstone[148]	NumPy [82] linalg 62 scalar
smallpt [139] smallpt-c smallpt-cpp smallpt-java smallpt-erlang smallpt-numpy smallpt-python3		CoreMark-PRO [57] loops-all-mid-10k linear_alg-mid-100x100 nnet_test radix2-big-64k	mibench [78] adcpm basicmath bitcount blowfish crc32 dijkstra fft gsm jpeg lame patricia qsort rsynth sha stringsearch susan-edges typeset

Figure A.2: The 114 benchmarks performed 169,111,077,904,832 register accesses.

```

1  #include <bit>
2  #include <cfenv>
3  #include <cmath>
4  #include <cstdint>
5  #include <iostream>
6  #include <stdfloat>
7
8  using namespace std;
9  using u16 = uint16_t;
10 using f16 = float16_t;
11 using f32 = float32_t;
12
13 void PrintRes(f16 res) {
14     cout
15         << "Result: 0x" << hex << bit_cast<u16>(res) << "\n"
16         << "Invalid: 000" << fetestexcept(FE_INVALID) << "\n"
17         << "Underflow: 0" << fetestexcept(FE_UNDERFLOW) << "\n"
18         << "Is normal: 0" << isnormal(res) << "\n"
19         << "Inexact: 000" << fetestexcept(FE_INEXACT) << "\n\n";
20 }
21
22 int main() {
23     // Always underflows.
24     f32 a = bit_cast<f32>(0x387FE000u);
25     // Underflows only for tininess before rounding.
26     f32 b = bit_cast<f32>(0x387FF800u);
27     // Always underflows.
28     f32 c = bit_cast<f32>(0x387FE800u);
29
30     for (auto v : {a, b, c}) {
31         f16 res;
32         feclearexcept(FE_ALL_EXCEPT);
33         res = static_cast<f16>(v);
34         PrintRes(res);
35     }
36
37     return 0;
38 }

```

Code A.1: Test program showcasing how normal results can lead to the underflow flag being set.

```

1
2 // x64                                RISC-V
3 // max(sNaN, 5.f) = 5                  max(sNaN, 5.f) = 5
4 // max(5.f, sNaN) = nan                max(5.f, sNaN) = 5
5 // Invalid: 0                          Invalid: 16
6 // (int)2e10 = -2147483648              (int)2e10 = 2147483647
7
8 #include <cfenv>
9 #include <cmath>
10 #include <iostream>
11 #include <limits>
12
13 template <typename T>
14 using nl = std::numeric_limits<T>;
15
16 int main() {
17     // Maximum/Minimum
18     float r1 = nl<float>::signaling_NaN();
19     float r2 = 5.f;
20     float snan = nl<float>::signaling_NaN();
21     float qnan = nl<float>::quiet_NaN();
22 #ifdef __x86_64
23     asm("maxss%0,%1" : "=x"(r1) : "x"(5.0f));
24     asm("maxss%0,%1" : "=x"(r2) : "x"(snan));
25 #elif __riscv
26     asm("fmax.s%0,%1,%2" : "=f"(r1) : "f"(5.0f) , "f"(r1));
27     asm("fmax.s%0,%1,%2" : "=f"(r2) : "f"(snan) , "f"(r2));
28 #else
29     static_assert(false, "No architecture detected.");
30 #endif
31     std::cout << "max(sNaN, 5.f) = " << r1 << std::endl
32               << "max(5.f, sNaN) = " << r2 << std::endl;
33
34     // Fused Multiply-Add
35     std::feclearexcept(FE_ALL_EXCEPT);
36     float res = std::fma(0, nl<float>::infinity(), qnan);
37     std::cout << "Invalid: "
38               << std::fetestexcept(FE_INVALID) << std::endl;
39
40     // Float to Integer
41     volatile float a = 2e10;
42     std::cout << "(int)2e10 = " << (int) a << std::endl;
43
44     return 0;
45 }

```

Code A.2: Test program to trigger different FP behaviour on RISC-V and x64.

```

1  #include <iostream>
2
3  int main() {
4      const float a = -0.f;
5      const double b = -0.;
6      double out;
7
8      #ifdef __x86_64
9          // Storing the float does not touch the upper bits.
10         // Hence, the output is 0x8000000080000000 (-1.06099e-314).
11         asm volatile("movsd_%2,_%%%xmm0\n\t\
12         movss_%1,_%%%xmm0\n\t\
13         movsd_%%%xmm0,_%0"
14             : "=x" (out) : "x" (a), "x" (b) : "xmm0");
15     #elif __riscv
16         // Output should be -qNaN due to RISC-V NaN boxing.
17         asm volatile("fmv.d_f0,_%2\n\t\
18         fmv.s_f0,_%1\n\t\
19         fmv.d_f0,_%0"
20             : "=f" (out), "f" (a) : "f" (b) : "f0");
21     #else
22         static_assert(false, "No_architecture_detected.");
23     #endif
24
25     std::cout << "out_=" << out << std::endl;
26     return 0;
27 }

```

Code A.3: Test program showcasing NaN boxing.

```

1  // x64:      0xffc00123
2  // RISC-V: 0x7fc00000
3  #include <iostream>
4
5  int main() {
6      float a = 0.f;
7      float b;
8      unsigned int *ai = reinterpret_cast<unsigned int *>(&a);
9      unsigned int *bi = reinterpret_cast<unsigned int *>(&b);
10     *bi = 0xffc00123;
11     a += b;
12
13     std::cout << std::hex << "0x" << *ai << std::endl;
14     return 0;
15 }

```

Code A.4: Test program highlighting NaN propagation of x64.

