Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

A hybrid RISC-V architecture supporting mixed timing-critical and high performance workloads

Bei der Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Mehrdad Poorhosseini
geboren am 06.01.1991 in Mashhad, Iran

Mehrdad Poorhosseini: *A hybrid RISC-V architecture supporting mixed timing-critical and high performance workloads*

Gutachter:
Prof. Dr.-Ing. Wolfgang Nebel

Weiterer Gutachter:
Prof. Dr.-Ing. Jan Reineke

Tag der Disputation:
13.02.2023

Abstract

The hardware platforms available today for embedded systems are already capable of implementing different classes of applications. These can be real-time applications, in which compliance with given time limits must be guaranteed, and high-performance applications, in which the aim is to execute as many instructions as possible per unit of time. Existing hardware platforms are usually designed and optimized for either the class of real-time applications or the class of high-performance requirements. However, if a mix of real-time and high-performance applications is to be executed on the same platform, this either causes a great deal of effort with regard to real-time verification or the platform does not meet the minimum requirements for the high-performance application.

The goal of this work is to propose a hybrid (i.e., run-time switchable) hardware platform that is capable of executing the above two classes of applications without suffering from mutually negative timing predictability and execution time optimization requirements.

This work presents a set of design requirements for such a hybrid hardware platform for embedded systems. Based on these requirements, various existing single- and multi-core platforms, ranging from high-performance embedded architectures to fully time-predictable architectures, are analyzed and compared. Based on this analysis, a new hybrid HW/SW architecture is proposed that can switch between a real-time and high-performance execution mode at runtime. In addition, this work describes the integration and implementation in an FPGA, based on an open-source RISC-V processor system and FreeRTOS as the SW management layer. Using an integrated measurement infrastructure, an analysis of software functionality, software execution timing, and switching times is performed in a single-core FPGA implementation of the proposed hybrid architecture.

Kurzzusammenfassung

Die heute verfügbaren Hardwareplattformen für eingebettete Systeme sind bereits in der Lage unterschiedliche Klassen von Anwendungen zu realisieren. Dies können zum einen Echtzeitanwendungen sein, bei denen die Einhaltung gegebener Zeitschranken garantiert werden muss, und zum andere Hochleistungsanwendungen, bei denen es darum geht möglichst viele Instruktionen pro Zeiteinheit auszuführen. Existierende Hardwareplattformen sind in der Regel entweder für die Klasse der Echtzeitanwendungen oder die Klasse der Hochleistungsanforderungen ausgelegt und optimiert. Soll nun aber ein Mix aus Echtzeit- und Hochleistungsanwendungen auf derselben Plattform ausgeführt werden, dann verursacht das entweder einen großen Aufwand bzgl. des Echtzeitnachweises oder die Plattform erfüllt nicht die Minimalanforderungen für die Hochleistungsanwendung.
Ziel dieser Arbeit ist es, eine hybride (d.h. zur Laufzeit umschaltbare) Hardwareplattform vorzuschlagen, die in der Lage ist, die beiden o.g. Anwendungsklassen auszuführen, ohne dass diese unter den sich wechselseitig negativ beeinflussenden Anforderungen an die Zeitvorhersagbarkeit und Ausführungszeitoptimierung leiden.
Diese Arbeit stellt eine Reihe von Designanforderungen an eine solche hybride Hardwareplattform für eingebettete Systeme vor. Basierend auf diesen Anforderungen werden verschiedene existierende Single- und Multi-Core-Plattformen, von eingebetteten Hochleistungsarchitekturen, bis hin vollständig zeitvorhersagbarer Architekturen, analysiert und miteinander verglichen. Basierend auf dieser Analyse wird eine neue hybride HW/SW-Architektur vorgeschlagen, welche zur Laufzeit zwischen einem Echtzeit- und Hochleistungsausführungsmodus umschalten kann. Zusätzlich beschreibt diese Arbeit einen Ansatz zur Integration und Implementierung in einem FPGA, basierend auf einem Open-Source RISC-V Prozessorsystems und FreeRTOS als SW-Verwaltungsschicht. Mit Hilfe einer integrierten Messinfrastruktur wird eine Analyse der Funktionalität der Software, dem Softwareausführungszeitverhalten und der Umschaltzeiten

in einer Single-Core FPGA Implementierung der vorgeschlagenen hybriden Architektur durchgeführt.

Acknowledgement

I would like to thank my supervisor Prof. Dr.-Ing. Wolfgang Nebel for the continuous support of my Ph.D. studies and related research, for his patience, motivation, and immense knowledge. I would like to thank Dr. Kim Grüttner, in the Hardware/Software Design Methodology group of OFFIS institute, who provided detailed scientific feedback, gave me adequate research freedom, and helped me become an independent researcher. I thank Prof. Dr. Martin Georg Fränzle who was always ready for scientific discussions and interested in hearing an update on my progress. I could not have imagined a better supervision team for my Ph.D.

I would like to express my deep and sincere gratitude to Ira Wempe, who has given me suggestions, advice, and support in non-scientific works. I also want to thank Moritz Brähler, Stephan Adolf, and Razi Seyyedi for the stimulating discussions and all the fun we have had over the past four years.

I am fortunate to have family and friends who have always supported me in every decision. Thanks to my close friends who stood by me during these years: Vahid, Amir, Payam, Sasan, Danial, Iman, and Kamran. I am infinitely grateful to my parents and sister for their understanding, motivation, and all the opportunities they gave me. Last but not least, I thank my beloved wife Yasaman for her sincere love, patience, and optimism even in difficult moments; without her endless support, this dissertation would not have been possible.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AXI	Advanced eXtensible Interface
BCET	Best-Case Execution Time
CDF	Cumulative Distribution Function
CLINT	core level interrupt controler
COTS	Commercially available Off-The-Shelf
CPS	Cyber-Physical System
CPU	Central Processing Unit
CSR	Control and Status Register
DDR	Double Data Rate
DMA	Direct Memory Access
DMEM	Data Memory
DRAM	Dynamic RAM
FPGA	Field Programmable Gate Array

FPU	Floating Point Unit
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HP	High-Performance
HPC	High-Performance Computing
HRT	Hard Real-Time
IMEM	Instruction Memory
IoT	Internet of Thing
ISA	instruction set architecture
MMU	Memory Management Unit
MPSoC	Multiprocessor System on a Chip
NoC	Network-on-Chip
OS	Operating System
PL	Programmable Logic
PREM	PRedictable Execution Model
PS	Processing System

RISC-V	Fifth Generation of Reduced Instruction Set Computer
RT	Real-Time
SDRAM	Synchronous DRAM
SPM	Scratchpad Memory
SRAM	Static Random Access Memory
SRT	Soft Real-Time
TDM	Time-Division Multiplexing
WCET	Worst-Case Execution Time

Chapter 1

Introduction

This chapter provides an overview of the thesis, motivating the necessity for supporting high-performance and time-predictable embedded computing in one platform. Then the scope and research questions will be introduced. Finally, it will be explained how we cover the related contents in the rest of the thesis.

1.1 Motivation

In today's embedded system design, developers try to reduce cost, size, and power consumption. In order to have an efficient model in mixed-criticality systems, consisting of different tasks with different levels of assurance in one platform, we need to propose a dynamic switching between different execution modes to ease the integration of real-time and non-real-time workloads on the same system [2]. In Cyber-Physical System (CPS), where computation interacts with physical processes, the design of timing predictable architecture plays an important role because delivering the correct information to the physical world in time is essential.

We could observe that current available embedded platforms in industry and academia are mainly optimized for timing predictability or High-Performance (HP) computation. There are some reasons for these optimizations. For instance, we could consider a workload consisting of some Real-

Time (RT) tasks that are highly timing critical, and it should be guaranteed that RT tasks finish within the desired timing. On the other hand, we could also design a processor to meet the requirements associated with application performance, such as fast response time and processing speed. Accordingly, there is a trade-off between typical case performance and predictable systems, which means the more the design is optimized in one direction, the more we lose features in the other direction. Therefore, if we have embedded high-performance requirements, many platforms are typically available. If we have tight real-time constraints, typically, there are other platforms available, and we can separate these systems from each other. The motivation of this project is to propose an architecture that can support both of them in a single-core architecture.