List of Figures

1.1	Landscape of simulation in computer engineering.	1
2.1	Principles of an interpreter-based Instruction Set Simulator (ISS) in C-like pseudo code.	8
2.2	Principles of Dynamic Binary Translators (DBTs).	9
2.3	Real-Time Factor (RTF) values of QEMU for various benchmarks when simulating RISC-V user mode on an x86 host (AMD Threadripper 3990X). Benchmarks from Figure A.2.	11
3.1	Development of transistors per processor and highest single-threaded PassMark results [149] for state-of-the-art general-purpose processors released in the given year. From 2012 (Intel Xeon E3-1290 v2) to 2021 (Apple M1), the number of transistors increased by 1042%, while single-thread performance only grew by 73%.	15
3.2	gem5 Discrete Event Simulation (DES) in sequential and parallel mode.	17
3.3	gem5 port protocols.	21
3.4	Assignment of event queues (q_i) in atomic mode. Transactions travelling from the CPUs to the memory system are referred to as travelling downstream, while the other direction is referred to as upstream.	23
3.5	Number of simulated instructions for bubble sort using various quanta.	28
3.6	Accuracy of simulated time for bubble sort with various quanta.	28
3.7	Speedup for individual NAS Parallel Benchmarks (NPB) benchmarks.	29
3.8	Accuracy of simulated time when executing the NPB IS with various quantum sizes.	30
4.1	Simplified mental model of accuracy and performance used in many works using TD.	33
4.2	Principle of TD.	35
4.3	Example timings for a sequential temporally-decoupled simulation with (T_{real}) and without (T_{ideal}) context-switching costs.	38
4.4	Time per sync for condition variables and barriers on Linux (black) and Windows (orange).	40
4.5	Estimated speedup vs. measurement for a bare-metal bubble sort benchmark using the simple, imprecise model as described by Equation 4.8.	41
4.6	Types of timing errors in TD-based simulations.	42

4.7	Refined accuracy and performance mental model of TD.	44
4.8	Sequential temporal decoupling performance results.	46
4.9	Sequential temporal decoupling accuracy results.	47
4.10	Parallel temporal decoupling speedup results.	49
4.11	Parallel temporal decoupling accuracy results.	50
4.12	Comparing linux boot timestamps of a TD and a default simulation in par-gem5.	51
4.13	Comparing Million Instructions Per Seconds (MIPS), speedup, and instructions for the NPB IS benchmark running on the AVP64 SystemC VP. Each metric was normalized to their value at $t_{\Delta q} = 1$ ms.	52
4.14	Throughput of the iperf3 benchmark as a function of the quantum.	53
5.1	Bit representation of IEEE 754 binary32 and binary64.	57
5.2	Floating point control and status register of x64 and RISC-V.	60
5.3	Comparing RISC-V NaN boxing to ARM64.	64
5.4	Comparing NaN propagation on RISC-V and x64.	65
5.5	Missing enemies due to wrong Not a Number (NaN) simulation in <i>Dragon Ball: Revenge of King Piccolo</i>	75
5.6	Upwards moving platform in <i>Super Mario 64</i> due to rounding errors caused by Round to Zero (RTZ). Footage from [180].	76
5.7	Relative frequency of 64-bit FP exponents in 78 different FP benchmarks from Figure A.1. The solid, blue line represents the average while individual benchmarks are represented in faint tones.	84
5.8	Performance results of soft, floppy, and hard float for individual instructions. Floppy float refers to the approach presented in this thesis, while soft float is based on SoftFP [18].	89
5.9	Comparing the method of this thesis against QEMU's method with Round to Nearest, Ties to Even (RNE) rounding. Soft float was used as the baseline.	90
5.10	Comparing the method of this thesis against QEMU's method with Round Up (RUP) rounding. Soft float was used as the baseline.	90
5.11	FP instruction distribution of the 78 benchmarks from Figure A.1.	91
5.12	Comparing the performance of RISC-V FP vector instructions.	92
5.13	Overall share of vector instructions per benchmark.	93
5.14	Comparing the performance of RISC-V vector benchmarks.	93
6.1	Size distribution of executed basic blocks in individual RISC-V binaries (applications from Figure A.1). On average, a basic block comprises 12.2 instructions.	101
6.2	Utilization of RISC-V general-purpose registers in 114 benchmarks from Figure A.2. In total 169,111,077,904,832 register accesses were performed. Shaded bars represent individual benchmarks. The average is given by the orange bar. Most frequently used registers in descending order are: a5, sp, a4, a0, a3, s0, zero, a2, a1, ra, s1, a6, s2, s3, a7, s8, s4, s5, s7, t3, t1, s10, s11, s9, s6, t4, t5, t6, t0, t2, gp, tp.	102

6.3	Extended principles of a Dynamic Binary Translator (DBT).	103
6.4	Speedup of the static RA approach using different numbers of statically mapped registers. Each setup maps the $N \in \{0, 2, 4, 6, 8, 10, 12, 14, 16\}$ most frequently used target registers. QEMU v7.2.0 was used as a reference for the speedup.	104
6.5	Performance of different RA methods in RISE-SIM. User-mode QEMU v7.2.0 was used as a performance reference.	105
6.6	Depicting DBT-generated ARM64 code for the Fibonacci microbenchmark. RISE-SIM static uses eight statically mapped registers.	107
A.1	All 78 benchmarks and their corresponding index for this work.	113
A.2	The 114 benchmarks performed 169,111,077,904,832 register accesses. .	114

List of Tables

2.1	List of popular open-source FSS. All links accessed on 2024-07-01. . . .	13
3.1	Published work on parallel FSSs since 2004.	20
3.2	Configuration of the host and target system.	27
3.3	4-core 1 μ s quantum simulation of NPB IS.	30
5.1	Constraints for Equation 5.3	58
5.2	32-bit quiet Not a Number (qNaN) values of different ISAs.	62
5.3	Latencies and throughput of instructions for loading and storing the <code>mxcsr</code> register according to [65]. The throughput is the maximum number of instructions of the same kind that can be executed per clock cycle assuming that operands are independent of preceding instructions. The latency of an instruction is the delay of an instruction in a dependency chain.	66
5.4	Differences of FP-related characteristics among ISAs.	68
6.1	General-purpose registers and semantics for RISC-V [161] and ARM64 according to AAPCS64 [12].	97