1.2 Scope and Research Questions

The scope of this work aims to investigate how we can get the best from both timing predictable software execution (usually supported by the simple microcontroller without cache) and high-performance software execution (usually supported by larger processors with a complex microarchitecture and several cache hierarchies). The main focus of the project is to propose a single processor architecture in the RISC-V ecosystem, which is based on a high-performance embedded processor core formerly named Ariane [3, 4] that can be switched at run-time from the high-performance to a real-time mode and vice versa. In the future, the final goal is to make this architecture applicable also in a multi-core environment.

In approaching an embedded processor, this thesis will focus on improving the timing predictable execution capabilities of a high-performance embedded single-core platform. Therefore, the overall goal is to execute real-time software with minimum speed overhead compared to non-real-time software on the same hardware platform to yield predictability as much as possible. Figure 1.1 depicts the scope of this research work. We have visualized the scope using circles. So we have Central Processing Unit (CPU) on top, and predictable hardware (PR HW) belongs to the CPU as a processing part

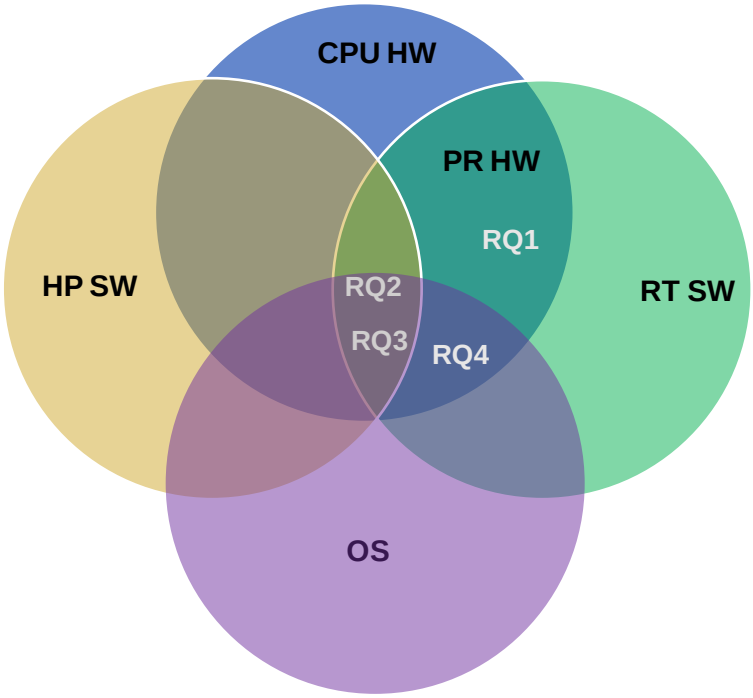

Figure 1.1: Scope of this thesis.

for executing real-time software. We have real-time software (RT SW) and high-performance software (HP SW) on the right and left, respectively, eventually, as software support operating system (OS) on the bottom. Hence, each research question can be positioned in one of those circles or between the overlapping zones.

We formulate the following scientific questions regarding designing a switchable platform in the RISC-V ecosystem

RQ1 What are the microarchitecture and memory subsystems that make timing predictability difficult?

RQ2 How can timing-predictability and high-performance computation be supported on a single-core architecture by switching modes?

RQ3 How can we benefit from an Operating System that allows switching between Real-time (RT) and High-Performance (HP) mode at runtime?

RQ4 What overheads do these hardware and software changes have on overall execution time?

1.3 Thesis Organization

This thesis is organized into eight chapters. The second chapter provides an overview of the foundations for this thesis. Once the basics are set, Chapter 3 discusses related scientific work covering application class and real-time class processors. At the end of this chapter, the existing gap will be covered. Afterward, Chapter 4 contains the presentation of contributions and constraints. Next, in Chapter 5, the concept is presented, which starts with the application and execution model definitions. Once the model is presented, two possible example executions which could be directly extracted from the concept will be discussed. The chapter concludes with an overview of the switchable architecture design and the memory hierarchy of the concept implementation. The proposed mode switchable architecture is then implemented as described in Chapter 6. Chapter 7 of this thesis contains the evaluation of the contributions and discusses the results. Chapter 8 concludes the thesis and discusses possible future activities. Finally, Appendices presents the measurement results for all executed benchmarks.

Chapter 2

Background and Foundations

This chapter includes essential information about the embedded system, the real-time embedded world, architectural elements that affect the predictability of a system, and some information about the RISC-V.

Therefore, in Section 2.1 preliminaries of embedded systems, and the connection between embedded systems and real-time computing will be explained. The timing predictability of a system and Worst-Case Execution Time (WCET) analysis as the key design requirements for real-time embedded systems are illustrated in Sections 2.1.2 and 2.1.3. Afterward, hybrid computer architecture design will be characterized by explaining the requirements that a system should meet to call a system suitable for high-performance and real-time computing. Then, we describe the RISC-V instruction set architecture (ISA) and the differences from other available ISAs. Finally, some information about available compilers and cores in the RISC-V ecosystem will complete Chapter 2.

2.1 Embedded systems

Embedded systems are microprocessor-based information processing systems designed to perform certain routines of specific functions repeatedly. Embedded systems, as small and intelligent electronic systems, have become a key technological factor for complex systems as widely used, ranging from com-

mercial electronics such as cell phones to critical infrastructures such as factory production lines and intelligence systems [5, 6, 7]. Compared to general-purpose computers, embedded systems perform limited dedicated functions with limited computing capability and power sources. Today, around 95% of all innovations are driven and controlled by embedded electronics components and software [8]. Many of the current embedded systems are required to operate in dynamic environments, where the characteristics of the computational workload cannot consistently be predicted in advance [9].

2.1.1 Real-time Embedded Systems

In general, we can classify the real-time embedded systems into two categories hard and soft real-time embedded systems. For Hard Real-Time (HRT) embedded systems, tasks must execute within a particular deadline, while in contrast, for Soft Real-Time (SRT) embedded systems, minor deadline violations will not be considered as system failure. In addition, a Scheduling policy is considered one of the main factors affecting real-time embedded systems performance. It helps to choose which task should be selected first from a queue ready to run [5]. Embedded systems that are subject to responses to the event within the precise timing constraints and predictable execution are named real-time embedded systems [10].

There are two categories of real-time tasks in real-time embedded systems. 1. Aperiodic tasks and 2. Periodic tasks. In each category, two types of tasks exist, which are: A) Preemptive: where a process "task" is blocked or interrupted by another process that has a higher priority, and B) Non-preemptive: any task completes its execution cycle even though there is another task with higher priority in the ready queue [11]. These task definitions are essential since each task could be one of the mentioned categories or subcategories in an execution flow that includes different tasks.

2.1.2 Timing predictability concept in an embedded system

As explained in [12], there are three different kinds of predictable systems. Simple systems with a fixed worst-case response time. In systems with only periodic tasks, the designer could choose a fixed cyclic schedule for all tasks in order to facilitate the computation. And finally, the ability to realize complex computation and react to timing relevant aspects in a pre-defined and nondeterministic way with tight and loose time constraints requires complex software scheduling algorithms. In cyber-physical real-time embedded systems, developers try to design a hardware platform that is able to integrate different levels of criticality, from highly critical workloads to high-performance workloads. However, it is challenging to design a processor that supports different levels, such as critical and non-critical workloads.

Characterizing upper and lower bounds of execution times is required to perform schedulability analysis. Therefore, a reasonable quality measure is the quotient of Best-Case Execution Time (BCET) over WCET [13]. The sources of uncertainty are the program input and the hardware state in which execution begins. Figure 2.1 illustrates the situation and displays essential notions. Accordingly, based on [14] here we can provide a formal definition of timing predictability:

Definition 1. (Timing Predictability). Given the uncertainty about the initial hardware states $Q \subseteq Q$ and uncertainty about the program inputs $I \subseteq \iota$, the timing predictability of a program p is

$$\Pr_p(Q,I) := \min_{q_1,q_2 \in Q} \min_{i_1,i_2 \in I} \frac{T_p(q_1,i_1)}{T_p(q_2,i_2)}$$

where Q denotes the set of all hardware states and I denotes the set of all program inputs. Furthermore, $T_p(q,i)$ be the execution time of program p starting in hardware state $q \in Q$ with input $i \in I$. The quality measure quotient is $\Pr_p \in [0,1]$, where 1 means surely predictable.

An informal definition of timing predictability is the ability to calculate the execution time on a specific hardware platform and prove that the system

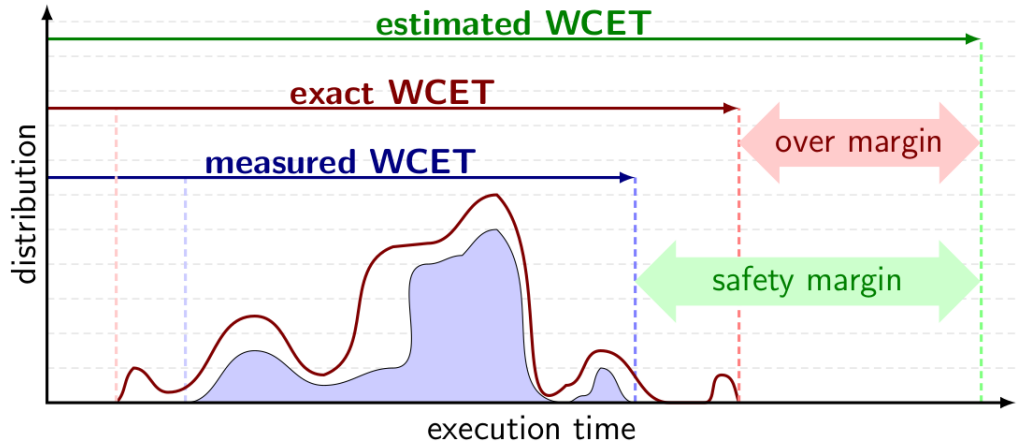

Figure 2.1: Example distribution of execution time ranging from best-case to worst-case execution time (BCET/WCET)(figure from [1]).

fulfills all requirements concerning workloads [12, 15]. In other words, a predictable execution time means that in a set of periodically repeated tasks, we could accurately predict the execution time of all tasks, regardless of how often the experiments were repeated. Therefore, running experiments with the same program under similar conditions lead to the same results. In combination with timing predictability, we require real-time analysis before using the system under consideration.

2.1.3 Worst-Case Execution Time analysis

WCET analysis could be performed in many ways using different tools. Two of them are explained here:

Measurement-based WCET analysis is suitable for less time-critical software and for which the average-case behavior is more significant than a precise estimate like, for instance, in systems where the worst-case scenario is improbable to occur. The traditional and most common method in the industry to determine program timing is by measurements. The basic principle of this method follows the statement that "the processor is the best hardware model" [16].

Static WCET analysis could be performed in different phases, from flow analysis to low-level phase. There are some advantages of using static analysis techniques that rely on mathematical models, such as eliminating the need to set up a real hardware platform and calculating the safe WCET upper-bounds without running the program on the target platform. Static analysis typically uses models of all the hardware components and their arbitration policies, including CPU caches, instruction pipeline, memory bus, arbitration policies, etc. These models are typically represented in complex mathematical abstractions for which a worst-case operation can be estimated [16].

2.2 Hybrid embedded computer architecture design

Traditionally, High-Performance Computing (HPC) and Embedded Computing have different design, programmability, and energy efficiency objectives. However, on the one hand, due to the ever-increasing demand for performance and the need to support more robust applications - such as, e.g., smart video surveillance and, more in general, mobile and Cyber-Physical System applications - those objectives tend to adopt similar solutions [17, 18]. On the other hand, the high demands on embedded systems products put pressure on the designers to maximize the system performance and minimize the prices [19].

2.2.1 Architectural elements

Single-core and multi-core devices developed using Commercially available Off-The-Shelf (COTS) components have become the preferred choice in embedded systems design. The reason is that COTS components are optimized for faster average case computation by increasing the design complexity. Nevertheless, as soon as it comes to a real-time computation where we have to meet the deadline and require a more straightforward timing analysis, these elements can cause trouble [20]. In order to improve the predictability of

a COTS system, we need to replace some architectural elements by predictable versions. Some real-time researchers have proposed several solutions to improve the predictability in COTS components e.g. Pellizzoni et al. [21] introduced the PRedictable Execution Model (PREM), which enforces predictable execution for some tasks by annotating a code called by getting benefits from a compiler extension. This technique requires support from both compiler and Operating System (OS).

It is clearly evident that there is a conflict between performance and predictability. The more we achieve performance, the more we lose predictability. Here we explain the architectural elements that make predictability difficult while on the other side improve the performance.

Pipeline: Pipelines increase the performance of the processor by executing different instructions in parallel with little hardware cost. Hence, depending on the employed pipeline features, the complexity of WCET analysis varies. For example, dynamic branch prediction and deeper pipeline design bring about more dependencies and influence both execution time and nondeterminism in terms of predictability.

Cache hierarchy (L1 and L2): Analysis of modern caches which designed for high-performance multi-core systems is extremely challenging due to the replacement policy and high set-associativity. It is necessary to determine the cache latencies, but usually, the latency provided by companies like ARM and Intel could not be used to guarantee WCET tight estimates because these latencies are more for best or average cases.

Scratchpad Memory (SPM): In [22] a Dynamic Scratchpad Memory (SPM) unit has been introduced in order to protect the critical tasks from interference of non-critical tasks. Any task that runs out of the SPM is guaranteed to have a predictable execution time. On the other hand, because the non-critical tasks run out of the cache, there is no degrading in the performance of other non-critical tasks.

Shared bus: Multiple cores access the main memory via a bus. The impact of bus arbitration protocol and communication delay must be accounted for to obtain a tight WCET analysis. One solution is the TDMA bus arbitration which is an excellent mechanism to isolate the tasks and is

predictable and composable. However, most of the existing high-performance COTS-based systems did not apply the TDMA, mainly because it requires additional hardware modifications.

DRAM memory: The unpredictability in DRAMs originate from their internal architecture, which is designed to deliver high volume storage at low cost. The target row must first be opened on a DRAM access before reading, or writing operations can be issued to the word-sized column elements. The response time of memory requests in DRAM is highly variable because a single DRAM chip is designed to have a limited data bus (usually 8 bits) to minimize chip cost [23]. So, this impacts the execution time from a few clock cycles to tens of cycles. Therefore, DRAM memories should be considered a highly unpredictable resource.

Memory Controller: The memory controller connects the processor tile to the off-chip DRAM and is responsible for scheduling memory accesses according to the system requirements. Thus, the response time to a memory request strongly depends on the page policy, the scheduling algorithm, and the power-management policy used in the memory controller.

Direct Memory Access (DMA): In many embedded systems, whenever a task requires to communicate with an off-chip memory, DMA allows this access independently from the processor. This feature is useful for decreasing the CPU overhead, leading to better performance at the end. However, the composition of DMA with other components, for instance, simultaneously accessing a bus, can make the predictability difficult [24].

2.2.2 Requirement for hybrid switchable architecture design

From the perspective of the architectural elements, some characteristics make a system suitable for predictable or high-performance computation. These characteristics could be related to either the processor core or the whole platform. Our analysis focuses on identifying the suitability of different platforms for the realization of implementing a switchable architecture between high-performance and real-time modes. There are some constraints that our

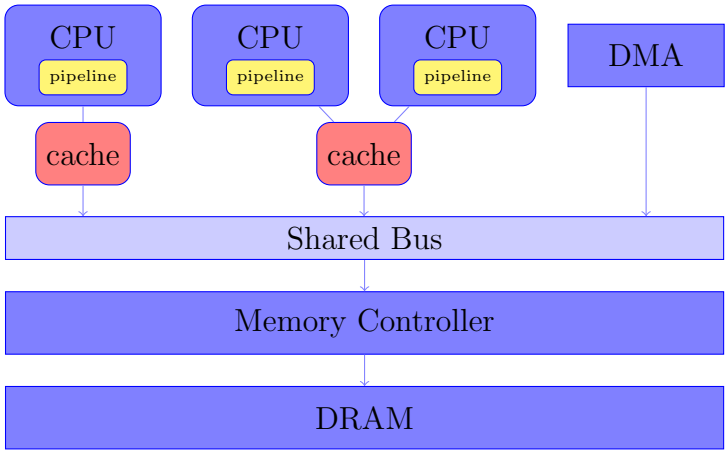

Figure 2.2: Architectural elements which are essential sources of unpredictability.

desired system should meet for building a switchable architecture. These constraints could be identified as specific requirements for the switchable architecture.

Reproducible and predictable execution times: In order to fulfill this requirement, both processors and the memory subsystems directly connected to the core need to be time-predictable. In this regard, static execution time analysis is required to estimate the WCET to guarantee that execution is free of interferences. For instance, a compiler can increase the predictability by annotating metadata to the binary file of critical applications so that these applications execute e.g. from the scratchpad memory, to omit the impact of the Dynamic RAM (DRAM) for hard real-time workloads.