List of Algorithms

Bibliography

- [1] “Dolphin Emulator,” <https://dolphin-emu.org/>, accessed: 2025-04-15.
- [2] “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.
- [3] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter, “X86-64 Instruction Usage among C/C++ Applications,” in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 68–79. [Online]. Available: <https://doi.org/10.1145/3319647.3325833>
- [4] M. Alian, D. Kim, and N. S. Kim, “pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems,” *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 41–44, 2015.
- [5] K. Alshaya, “StackOverflow - Simulator or Emulator? What is the difference?” <https://stackoverflow.com/questions/1584617/simulator-or-emulator-what-is-the-difference>, 2009, accessed: 2025-04-15.
- [6] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: <https://doi.org/10.1145/1465482.1465560>
- [7] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On Subnormal Floating Point and Abnormal Timing,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 623–639.
- [8] Apple Inc., “Apple unleashes M1,” <https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/>, 11 2020.
- [9] “Apple M3 Characteristics,” <https://nr.apple.com/Di5I4t7da8>, Apple Inc., 10 2023, accessed: 2025-04-15.
- [10] “ARMv8 Virtual Platform (AVP64),” <https://github.com/aut0/avp64>, ARM, accessed: 2025-04-15.

- [11] “ARM Architecture Reference Manual,” <https://developer.arm.com/documentation/ddi0487/latest>, Arm Holdings plc, accessed: 2025-04-15.
- [12] *Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)*, <https://github.com/ARM-software/abi-aa>, Arm Holdings plc, accessed: 2025-04-15.
- [13] “OpenNN Examples,” <https://github.com/Artelnics/opennn/tree/master/examples>, Artelnics, accessed: 2025-04-15.
- [14] “linpack,” <https://www.netlib.org/linpack/>, AT&T Bell Laboratories, the University of Tennessee and Oak Ridge National Laboratory, accessed: 2025-04-15.
- [15] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song, “Parsec: a parallel simulation environment for complex systems,” *Computer*, vol. 31, no. 10, pp. 77–85, 1998.
- [16] R. Bagrodia, “Language support for parallel discrete-event simulations,” in *Proceedings of Winter Simulation Conference*, 1994.
- [17] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. USA: USENIX Association, 2005, p. 41.
- [18] F. Bellard, “SoftFP,” <https://bellard.org/softfp/>, 2018, accessed: 2025-04-15.
- [19] C. Bertin, N. Brisebarre, B. Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. Raina, and A. Tisserand, “A floating-point library for integer processors,” *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5559, 10 2004.
- [20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” *SIGARCH Comput. Archit. News*, 2011.
- [21] S. Boldo and J.-M. Muller, “Exact and approximated error of the FMA,” *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 157–164, 2010.
- [22] N. Bosbach, N. Zurstraßen, R. Pelke, L. Jünger, J. H. Weinstock, and R. Leupers, “Towards High-Performance Virtual Platforms: A Parallelization Strategy for SystemC TLM-2.0 CPU Models,” in *61st ACM/IEEE Design Automation Conference (DAC ’24)*, Jun. 2024.
- [23] Bosbach, Nils and Pelke, Rebecca, and Zurstraßen, Niko and Jünger, Lukas, and Weinstock, Jan Henrik and Leupers, Rainer, “Work-in-Progress: A Generic Non-Intrusive Parallelization Approach for SystemC TLM-2.0-based Virtual Platforms,” 2023, to appear in CODES+ISSS 2023.

- [24] A. Bradbury, "NaN Boxing ISA-Dev Group," https://groups.google.com/a/groups.riscv.org/g/isa-dev/c/_r7hBlzsEd8/m/z1rjr2BaAwAJ, 03 2017, accessed: 2025-04-15.
- [25] A. Bradbury, "NaN Boxing RFC," <https://gist.github.com/asb/a3a54c57281447fc7eac1eec3a0763fa>, 03 2017, accessed: 2025-04-15.
- [26] A. Brokalakis, N. Tampouratzis, A. Nikitakis, I. Papaefstathiou, S. Andrianakis, D. Pau, E. Plebani, M. Paracchini, M. Marcon, I. Sourdis, P. R. Geethakumari, M. C. Palacios, M. A. Anton, and A. Szasz, "COSSIM: An Open-Source Integrated Solution to Address the Simulator Gap for Systems of Systems," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 115–120.
- [27] R. E. Bryant, "Simulation of packet communication architecture computer systems," USA, Tech. Rep., 1977.
- [28] D. Burger and D. Wood, "Accuracy vs. performance in parallel simulation of interconnection networks," in *Proceedings of 9th International Parallel Processing Symposium*, 1995, pp. 22–31.
- [29] G. Busnot, T. Sassolas, N. Ventrux, and M. Moy, "Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020, pp. 363–368.
- [30] G. Busnot, T. Sassolas, N. Ventrux, and M. Moy, "Standard-compliant parallel SystemC simulation of loosely-timed transaction level models: From baremetal to Linux-based applications support," *Integration*, vol. 79, pp. 23–40, 2021.
- [31] Z. Cai, A. Liang, Z. Qi, L. Jiang, X. Li, H. Guan, and Y. Chen, "Performance Comparison of Register Allocation Algorithms in Dynamic Binary Translation," in *2009 International Conference on Knowledge and Systems Engineering*, 2009, pp. 113–119.
- [32] J. Caron, "Java: Status Report and Language Overview," *CSCI 5535 Project*, Dec. 1995, *University of Colorado at Boulder*, 12 1995.
- [33] J. Cavazos, S. Grauer-Gray, W. Killian, and R. Searles, "FinanceBench," <https://github.com/cavazos-lab/FinanceBench>, accessed: 2025-04-15.
- [34] O. Certner, Z. Li, A. Raman, and O. Temam, "A Very Fast Simulator for Exploring the Many-Core Future," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 443–454.
- [35] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0096055181900485>

- [36] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.
- [37] W. Chen, X. Han, and R. Dömer, "Out-of-order parallel simulation for ESL design," in *2012 Design, Automation and Test in Europe Conference (DATE)*, 2012, pp. 141–146.
- [38] Z. Cheng and R. Dömer, "Analyzing variable entanglement for parallel simulation of SystemC TLM-2.0 models," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.
- [39] Z. Cheng, T. Schmidt, and R. Dömer, "Systemc coding guideline for faster out-of-order parallel discrete event simulation," *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2018*, pp. 99–114, 2020.
- [40] "Whisper Github Repository," <https://github.com/chipsalliance/VeeR-ISS>, CHIPS Alliance, accessed: 2025-04-15.
- [41] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *Computational Science – ICCS*, 2006.
- [42] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *Computational Science–ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV* 6. Springer, 2006, pp. 653–660.
- [43] M.-K. Chung, J.-K. Kim, and S. Ryu, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 1472–1475.
- [44] M. Clark and B. Hoult, "rv8 - RISC-V simulator for x86-64," <https://github.com/michaeljclark/rv8>, accessed: 2025-04-15.
- [45] M. Clark and B. Hoult, "rv8: a high performance RISC-V to x86 binary translator," *CARRV*, 10 2017.
- [46] B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1994, pp. 128–137.
- [47] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing synchronization in a parallel SystemC kernel," in *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2008, pp. 180–187.
- [48] E. G. Cota and L. P. Carloni, "Cross-ISA Machine Instrumentation Using Fast and Scalable Dynamic Binary Translation," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser.

- VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 74–87. [Online]. Available: <https://doi.org/10.1145/3313808.3313811>
- [49] D. R. Cox, “RITSim: Distributed SystemC Simulation,” 2005.
- [50] J. Cubero-Cascante, N. Zurstraßen, J. Nöller, R. Leupers, and J. M. Joseph, “parti-gem5: gem5’s Timing Mode Parallelised,” in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Springer, 2023, pp. 177–192.
- [51] T. J. Dekker, “A floating-point technique for extending the available precision,” *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.
- [52] 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*, 2011, pp. 276–283.
- [53] I. Dooley, L. Kale, and N. Goodwin, “Quantifying the interference caused by subnormal floating-point values,” in *Proceedings of the workshop on operating system interference in high performance applications*, 2006.
- [54] R. Dömer, W. Chen, and X. Han, “Parallel discrete event simulation of Transaction Level Models,” in *17th Asia and South Pacific Design Automation Conference*, 2012, pp. 227–231.
- [55] R. Dömer, Z. Cheng, D. Mendoza, and E. M. Arasteh, *Pushing the Limits of Parallel Discrete Event Simulation for SystemC*. Springer International Publishing, 01 2021, pp. 97–105.
- [56] J. Eisl, M. Grimmer, D. Simon, T. Würthinger, and H. Mössenböck, “Trace-based register allocation in a jit compiler,” in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–11.
- [57] “CoreMark®-PRO,” <https://github.com/eembc/coremark-pro>, Embedded Microprocessor Benchmark Consortium, accessed: 2025-04-15.
- [58] J. Engblom, “Temporal Decoupling-Are ‘Fast’ and ‘Correct’ Mutually Exclusive?” in *Design and Verification Conference Europe (DVCon Europe)*, 2018.
- [59] J. Engblom, “How the Intel® Simics® Simulator Executes Instructions,” <https://community.intel.com/t5/Blogs/Products-and-Solutions/Software/How-the-Intel-Simics-Simulator-Executes-Instructions/post/1543049>, Intel Corporation, 2023, accessed: 2025-04-15.
- [60] M. A. Ertl and D. Gregg, “The Structure and Performance of Efficient Interpreters,” *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–25, 2003.

- [61] “iperf3 benchmark,” <https://software.es.net/iperf/>, ESnet, accessed: 2025-04-15.
- [62] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, “Parallelizing SystemC kernel for fast hardware simulation on SMP machines,” in *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 2009, pp. 80–87.
- [63] B. Falsafi and D. A. Wood, “Cost/Performance of a Parallel Computer Simulator,” in *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*, ser. PADS '94. Association for Computing Machinery, 1994. [Online]. Available: <https://doi.org/10.1145/182478.182587>
- [64] flacs, “Dolphin Progress Report: June 2015 - Optionally Emulate Accurate NaNs,” <https://dolphin-emu.org/blog/2015/07/01/dolphin-progress-report-june-2015/#40-6704-optionally-emulate-accurate-nans-by-flacs>, 2015, accessed: 2025-04-15.
- [65] A. Fog, “The microarchitecture of Intel, AMD, and VIA CPUs,” https://www.agner.org/optimize/instruction_tables.pdf, accessed: 2025-04-15.
- [66] A. Fog, “Parallel floating point exception tracking and NaN propagation,” https://www.agner.org/optimize/nan_propagation.pdf, 2024, accessed: 2025-04-15.
- [67] A. Frantzis and J. Barker, “glmark2,” <https://github.com/glmark2/glmark2>, accessed: 2025-04-15.
- [68] A. Frumusanu, “The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test,” <https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6>, 2020, accessed: 2025-04-15.
- [69] K. Fuchi, H. Tanaka, Y. Manago, and T. Yuba, “A Program Simulator by Partial Interpretation,” in *Proceedings of the Second Symposium on Operating Systems Principles*, ser. SOSP '69. New York, NY, USA: Association for Computing Machinery, 1969, p. 97–104. [Online]. Available: <https://doi.org/10.1145/961053.961092>
- [70] R. M. Fujimoto, “Parallel Discrete Event Simulation,” *Commun. ACM*, vol. 33, no. 10, p. 30–53, oct 1990. [Online]. Available: <https://doi.org/10.1145/84537.84545>
- [71] R. M. Fujimoto, R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, D. Nicol, and B. Unger, “Parallel discrete event simulation: The making of a field,” in *2017 Winter Simulation Conference (WSC)*, 2017, pp. 262–291.
- [72] S. Fujita, “aobench,” <https://github.com/syoyo/aobench>, accessed: 2025-04-15.