Reproducible and predictable communication times: In a single-core scenario, it is not a big problem, but as soon as we have DMA, it can be problematic whenever some master components want to access the bus simultaneously. Here we are interested in evaluating the predictability of components like busses that are being used to safely upper bound the communication times of the shared resources. A critical decision here is which kind of memory controller each platform uses and how DRAM accesses are managed in the scope of predictability. Some types of memory controllers could significantly enhance predictability.

Operating system capability to support task scheduling for mixed-criticality workloads: Here the task scheduler has to figure out the sequence of programs while keeping specific timing constraints. In this regard, the predictability of a platform helps a lot to provide the scheduling guarantee because, without it, there is no guarantee for all times. Thus, in some cases, it turns out to be operating system support for real-time systems to handle the execution of hard real-time and soft real-time tasks. Time-Division Multiplexing (TDM) is a common solution for mixed workloads, but this method's problem is higher latency.

High-performance features: This can be assessed in terms of architectural elements and computing resources that have been designed for performance and not necessarily tuned out for RT guarantees. For instance, how well the platform's processor core and memory hierarchy are optimized for average-case performance computing.

Execution mode switching for mixed workloads: With this requirement, we want to explain the potentiality of each platform for separating between real-time and non-real-time workloads, which enables the system to handle multiple context switches between high-performance and predictable execution modes at run time. This is not available in many processors and platforms to the best of our knowledge, but we aim to provide a processor that fulfills this requirement.

2.3 Overview of RISC-V ecosystem

Throughout the last decades, large semiconductor companies like INTEL and AMD have dominated the CPU market, which concerns their x86 architectures. Similarly, ARM develops processor designs mainly for mobile devices and microcontrollers. In 2010, RISC-V was born in a lab at the University of California, Berkeley. RISC-V is a new instruction set architecture (ISA) originally designed to support computer architecture research and education. RISC-V evolved a new era of computing on an open and collaborative concept. Recently RISC-V became a standard free and open architecture also for industry implementations [25]. Some fundamental characteristics of RISC-V

are simplicity, modularity, stability, and design for specialization. In order to manage the diversity and support the modularity, there are 47 base instructions from 32-bit to 128-bit address space and some optional standard extensions. However, a single fixed ISA spec can not work for all domains.

Currently, there are different RISC-V ISAs as standard extensions available. It is also possible to produce a binary by combining these RISC-V ISA configurations. In the following, different RISC-V ISAs for the 64-bit processors, as an example, have been introduced:

- `rv64i` for base integer instruction set
- `rv64a` for atomic memory operations
- `rv64f` for single-precision floating-point operations
- `rv64d` for double precision floating-point operations
- `rv64m` for integer multiplication and division
- `rv64c` for compressed instructions

For instance, "rv64imac" describes a 64-bit implementation with integer multiplication, atomic memory instructions, and support for compressed instructions.

The specification is divided into two parts: the unprivileged and privileged specifications. The unprivileged specification, also called the user mode ISA, defines general-purpose computational instructions. The privileged ISA describes capabilities used for an operating system or platform level code [26].

2.3.1 Compilers

Regarding the available software toolchain for RISC-V[1], compiler support for C and C++ is provided through GCC and Clang/LLVM.

[1] A full overview of the RISC-V software ecosystem is available at https://github.com/riscv/riscv-software-list.

The GNU Compiler Collection (GCC) has been the de-facto standard compiler for embedded systems for a long time. The main reason was its support for many embedded processors and microcontrollers and its place as the official compiler for GNU/Linux. The main reason for this success was the open-source model and the support from the GNU/Linux community. Today GCC is one of the most mature compilers for all kinds of computer systems from the embedded to the HPC spectrum. Furthermore, for RISC-V, GCC was the first available compiler and is still the "default" one.

Clang/LLVM [27] is the new "challenger" for GCC. LLVM is an open-source software developed at the University of Illinois/NCSA. The initial version was released in 2003. The name of LLVM was first abbreviated from Low-Level Virtual Machine. Clang is the C, C++, and Objective-C front-end for LLVM. In previous years, back-end support for RISC-V has been added into the Clang/LLVM release [28].

With the availability of two different mature compilers for RISC-V processors, one interesting question is which compiler offers the best performance w.r.t. binary size and execution time. The execution time analysis of the compiled binaries in [29] shows that in 42% of the experiments, GCC and LLVM have nearly the same execution time clock cycles while in 40% GCC execute faster and in 18% LLVM binaries executed faster. Regarding the binary size, in 94% of the experiments, these compilers provide the same binary size. So, in general, the GCC still performs better, and we continue the project with the GCC compiler.

2.3.2 RISC-V cores

There are many core implementations based on RISC-V ISA available in the market. These processors are either optimized for size and code density [30] or optimized for embedded high-performance computing [3, 31]. Developing 32-bit cores essential for Internet of Thing (IoT) and microcontrollers is much more populated than 64-bit in RISC-V. From the design point of view, the first reason could be the complexity of the 64-bit processor design. The second reason is that it is much more expensive to make 64-bit cores available in

Table 2.1: RISC-V cores from Low cost to Linux capable cores

32-bit		**64-bit**
Low cost	DSP	Linux capable
• Zero-riscy – RV32-ICM • Micro-riscy – RV32-CE	• RI5CY – RV32-ICMFX – SIMD – HW loops – Bit manipulation – Fixed point	• Ariane – RV64-ICMAFD – Full privileged specification • Boom – RV64-ICMAFD – Full privileged specification – Out-of-order pipeline

silicon or instantiate it on a Field Programmable Gate Array (FPGA). Moreover, supporting the Linux environment for open source implementations is essential for a 64-bit core. Nevertheless, there are several 64-bit cores, e.g., Boom [32, 33] and Ariane [3] available. We focus on the Ariane processor in the rest of this thesis.

Chapter 3

Related Work

In this chapter, different embedded platforms have been widely investigated. Therefore, this chapter gives an overview of existing platforms that provide solutions for time predictable, high-performance, or both and places them in the context of the work presented in this thesis. In the end, an analysis of the introduced related work platforms will be presented to check the suitability of existing platforms with the concept of switchable architecture design based on the requirements proposed in Section 2.2.2.

3.1 High-Performance platforms

Some embedded high-performance platforms will be reviewed in this section, and their characteristics will be investigated. These platforms are typically available in the embedded system domain if we want to use an embedded high-performance system. Some representations are reviewed here.

3.1.1 Ariane (CVA6)

Ariane [3] is a processor that is optimized as a low area high-performance class processor in the RISC-V ecosystem. On the other hand, Ariane features support address translation via a Memory Management Unit (MMU) to support an operating system [3]. The processor supports the Linux kernel, but there is no real-time task scheduling in the current version. So, its

performance for a comparatively low-area chip is acceptable, but the caches and memory subsystem were not designed for predictability. My approach will introduce predictable execution capabilities to the core. We will also add real-time operating system support for scheduling tasks with different real-time requirements to make it capable of executing tasks with different criticality levels.

3.1.2 Raspberry Pi 4

The Raspberry Pi 4 [34] consists of ARM Cortex A-series processors with the hierarchy of shared caches and the out-of-order pipeline, which is optimized for average-case performance. So, it is not designed to fulfill the predictability and composability requirements for timing critical workload in a switchable architecture. In this platform, we could integrate a system partitioner using a software layer to separate the cache accesses or install a hypervisor on it to fulfill the predictability requirements to a certain degree but still not fully reach the requirements. So it is not that easy to realize predictable support with the hardware components. The Raspberry Pi supports the operating system but is not out of the box for the scheduling of real-time applications. Therefore, we can easily claim that it is an excellent high-performance platform but not a good option for mixed-criticality environments.

3.1.3 NVIDIA Jetson Xavier

Regarding NVIDIA Jetson Xavier [35], the explanation about the architecture and the compatibility with the requirements is quite similar to Raspberry Pi plus additional high-performance feature that it has w.r.t its heterogeneity of the embedded Graphics Processing Unit (GPU) inside the architecture. So controlling the accesses and static analysis for predictability is very difficult in such an architecture. Thus, we will not elaborate more about this high-performance platform because it cannot fulfill predictability requirements and support mixed critical workloads.