- [73] N. Gala and M. Karasek, "RISC-V Architecture Test SIG," <https://github.com/riscv-non-isa/riscv-arch-test>, accessed: 2025-04-15.
- [74] M. Galicia, F. Merchant, and R. Leupers, "A Parallel SystemC Virtual Platform for Neuromorphic Architectures," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2022, pp. 1–6.
- [75] G. Glaser, G. Nitsche, and E. Hennig, "Temporal decoupling with error-bounded predictive quantum control," in *2015 Forum on Specification and Design Languages (FDL)*, 2015, pp. 1–6.
- [76] Y.-C. Guo, W. Yang, J.-Y. Chen, and J.-K. Lee, "Translating the ARM Neon and VFP Instructions in a Binary Translator," *Softw. Pract. Exper.*, vol. 46, no. 12, dec 2016.
- [77] J. Gustafson and I. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, pp. 71–86, 06 2017.
- [78] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.
- [79] B. Haetzer and M. Radetzki, "A case study on message-based discrete event simulation for Transaction Level Modeling," in *FDL 2011 Proceedings*, 2011, pp. 1–8.
- [80] B. Haetzer and M. Radetzki, "Asynchronous Parallel Simulation with Transaction Events," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 242–249.
- [81] Z. Hao, L. Qian, H. Li, X. Xie, and K. Zhang, "A Parallel Logic Simulation Framework: Study, Implementation, and Performance," in *Proceedings of the 2010 Spring Simulation Multiconference*, 2010, pp. 1–10.
- [82] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "NumPy benchmarks," <https://github.com/numpy/numpy/tree/main/benchmarks>, accessed: 2025-04-15.
- [83] J. R. Hauser, "Berkeley SoftFloat," <https://github.com/ucb-bar/berkeley-softfloat-3>, 1996, accessed: 2025-04-15.
- [84] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "AGRA Uni Bremen RISC-VP," <https://github.com/agra-uni-bremen/riscv-vp>, accessed: 2025-04-15.

- [85] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1383762120300503>
- [86] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2002.
- [87] R. Himeno, "Himeno Benchmark," <https://github.com/kowsalyaChidambaram/Himeno-Benchmark>, accessed: 2025-04-15.
- [88] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 104–113.
- [89] D. G. Hough, "The IEEE Standard 754: One for the History Books," *Computer*, vol. 52, no. 12, pp. 109–112, 2019.
- [90] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, and G. Li, "Godson-3: A Scalable Multicore RISC Processor with x86 Emulation," *IEEE Micro*, vol. 29, no. 2, pp. 17–29, 2009.
- [91] Y. Hu, H. Jin, Z. Yu, and H. Zheng, "An Optimization Approach for QEMU," in *2009 First International Conference on Information Science and Engineering*, 2009, pp. 129–132.
- [92] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, "Scalably distributed SystemC simulation for embedded applications," in *2008 International Symposium on Industrial Embedded Systems*. IEEE, 2008, pp. 271–274.
- [93] "PowerPC User Instruction Set Architecture Book I Version 2.01," IBM, 2003.
- [94] "IEEE Standard for Binary Floating-Point Arithmetic," IEEE, 1985.
- [95] "IEEE Standard for Floating-Point Arithmetic," IEEE, 2008.
- [96] "IEEE Standard for Floating-Point Arithmetic," IEEE, 2019.
- [97] "Intel 4004 Characteristics," <https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html>, Intel Corporation, accessed: 2025-04-15.
- [98] "Introducing Intel Advanced Performance Extensions (Intel APX)," <https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html>, Intel Corporation, accessed: 2025-04-15.

- [99] “Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture,” Intel Corporation, 2016.
- [100] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When, “Exploring system performance using elastic traces: Fast, accurate and portable,” in *SAMOS*, 2016.
- [101] D. R. Jefferson, “Virtual Time,” *ACM Trans. Program. Lang. Syst.*, 1985.
- [102] D. Johnson, “Why is Rosetta 2 fast?” <https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/>, accessed: 2025-04-15.
- [103] JosJuice, “5.0-15330 - Raise Program Exceptions On Floating Point Exceptions,” <https://dolphin-emu.org/blog/2021/11/13/dolphin-progress-report-september-and-october-2021/#50-15330-raise-program-exceptions-on-floating-point-exceptions-by-josjuice>, 2021, accessed: 2024-04-03.
- [104] J. Joy, “Evaluating Temporal Decoupling in a Virtual Platform,” <https://www.diva-portal.org/smash/get/diva2:1530379/FULLTEXT01.pdf>, 2020.
- [105] M. Jung, F. Schnicke, M. Damm, T. Kuhn, and N. Wehn, “Speculative Temporal Decoupling Using fork(),” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1721–1726.
- [106] L. Jünger, A. Belke, and R. Leupers, “Software-defined Temporal Decoupling in Virtual Platforms,” in *2021 IEEE 34th International System-on-Chip Conference (SOCC)*, 2021, pp. 40–45.
- [107] L. Jünger, C. Bianco, K. Niederholtmeyer, D. Petras, and R. Leupers, “Optimizing Temporal Decoupling using Event Relevance,” in *ASP-DAC*, 2021.
- [108] L. Jünger, J. H. Weinstock, and R. Leupers, “SIM-V: Fast, Parallel RISC-V Simulation for Rapid Software Verification,” *DVCON Europe 2022*, 2022.
- [109] L. Jünger, S. Winther, and R. Leupers, “X-on-x: Distributed parallel virtual platforms for heterogeneous systems,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 142–148.
- [110] W. M. Kahan and C. Severance, “An Interview with the Old Man of Floating-Point,” <https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html>, 1998, accessed: 2025-04-15.
- [111] L. Kaouane, D. Houzet, and S. Huet, “SysCellC: Systemc on cell,” in *2008 International Conference on Computational Sciences and Its Applications*. IEEE, 2008, pp. 234–244.

- [112] K. L. Kapp, T. C. Hartrum, and T. S. Wailes, "An Improved Cost Function for Static Partitioning of Parallel Circuit Simulations Using a Conservative Synchronization Protocol," in *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, ser. PADS '95. USA: IEEE Computer Society, 1995, p. 78–85. [Online]. Available: <https://doi.org/10.1145/214282.214306>
- [113] R. S. Khaligh and M. Radetzki, "Modeling Constructs and Kernel for Parallel Simulation of Accuracy Adaptive TLMs," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 1183–1188.
- [114] A. Khan, W. Ma, B. Werner, and C. Wolf, "Multi-threaded Simics SystemC Virtual Platform," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 373–379.
- [115] A. Klaiber *et al.*, "The Technology Behind Crusoe Processors," *Transmeta Technical Brief*, pp. 77–84, 2000.
- [116] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr, "HySim: A Fast Simulation Framework for Embedded Software Development," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 75–80.
- [117] M. Kretz, "Merge data-parallel types from the Parallelism TS 2," <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1928r3.pdf>, 2023, accessed: 2025-04-15.
- [118] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, 1985.
- [119] K. P. Lawton, "Bochs: A Portable PC Emulator For Unix/X," *Linux Journal*, vol. 1996, no. 29es, pp. 7–es, 1996.
- [120] Lazer, "StackOverflow - What's the difference between emulation and simulation?" <https://stackoverflow.com/questions/2174638/whats-the-difference-between-emulation-and-simulationdifference>, 2010, accessed: 2025-04-15.
- [121] X. Liao and T. Srikanthan, "Accelerating UNISIM-Based Cycle-Level Microarchitectural Simulations on Multicore Platforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, jun 2011. [Online]. Available: <https://doi.org/10.1145/1970353.1970359>
- [122] "LoongArch Reference Manual Volume 1: Basic Architecture Version 1.10," Loongson Technology Corporation Limited.
- [123] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom,

- B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kanno, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 Simulator: Version 20.0+," 2020.
- [124] "MachineWare SIM-V," <https://www.machineware.de/>, MachineWare, accessed: 2025-04-15.
- [125] "Virtual Components Modeling Library (VCML)," <https://github.com/machineware-gmbh/vcml>, MachineWare, accessed: 2025-04-15.
- [126] I. Maia Pessoa, A. Mello, A. Greiner, and F. Pêcheux, "Parallel TLM simulation of MPSoC on SMP workstations: Influence of communication locality," in *2010 International Conference on Microelectronics*, 2010, pp. 359–362.
- [127] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.
- [128] J. D. McCalpin, "STREAM benchmark," <https://www.cs.virginia.edu/stream/>, accessed: 2025-04-15.
- [129] S. Meftali and J.-L. Dekeyser, "An optimal charge balancing model for fast distributed SystemC simulation in IP/SoC design," in *4th IEEE International Workshop on System-on-Chip for Real-Time Applications*. IEEE, 2004, pp. 55–58.
- [130] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 606–609.
- [131] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, p. 21–65, feb 1991. [Online]. Available: <https://doi.org/10.1145/103727.103729>
- [132] D. Mihocka and S. Shwartsman, "Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure," in *Proceedings of the Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2008, pp. 55–70.

- [133] B. Miller and R. Pozo, "SciMark 2.0," <https://math.nist.gov/scimark2/>, accessed: 2025-04-15.
- [134] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.
- [135] "MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual Revision 6.05," MIPS, 2016.
- [136] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 153–162.
- [137] O. Møller, "Quasi Double-Precision in Floating Point Addition," *BIT*, vol. 5, no. 1, pp. 37–50, mar 1965.
- [138] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.
- [139] M. Moulin, "smallpt," <https://github.com/matt77hias/smallpt>, accessed: 2025-04-15.
- [140] M. Moy, "Parallel programming with SystemC for loosely timed models: A non-intrusive approach," in *2013 Design, Automation and Test in Europe Conference (DATE)*, 2013, pp. 9–14.
- [141] J.-M. Muller, N. Brisebarre, F. Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*, 01 2010.
- [142] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 149–154.
- [143] NASA, "NAS Parallel Benchmarks," <https://www.nas.nasa.gov/software/npb.html>, accessed: 2025-04-15.
- [144] "OSCI TLM-2.0 Language Reference Manual," https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf, Open SystemC Initiative (OSCI), accessed: 2025-04-15.
- [145] "OpenCores Ethernet MAC 10/100 Mbps," <https://opencores.org/projects/ethmac>, OpenCores, accessed: 2025-04-15.

- [146] "OpenRISC 1000 Architecture Manual Architecture Version 1.4," <https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.4-rev0.pdf>, OPENRISC.IO, 2022, accessed: 2025-04-15.
- [147] A. Over, B. Clarke, and P. Strazdins, "A Comparison of Two Approaches to Parallel Simulation of Multiprocessors," *Performance Analysis of Systems and Software, IEEE International Symposium on*, vol. 0, pp. 12–22, 04 2007.
- [148] I. Painter Engineering, "whetstone," <https://netlib.org/benchmark/whetstone.c>, accessed: 2025-04-15.
- [149] "Passmark Single Thread Performance," <https://www.cpubenchmark.net/singleThread.html>, PassMark Software Pty Ltd, accessed: 2025-04-15.
- [150] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A Full System Simulator for Multicore x86 CPUs," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 1050–1055.
- [151] S. Pees, A. Hoffmann, and H. Meyr, "Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 4, pp. 815–834, Oct. 2000. [Online]. Available: <http://doi.acm.org/10.1145/362652.362662>
- [152] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne, "A SystemC TLM Framework for Distributed Simulation of Complex Systems with Unpredictable Communication," in *Proceedings of the 2011 Conference on Design Architectures for Signal Image Processing (DASIP)*, 2011, pp. 1–8.
- [153] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, 2003.
- [154] M. Perotti, G. Tagliavini, S. Mach, L. Bertaccini, and L. Benini, "RVfplib: A Fast and Compact Open-Source Floating-Point Emulation Library for Tiny RISC-V Processors," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2022, pp. 16–32.
- [155] M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.
- [156] M. Probst, "Dynamic Binary Translation," in *UKUUG Linux Developer's Conference*, vol. 2002, 2002.
- [157] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," *Proceedings Linux Symposium*, vol. 15, 01 2007.