3.1.4 Zynq 7000

The Zynq 7000 includes Programmable Logic (PL) and Processing System (PS). The exciting feature inside the PL is the potential to implement arbitrary hardware depending on whether we want to customize hardware for timing predictable computation using the Microblaze processor or realize custom hardware accelerators to boost performance. There are also some designs available that successfully developed a mixed timing criticality system using FPGA in Zynq 7000 [36]. Furthermore, regarding the operating system support, Zynq UltraScale+ supports hypervisor or Linux to make shared caches access times more predictable by providing cache partitioning or cache coloring. On the other hand, Zynq UltraScale+ is an Multiprocessor System on a Chip (MPSoC) that inherits the properties of Zynq 7000 plus the so-called safety island design for real-time applications utilizing a dual-core ARM Cortex-R5 based processing system. Regarding the communication times, the system could be composable and predictable due to the globally time-triggered architecture of the NoC, but this requires some manipulations in components. It is possible to support execution mode switching in the Zynq 7000 but not in a single core rather than using both PL and PS parts or in multi-core scenario implementations.

3.2 Real-time class platfroms

Some platforms designed to satisfy timing predictability requirements will be reviewed in this section, and their characteristics will be investigated. These platforms are typically available in the embedded system domain if we have tight real-time constraints. Some representations are reviewed here.

3.2.1 CompSoC

CompSoC [37] is being categorized as a system that provides solutions for predictability and composability on a platform level by having a global schedule for the whole system, so we analyze it tile by tile and on the platform

level. In order to assess the predictable execution capability, before executing an application on the processor tile, a WCET analysis is done to ensure that the execution time will be within the deadline on the Microblaze processor. The CompSoC platform does not have any cache, but instead, CompSoC utilized local Instruction Memory (IMEM) and Data Memory (DMEM) as a predictable solution with the precise size of instruction and data of programs that are allowed to be executed. Of course, this trade-off affects the platform's performance, but this allows the virtual platform to maintain the average case performance and improve the predictability potential. The Network-on-Chip (NoC) of CompSoC is entirely predictable and responsible for transporting requests from the direct memory access unit (DMA) to the off-chip DRAM memory. Another essential component approach in this architecture are predictable memory controllers with a combination of statically and dynamically scheduled Synchronous DRAM (SDRAM) controllers [38]. CompSoC resets the resource state between scheduling intervals. Thus, the interconnects and memory subsystem of the CompSoC are organized to decrease interferences with other communicational elements by controlling the accesses. This makes it slower due to some buffering and WCET estimations but fulfills the composability and predictability requirements in terms of communication times. Regarding operating system support for different criticality tasks, by using a hypervisor-based software approach, a description is annotated to each application code to make it ready to start the execution on a specific physical tile on the virtual platform. Also, CompSoC provides time-triggered scheduling. This causes CompSoC to be able to execute applications with different timing requirements simultaneously [39]. In this multi-core platform, the potentiality of processor separation from each other is very important to reduce the probability of interferences. Moreover, this platform can exchange the Microblaze processor with the ARM processor in the tiles to improve the high-performance features of the platform, which is an outstanding feature in multi-core scenarios. In a switchable platform, it could be configured in case of predictable and high-performance computation, which processor gets the priority of accessing the NoC. This architecture puts the processors in independent tiles with independent local memories to avoid the

interferences of the shared memory. If we want to consider it as a platform capable of switching between different modes at run-time, it could be easily available thanks to the software features of the virtual platform. Based on the explained architectural characteristics of the CompSoC virtual platform, it could be understood that the application execution is completely partitioned, so executed applications cannot affect others. Furthermore, the utilization of shared resources for different applications with different criticality levels has been done thanks to the composability of the platform efficiently. The stable software scheduling, predictable hardware design components, and average performance of the processor make this platform a good choice as a run-time switchable platform. The only drawback of CompSoC is that it does not support high-performance execution.

3.2.2 PATMOS

Generally, T-CREST [40] is optimized to be a time-predictable multi-core platform. From the processor point of view, PATMOS is used as the fully predictable core in the T-CREST platform that provides tight integration of the compiler and WCET analysis to enable higher processor utilization [41]. On the other hand, to enable the overall predictability of the whole platform, T-CREST benefits from system partitioning to assign each real-time application a unique execution environment (hardware resources) and avoid other applications using this exclusive hardware. So, the assumption is that there is enough hardware to isolate for executing a program, which is costly but practical. It statically serves the WCET execution time with average performance output and does not waste much performance. PATMOS contains size configurable method, data, and stack caches and two SPMs, but for T-CREST as a multi-core platform, the SPM usage is under MOSSCA operating system control [42]. Regarding operating system support, the TDMA based memory access arbitration is used for fair and predictable accessing the controller. Moreover, a combination of the time-predictable memory tree and the time-predictable memory controller is needed on a multi-core system. T-CREST interconnects consist of predictable NoC, which is used to com-

municate between processors to enable time-predictable usage of a shared resource. Thus, PATMOS features the first four requirements because the processor has enough potential as an average case performance and a fully predictable processor. Hardware-based isolation is considered as a secure solution for predictability and also, in this case, provides good performance. The only drawback is that it is not the case in many other architectures due to hardware resources limitations. In addition, the PATMOS pipeline is configurable to be either a standard dual-issue for good performance without the unpredictability of dynamic instruction issuing or a single issue to occupy less area. Since T-CREST has been designed as a fully predictable platform, it does not support switching for mixed workloads. However, PATMOS is applicable to be a switchable platform due to its predictability and performance features, among many other available architectures. Nevertheless, it should be considered that this implementation is costly and also requires some modifications to separate between mixed HP and RT workloads.

3.2.3 FlexPRET

FlexPRET [43] is a 32-bit, 5-stage, a fine-grained multithreaded processor with software-controlled, flexible thread scheduling. It employs a classical RISC 5-stage pipeline with instruction fetch (F), decode (D), execute (E), memory (M), and writeback (W). FlexPRET naturally offers running soft-real-time and hard-real-time workloads in parallel. It supports an arbitrary interleaving of threads, controlled by a thread scheduler in a mixed-criticality system to reduce hardware costs. Applications executed in a hard real-time thread execute in predictable and reproducible time. Furthermore, for these threads, the execution times are composable because FlexPRET utilized hardware-based isolation for these HRTTs, so this is an excellent example of self composability inside the pipeline. Another aspect of FlexPRET is using instruction and data SPMs to avoid the unpredictability issues of the caches. Since FlexPRET is only a processor core and not an entire platform, no statement can be made concerning the properties of the communication times. One solution for operating system support for task scheduling could

have been software-based isolation provided by an RTOS, but the operating system support is not available so far. Compared with a predictable PATMOS processor in the T-CREST platform, FlexPRET supports executing tasks with lower predictability using soft real-time threads and avoids underutilization of resources. The overall performance of this processor is limited by the in-order pipeline design. In addition, the execution mode switching is available for hard and soft real-time applications and not for high-performance and real-time applications.

3.2.4 SPEAR

SPEAR consists of a 16-bit processor core with three pipeline stages to achieve reasonable performance; however, in SPEAR2 [44] the performance feature is improved by adding memory stage to the pipeline, but it is still far from performance optimizations. The goal of SPEAR architecture is deterministic timing behavior within the processor core. Moreover, SPEAR offers a constant one-clock cycle execution time for each instruction by avoiding all pipeline data and control hazards to keep the processor predictable. It fulfills the predictability of execution time requirement. Since it is a single processor that is optimized only for hard real-time workloads, the composability features and predictability for communication purposes are not available. Regarding the memory architecture, there are 4kB data and four kB instructions on-chip memories and external caches. SPEAR neither supports an operating system nor a switching environment for mixed workloads. Although SPEAR has a highly efficient architecture suitable for hard real-time applications, it needs more optimization in the memory subsystem in order to increase the capability of utilizing it for some average-performance workloads.

3.3 Analysis of suitability of existing platforms for switchable architecture design

In Section 2.2.2 the requirements for switchable processor design were demonstrated. As a general assessment, we have provided Table 3.1 to check for the processor whether it fulfills each specific requirement or not. In this regard, we introduced a scoring scheme to clarify to what extent they fulfill each requirement. Therefore, ++ is assigned to a processor that completely fulfills that requirement without needing any other modification in the microarchitectures or software level. + assigned to a processor capable of fulfilling that requirement but with minor modification and - is for the platform that does not perform well on that requirement. Finally, for the platforms that corresponding requirement is not defined or is not possible to progress in the sense of that requirement *na* assigned as not applicable.
In order to illustrate a summary of the results collected in this table, we can see that in the Zynq 7000, it is possible to support execution mode switching but not in a single core rather than using both PL and PS parts. CompSoC and PATMOS include average-case performance processors that fulfill most requirements because of the capability of providing high timing predictability levels. However, the exciting feature of the CompSoC platform is its potential to execute different workloads. Therefore, the CompSoC fulfills all requirements. Altogether, like Zynq 7000, this could be considered a multi-core solution, but the advantage is that CompSoC can fulfill all requirements. In addition, FlexPRET as a RISC-V-based processor naturally offers the execution mode switching for hard and soft real-time applications but not for high-performance and real-time applications. Ariane offers better performance than FlexPRET, but the caches and memory subsystem were not designed for predictability.

Table 3.1: Comparison of different processors and platforms based on the requirements of switchable architecture design

Platform \ **requirements**	**Pred. execution times**	**General OS support**	**HP feature**	**Execution mode switching**
CVA6 (Ariane)	-	+	+	na
Raspberry Pi 4	-	+	+	-
NVIDIA Jetson	-	+	+	-
XLINX Zynq 7000	+	+	+	na
CoMPSoC	+	+	+	+
PATMOS	+	+	+	na
FlexPRET	+	-	+	+
SPEAR	+	-	-	na
proposed	+	+	+	+

3.4 Gap analysis

Based on the literature review, this section maps existing platforms to the scope of this work to identify the research gap. To this end, related work is identified from R1 to R8. The following list shows the related work that has been mapped to the scope:

R1 Ariane an embedded application class RISC-V based platform [3] (see description in Section 3.1.1)

R2 Raspberry Pi 4 an average-case performance ARM-based platform [34] (see description in Section 3.1.2)

R3 NVIDIA Jetson Xavier a high-performance platform [35] (see description in Section 3.1.3)

R4 Zynq 7000 [36] (see description in Section 3.1.4)

R5 CompSoC virtual platform [37] (see description in Section 3.2.1)

R6 PATMOS fully predictable processor [41] (see description in Section 3.2.2)

R7 FlexPRET predictable processor [39] (see description in Section 3.2.3)

R8 SPEAR predictable processor [44] (see description in Section 3.2.4)

Figure 3.1 demonstrates the diagram resulting from mapping related work to the scope, which comprises two research areas: embedded high-performance platforms, where speed is more important than meeting the deadline for real-time tasks; real-time platforms, where executing a real-time task within the deadline is essential. However, none of them describes how they can be implemented so that it will be possible to run different software programs on a single-core platform where it is possible to meet the deadline for the real-time workloads as well as supporting non-real-time workloads.

Although, as shown in the figure, R4 and R5 could be named as platforms that can support mode switching, the solution they may provide for our research will be in the area of multi-core implementation where the implementation is costly. Our goal is to fill this gap for single-core implementation in the RISC-V ecosystem.

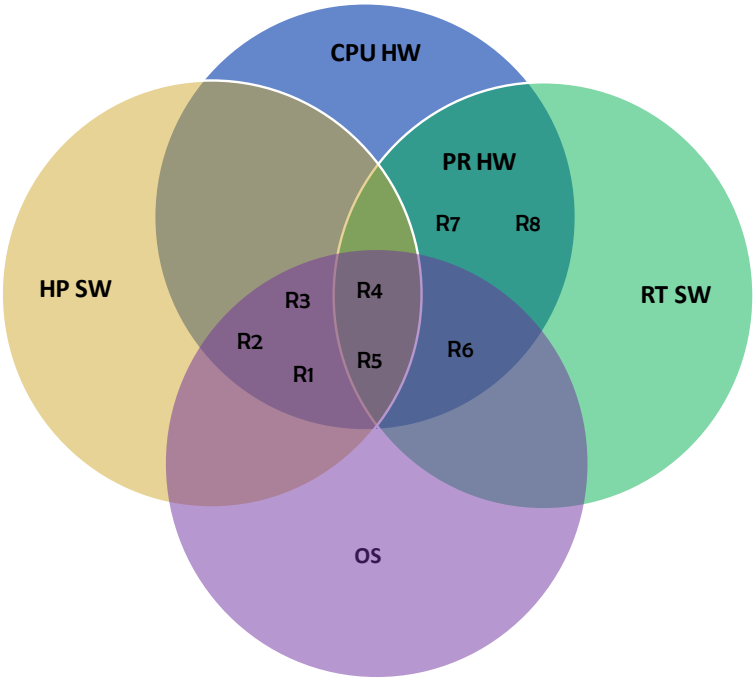

Figure 3.1: Related work analysis. Mapping related work to the scope.

Chapter 4

Thesis Contributions

This thesis covered the fundamentals of embedded system development for both real-time and high-performance systems and explained the challenges in the specification and development of the systems that can fulfill requirements for these different platforms. In addition, state-of-the-art platforms, excluding their microarchitecture, were introduced to be familiar with the benefits and drawbacks of these platforms for different criticality execution environments. From the gap analysis (see Section 3.4), we concluded that a couple of available platforms could provide solutions for both RT and HP applications; however, not for single-core implementation. In Chapter 4, contributions and assumptions of the thesis will be discussed.

4.1 Constraints

Before introducing the contributions, the following assumptions and constraints need to be determined:

A&C1 The platform we are targeting to propose our mode switchable design should be single-core, and we are not looking at multi-core solutions.

A&C2 It should be applicable in the RISC-V ecosystem. Therefore we are not comparing this approach with any other architecture.

A&C3 Since we are supporting both embedded high-performance and timing predictable workloads, the platform should include a high-performance class processor with the capability of adopting for real-time application by performing architectural and software modifications or manipulations.

A&C4 We are looking at a solution with light operating system support regarding the software complexity. Therefore, we are not comparing our approach with Linux-based or hypervisor-based platforms.

A&C5 We are not focusing on dynamic memory allocation for real-time tasks and scheduling these tasks statically before actual execution at runtime.

4.2 Contributions

This research work aims to propose a RISC-V-based processor with two different configurations (real-time and non-real-time) capable of switching between these two modes. The reason for designing such a system is to have the advantages of predictable and high-performance in one microarchitecture, which to the best of our knowledge, is not available in any other architecture in the RISC-V ecosystem. Therefore, for RISC-V, there is a dedicated need to satisfy more real-time requirements on an embedded application class processor. Therefore, this is not well addressed in the ecosystem. Altogether, the contributions of this thesis are:

Contribution C1: Timing analysis of different memory architecture configurations with respect to performance and predictability on Ariane platform.

Before starting any implementation, it is needed to perform a timing analysis to assess the impact of the memory subsystem of the Ariane processor to understand to what extent the caches and the DRAM is predictable. So, by completing this evaluation, we were able to think about a way to improve the platform's time-predictability for our real-time mode.

Contribution C2: Implementation of Static Random Access Memory (SRAM) into the memory subsystem of Ariane platform as a predictable building block for real-time applications. Ariane currently lacks this feature.

The hardware side of this project will consist of a modified Ariane processor core. The hardware is made ready to switch between high-performance and real-time mode by exploiting more predictable components compared to the Ariane platform. The idea is to benefit from the default platform configuration, including caches and DRAM for the high-performance mode, and design and implement a predictable platform for the real-time mode. In connection with improvements to the predictability of the hardware, we injected a SRAM into the processor implementation such that we can guarantee a predictable execution time for real-time tasks.

Contribution C3: Provide operating system support for switching tasks with different criticality levels using freeRTOS on the Ariane processor.

The software (operating system) will be able to trigger mode switching of the hardware. Tasks will be classified as Real-Time (RT) and High-Performance (HP) and executed in the corresponding hardware mode. The operating system will be able to guarantee the task scheduling for predictable execution of the real-time tasks with reasonable overhead, using hardware features of the processor. The idea is to configure a model that gives priority to real-time tasks in specific time frames which are repeated periodically while allowing the non-real-time tasks to be scheduled in unused slacks in the remaining time of each frame [45]. By executing different tasks with different levels of criticality, we improve the composability of the execution time on this platform.

Contribution C4: Propose an automatic measurement framework to capture the applications timing behavior for different compiler or execution configurations.

By mapping the presented contributions to the scope of the thesis, as shown in Figure 4.1, the main focus is on improving the overlapped area.