- [158] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020. [Online]. Available: <https://doi.org/10.1145/3422667>
- [159] "Spike RISC-V ISA Simulator," <https://github.com/riscv-software-src/riscv-isa-sim>, RISC-V Foundation, accessed: 2025-04-15.
- [160] *The RISC-V Instruction Set Manual*, RISC-V Foundation, 2019. [Online]. Available: <https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf>
- [161] *RISC-V ABIs Specification*, RISC-V Foundation, 2023.
- [162] C. Roth, S. Reder, O. Sander, M. Hübner, and J. Becker, "A Framework for Exploration of Parallel SystemC Simulation on the Single-Chip Cloud Computer," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS '12. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012, pp. 202–207.
- [163] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "VSim: Simulating Multi-Server Setups at near Native Hardware Speed," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, jan 2012. [Online]. Available: <https://doi.org/10.1145/2086696.2086731>
- [164] R. Salimi Khaligh and M. Radetzki, "Efficient Parallel Transaction Level Simulation by Exploiting Temporal Decoupling," in *Analysis, Architectures and Modelling of Embedded Systems*, A. Rettberg, M. C. Zanella, M. Amann, M. Keckeisen, and F. J. Rammig, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 149–158.
- [165] G. Sarrazin, N. Brunie, and F. Pétrot, "Virtual Prototyping of Floating Point Units," in *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2016, pp. 1–6.
- [166] C. Sauer, H.-M. Bluethgen, and H.-P. Loeb, "Distributed, loosely-synchronized SystemC/TLM simulations of many-processor platforms," in *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, vol. 978-2-9530504-9-3, 2014, pp. 1–8.
- [167] T. Schmidt, G. Liu, and R. Dömer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.
- [168] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 241–246.

- [169] C. Schumacher, J. H. Weinstock, R. Leupers, G. Ascheid, L. Tosoratto, A. Lonardo, D. Petras, and T. Grötter, “legaSCi: Legacy SystemC Model Integration into Parallel Systemc Simulators,” in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 2188–2193.
- [170] E. Schwarz, M. Schmookler, and S. Trong, “FPU Implementations with Denormalized Numbers,” *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 825–836, 2005.
- [171] S. Seo, G. Jo, and J. Lee, “Performance Characterization of the NAS Parallel Benchmarks in OpenCL,” in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 137–148.
- [172] R. Sinha, A. Prakash, and H. D. Patel, “Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs,” in *17th Asia and South Pacific Design Automation Conference*, 2012, pp. 455–460.
- [173] S. Sirowy, C. Huang, and F. Vahid, “Dynamic acceleration management for SystemC emulation,” *ACM SIGBED Review*, vol. 6, no. 3, pp. 1–4, 2009.
- [174] S. P. E. C. (SPEC), “SPEC CPU 2017,” <https://spec.org/cpu2017/>, accessed: 2025-04-15.
- [175] P. H. Sterbenz, *Floating Point Computation*, G. Forsythe, Ed. Prentice-Hall, 1974.
- [176] N. Tampouratzis, I. Papaefstathiou, A. Nikitakis, A. Brokalakis, S. Andrianakis, A. Dollas, M. Marcon, and E. Plebani, “A novel, highly integrated simulator for parallel and distributed systems,” *ACM Trans. Archit. Code Optim.*, vol. 17, no. 1, mar 2020. [Online]. Available: <https://doi.org/10.1145/3378934>
- [177] S. Thakkur and T. Huff, “Internet Streaming SIMD Extensions,” *Computer*, vol. 32, no. 12, pp. 26–34, 1999.
- [178] J. Tsiombikas, “c-ray,” <https://github.com/jtsiomb/c-ray>, accessed: 2025-04-15.
- [179] “Wii VC Round-to-Zero,” https://ukikipedia.net/wiki/Wii_VC_Round-to-Zero, Ukikipedia, accessed: 2025-04-15.
- [180] UncommentatedPannen, “Bowser in the Fire Sea without Red Coins 0xA (Wii VC Only),” <https://www.youtube.com/watch?v=TiUwJSOCbYE>, accessed: 2024-04-03.
- [181] “Octane 2.0,” <https://github.com/chromium/octane>, V8 project authors, accessed: 2025-04-15.
- [182] N. Ventrux, J. Peeters, T. Sassolas, and J. C. Hoe, “Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations,” in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 250–257.