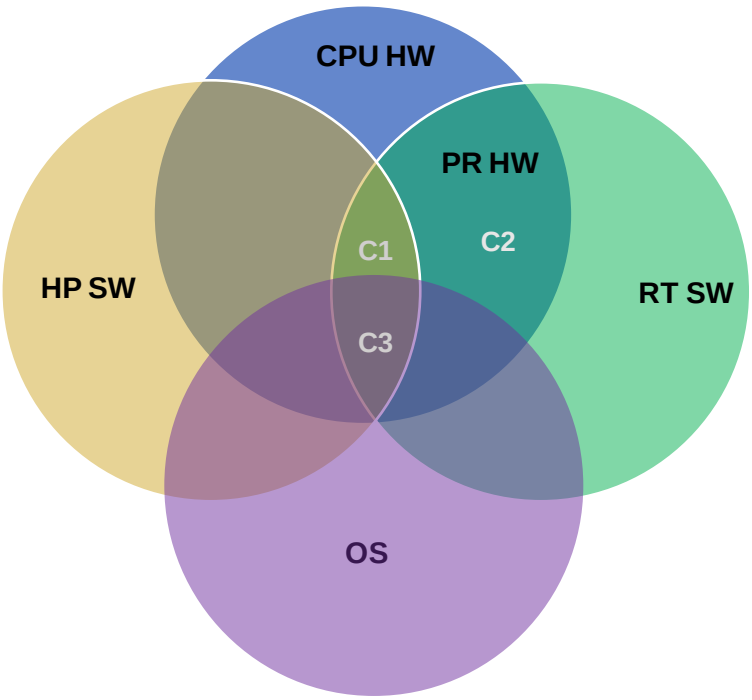

Figure 4.1: Contributions analysis. Mapping contributions to the scope.

Chapter 5

Concept

This section presents how it is possible to conceptually realize the idea of switchable architecture design in a single-core platform by providing a formal formulation and execution examples in theory. Hence, The first section presents an overview of the concept. Afterward, Section 5.2 introduces the application and execution model for the proposed concept. Then Section 5.3 describes how the memory architecture could look like if the concept applies to real hardware. Finally, the memory hierarchy with the data path for each execution mode will be depicted.

5.1 An overview of mode switchable concept

This project aims to design a system that by construction can fulfill the timing requirement of the real-time tasks executed by a time predictable architecture configuration and the non-real-time tasks executed by a performance-optimized architecture configuration. To realize this objective, we require a mode switchable architecture including RT and non-RT architecture configurations. The mode switchable architecture refers to the capability of different applications with different timing requirements on a single-core platform. Consequently, the platform itself, excluding any ISA, should have the capability to execute real-time and non-real-time applications without losing

either performance for non-real-time tasks or timing predictability for real-time tasks. So, we categorize all executing applications into two modes before run-time: applications in real-time mode and applications in non-real-time mode. Each of these two modes has specific characteristics. In this section, these two modes will be introduced, and the properties of each mode will be explained.

5.1.1 Characteristics of Real-Time mode

In a mode switchable platform, a task is in real-time mode if it needs guaranteeing execution within its deadline and other tasks cannot interrupt the execution of the RT task. We are not considering hard-real-time and soft-real-time tasks in different categories to simplify modeling. So, we consider all RT tasks in the same workload in real-time mode.

5.1.2 Characteristics of Non-Real-Time mode

Some applications do not have real-time requirements. For non-real-time tasks, we apply the best effort policy, which means the tasks executing in this architecture configuration should do the computation as fast as possible. In a mode switchable platform, a task is in non-real-time mode; if this task is executing in the background continuously or within the slacks of executing the real-time tasks. However, it should be guaranteed that real-time tasks should be executed without any timing constraints.

5.2 Switchable Architecture Model Definition

The mode switchable platform is a processor that executes a set of tasks. According to the static scheduling policy, one or some of the tasks could be periodic, which is defined prior to the execution. Similarly, some tasks could be executed in the background as non-real-time programs. The following

subsections define the main application and execution modeling elements through formal notation.

5.2.1 Application model

Definition 2. The application model A is defined as a tuple (T,S,M) consisting of a set of tasks T, switching modes S, and memory mapping M. A task $t_i \in T$={B, P} where B is a background task which is always non-real-time and consisting of a chain of functions f_i ; and P is a periodic task which is always real-time. Switching mode $s_i \in S$={R, N} where R is a real-time task and N is a non-real-time task. Memory mapping M differentiates the tasks based on their switching modes to be mapped on a predictable or high-performance memory region.

Definition 3. There are some other constraints for a periodic task P. Accordingly, P could be represented by a tuple (p_i,d_i,u_i,c_i). p_i is a period of the task, d_i is the task's deadline, u_i is upper bound or WCET, and finally, c_i is the capacity of the task that should be smaller or equal to the size of memory.

5.2.2 Execution model

This section explains the execution semantics in a mode switchable environment. Tasks should be mapped statically before execution in the real-time mode, and in non-real-time mode, tasks will be dynamically mapped during execution. The run-time model is inspired by [45], which separates the estimated resource temporally into frames. The frames will be executed according to the type of applications (Real-time and Non-real-time).

Definition 4. The execution model E consists of a set of frames $\{f_j\}$. The frame size F is set through the maximum execution duration of the periodic tasks. This could be determined by offline analysis before actual execution on the platform. Thus, the scheduling policy for each P should be static before execution, and the scheduling policy for each B could be dynamic during run-time. Furthermore, in order to achieve more performance, in the

case of execution of the periodic task P does not take until the end of the frame, the background task B in the slack phase could be scheduled during run-time so that we do not lose the time until the end of the frame. Based on the constraints of mode switchable architecture design of this thesis (Section 4.1) and the previous sections in this chapter, two examples are introduced in the following as probable scenarios. Figure 5.1 and Figure 5.2 depict these two example executions.

5.2.3 Example of Application Execution

To explain the example shown in Figure 5.1 and based on the application model described in Section 5.2.1, three tasks with different timing requirements are running that in this case, two of them are periodic tasks ($T1$ and $T2$), and one of them is a background task B. $T1$ and $T2$ are running in the switching mode R while B is running in the switching mode N. Accordingly, $T1$ and $T2$ conceptually should be mapped to the predictable memory region, and B should be mapped to the high-performance memory region. When we look at the first two tasks, on the one hand, we do not have preemption between two periodic tasks (real-time) because $T1$ and $T2$ were statically scheduled with known WCET before executing on a real platform. On the other hand, based on this static scheduling, we should know and guarantee that $T1$ and $T2$ execute before the frame ends. The frame size is calculated based on the WCET of the slowest periodic task, which is, in this example $T1$. We can observe that execution of the $T2$ finished before the frame ends. Therefore, conceptually it is possible to use the remained time frame for executing the background task. So they should be executed without any timing issue. Nevertheless, when there is no real-time task running, and background task B starts executing, it is always possible to interrupt this task (slim lanes show when an interrupt occurs and the scheduler decides to execute the real-time task). Therefore, underneath, we can observe that $T1$ and $T2$ are executing by RT mode, which could guarantee the predictable execution on the hardware, while B executes by the HP mode, which is the default architecture configuration without a predictable execution guarantee.

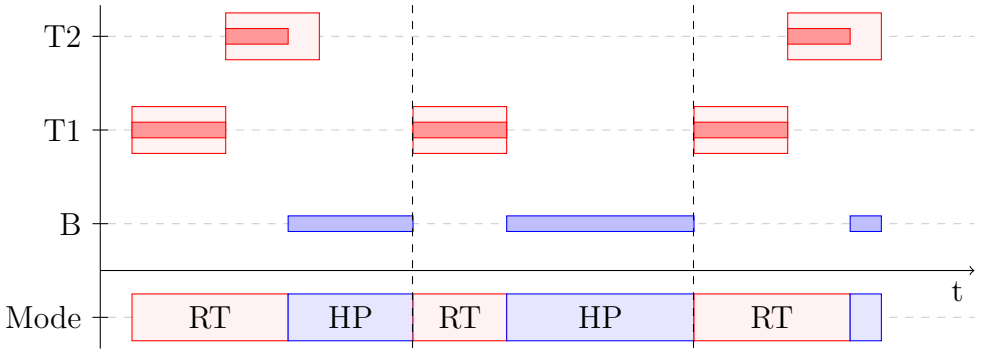

Figure 5.1: Example execution trace of two execution modes representing two Real-time and one Non-real-time task execution behavior

Figure 5.2, shows another scenario in which three tasks are running, and in this case, two of them are background tasks ($B1$ and $B2$), and one of them is a periodic task T. T runs in the switching mode R while $B1$ and $B2$ run in the switching mode N. Accordingly, T should be mapped to the predictable memory region, while $B1$ and $B2$ should be mapped to the high-performance memory region. So the first two tasks should be executed in the slacks that no real-time task is executing. We can notice that they are running in the first three execution frames. There is no preemption between these two background tasks and just a frame change based on the scheduling policy. However, for the periodic task based on this static scheduling, we know that it should be executed without any timing issue. Hence, the same as the previous figure, slim lanes show when a background task should be interrupted, and the real-time task execution should be triggered.