- [183] N. Ventroux and T. Sassolas, "A New Parallel SystemC Kernel Leveraging Manycore Architectures," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 487–492.
- [184] E. Viaud, F. Pecheux, and A. Greiner, "An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, 2006, pp. 1–6.
- [185] S. Vinco, V. Bertacco, D. Chatterjee, and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures," in *DAC Design Automation Conference 2012*, 2012, pp. 115–120.
- [186] J. Virtanen, P. Sjövall, M. Viitanen, T. D. Härmäläinen, and J. Vanne, "Distributed SystemC Simulation on Manycore Servers," in *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, 2016, pp. 1–6.
- [187] J. Walker, "fbench," <https://www.fourmilab.ch/fbench/fbench.html>, accessed: 2025-04-15.
- [188] J. Walker, "ffbench," <https://www.fourmilab.ch/fbench/ffbench.html>, accessed: 2025-04-15.
- [189] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili, "Manifold: A Parallel Simulation Framework for Multicore Systems," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 106–115.
- [190] J. Wang, J. Pang, L. Fu, Z. Shan, F. Yue, and J. Zhang, "A binary translation back-end registers allocation algorithm based on priority," in *Geo-Spatial Knowledge and Intelligence: 5th International Conference, GSKI 2017, Chiang Mai, Thailand, December 8-10, 2017, Revised Selected Papers, Part II 5*. Springer, 2018, pp. 414–425.
- [191] A. Waterman, *Design of the RISC-V Instruction Set Architecture*. University of California, Berkeley, 2016.
- [192] A. Waterman, "NaN Boxing Github Issue," <https://github.com/riscv/riscv-isa-manual/issues/30>, 03 2017, accessed: 2025-04-15.
- [193] J. Weinstock, "OpenRISC 1000 Instruction Set Simulator (or1kiss)," <https://github.com/janweinstock/or1kiss>, 2018.
- [194] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segment," in *2016 Design, Automation and Test in Europe Conference (DATE)*, 2016, pp. 493–498.
- [195] J. H. Weinstock, L. G. Murillo, R. Leupers, and G. Ascheid, "Parallel SystemC Simulation for ESL Design," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 1, pp. 1–25, 2016.

- [196] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-decoupled parallel SystemC simulation," in *2014 Design, Automation and Test in Europe Conference (DATE)*, 2014, pp. 1–4.
- [197] E. Witchel and M. Rosenblum, "Embra: Fast and Flexible Machine Simulation," in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1996, pp. 68–79.
- [198] L. Yi, Y. Shao, G. Yang, and J. Wu, "Register Allocation for QEMU Dynamic Binary Translation Systems," *International Journal of Hybrid Information Technology*, vol. 8, pp. 199–210, 02 2015.
- [199] Y.-P. You, T.-C. Lin, and W. Yang, "Translating AArch64 Floating-Point Instruction Set to the X86-64 Platform," in *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, 2019.
- [200] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 78–.
- [201] J. Zhu and D. D. Gajski, "A Retargetable, Ultra-fast Instruction Set Simulator," in *Proceedings of the conference on Design, automation and test in Europe*, 1999, pp. 62–es.
- [202] J. Zhu and D. D. Gajski, "An Ultra-Fast Instruction Set Simulator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 363–373, 2002.
- [203] V. Zivojnovic, S. Tijang, and H. Meyr, "Compiled Simulation of Programmable DSP Architectures," in *VLSI Signal Processing, VIII*, 1995, pp. 187–196.
- [204] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun, "A Parallel SystemC Environment: ArchSC," in *2009 15th International Conference on Parallel and Distributed Systems*. IEEE, 2009, pp. 617–623.
- [205] N. Zurstraßen, R. Brandhofer, J. Cubero-Cascante, N. Bosbach, L. Jünger, and R. Leupers, "The art of temporal decoupling," *Integration*, vol. 101, p. 102314, 2025.
- [206] N. Zurstrassen, L. M. Reimann, N. Bosbach, L. Juenger, and R. Leupers, "Evaluation of the RISC-V Floating Point Extensions," in *DVCon Europe 2023; Design and Verification Conference and Exhibition Europe*, 2023, pp. 57–64.
- [207] N. Zurstraßen, "TLMBoy," <https://github.com/not-chciken/TLMBoy>, 2020.
- [208] N. Zurstraßen, "Github Issue - Validity of Fused Multiply-Add Function," <https://github.com/ucb-bar/berkeley-softfloat-3/issues/33>, 2025.

- [209] N. Zurstraßen, N. Bosbach, J. M. Joseph, L. Jünger, J. H. Weinstock, and R. Leupers, “Efficient RISC-V-on-x64 Floating Point Simulation,” in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, 2023, pp. 1–6.
- [210] N. Zurstraßen, N. Bosbach, L. Jünger, and R. Leupers, “Rapid RISC-V Floating Point Vector Simulation,” 2025.
- [211] N. Zurstraßen, N. Bosbach, and R. Leupers, “FloppyFloat: An Open Source Floating Point Library for Instruction Set Simulators,” 2025.
- [212] N. Zurstraßen, N. Bosbach, L. M. Reimann, and R. Leupers, “Static Global Register Allocation For Dynamic Binary Translators,” 2025.
- [213] N. Zurstraßen, R. Brandhofer, J. Cubero-Cascante, N. Bosbach, L. Jünger, and R. Leupers, “The Optimal Quantum of Temporal Decoupling,” in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 686–691.
- [214] N. Zurstraßen, J. Cubero-Cascante, J. M. Joseph, L. Yichao, X. Xinghua, and R. Leupers, “par-gem5: Parallelizing gem5’s Atomic Mode,” in *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2023, pp. 1–6.
- [215] N. Zurstraßen, L. Jünger, T. Kogel, H. Keding, and R. Leupers, “AMAIX In-Depth: A Generic Analytical Model for Deep Learning Accelerators,” *International Journal of Parallel Programming*, 2022.

Curriculum Vitae

Name Niko Zurstraßen

11.09.2025 Doctoral Examination for Dr.-Ing.

2020-2025 Research Assistant and Doctoral Candidate, Institute
for Communication Technologies and Embedded
Systems, RWTH Aachen University, Germany

2013-2020 Studies of Electrical Engineering, Information
Technology and Computer Engineering, RWTH
Aachen University, Germany

April 2020 Master of Science

April 2017 Bachelor of Science

2005-2013 General Higher Education Entrance Qualification
(Abitur) Pascal-Gymnasium, Münster, Germany

1993 Born in Münster, Germany