5.3 Architectural Overview of Switchable Design

The concept of this thesis explains different modes and the integration of these modes. Thus different steps have to be done from the preparation to execution based on whether a task is real-time or non-real-time. This section explains the concept from an architectural perspective.

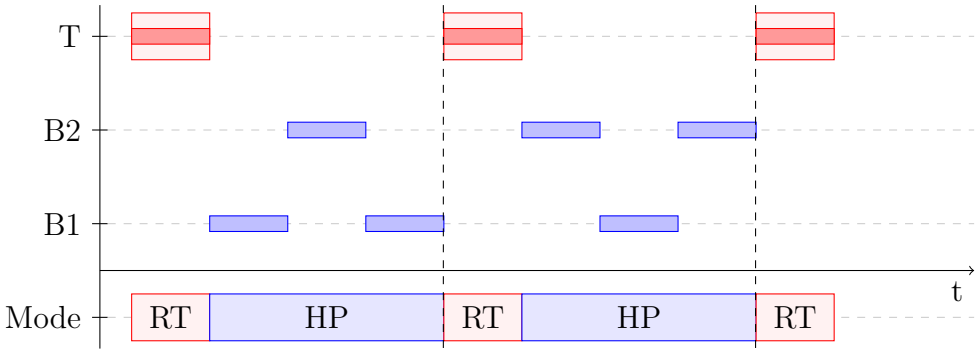

Figure 5.2: Example execution trace of two execution modes representing one Real-time and two Non-real-time task execution behavior

5.3.1 Memory requirements for preparation setup

A predictable memory should support the real-time section. In this thesis, SRAM is used as a memory subsection to guarantee a predictable execution time of the real-time tasks. The response time of memory requests in DRAM is highly variable, which has an impact on the execution time in the range of a few clock cycles to tens of cycles. In addition, refresh cycles make DRAM unpredictable. SRAM is known as an expensive, limited resource in computer architecture. Therefore, we need to do a pre-analysis step on our real-time application code to obtain an upper bound for the memory requirements. We can then combine these upper bounds for all real-time tasks to get the acsram amount that our platform must support. An accurate pre-analysis is not the focus of this work, but we will briefly introduce our approach in the following.

To analyze the memory requirements, we will first need to enumerate the capacity requirements for each benchmark program to be executed from within the SRAM. The following list shows the sources of memory consumption for each benchmark program: program code, static data, dynamic data, and execution stack.

The size of the code and static data can be obtained precisely from the image after compilation. Generally, the estimation of dynamic data is complicated and usually requires either a measurement approach with extensive testing or some static analysis, potentially combined with a coding style lim-

iting dynamic allocation to cases the analyzer can understand [46]. However, we have excluded dynamic allocation in real-time code as a prerequisite in our case. The execution stack is also naturally dynamic, and estimating the maximum stack depth is challenging in an embedded system due to size limitations. The most critical thing would be large stack allocation and deep recursion. We know that all programs will match to the stack size based on execution time measurements, so we did not focus intensely on this part.

As part of pre-analysis, memory mapping could be formalized through a size-counting function because the physical memory sizes (especially the SRAM size) are limited. We have to prove if all real-time tasks fit into the SRAM. This could be formalized as $\sum_{i \in I} S_i \leqslant C$ where C is the capacity of the SRAM, I is the set of all real time tasks and S_i is the size of task $i \in I$.

5.3.2 Architectural overview

As shown in the Figure 5.3 we have two modes (real-time & non-real-time). The non-real-time mode here means that a task is being executed in this mode can be preempted. In the same way, we conceptually do not want a task belonging to real-time mode to be preempted. Then we have the configuration as part of the kernel, which links to memory mapping (static memory allocation based on the task criticality), e.g., T1 should be mapped in a specific memory region in SRAM. It is automatically configured by freeRTOS or a bare-metal implementation so that it will be executed from within the specific memory region that T1 belongs to, which is for the T1 predictable memory section and, e.g., for the T4 high-performance memory section.

5.3.3 Memory hierarchy for real-time execution

To explain the concept precisely, we can zoom in to the memory parts and see how the hardware is modified to execute each task from each part of the memory subsystem that it belongs to, based on the criticality level (Real-time or non-real-time). Therefore, as depicted in Figure 5.4 for the predictable memory section, we bypass the cache. Then attach the SRAM for executing real-time applications employing the SRAM instead of DRAM or cache.

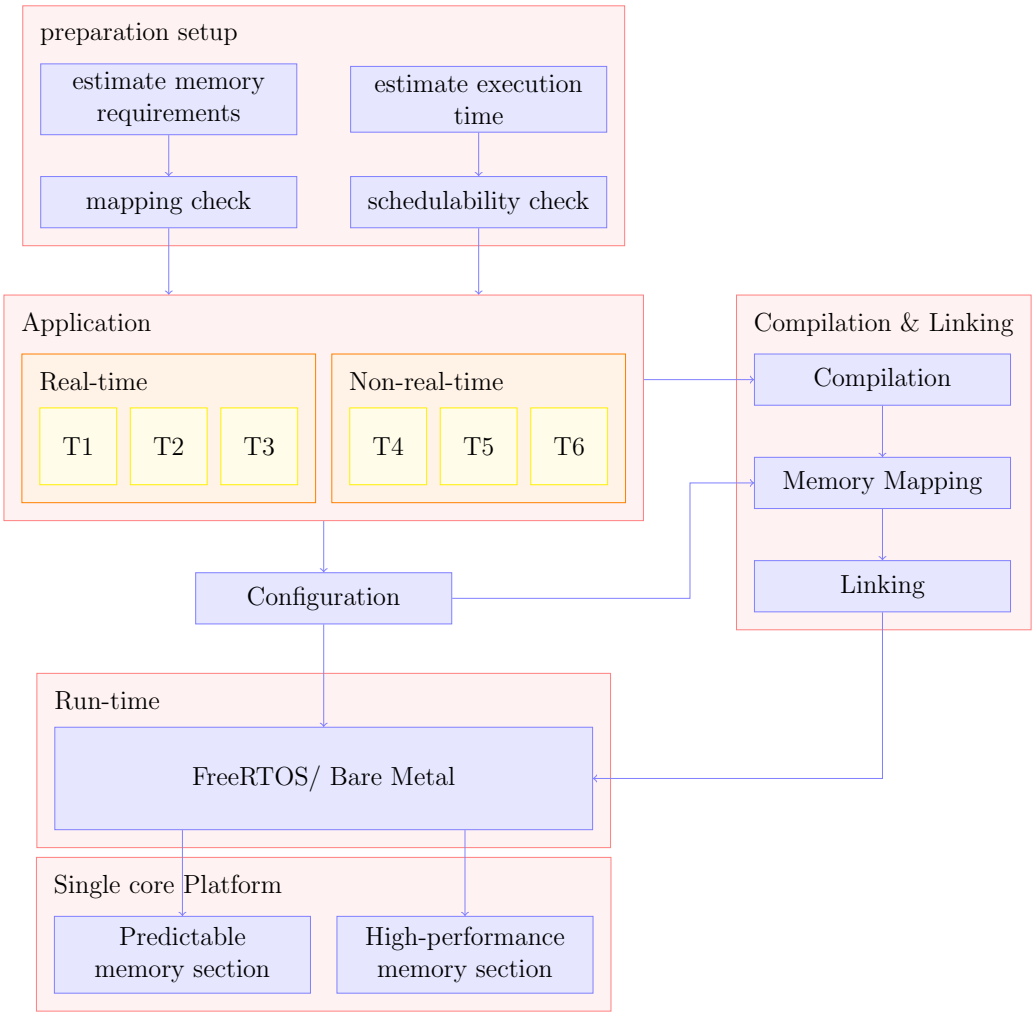

Figure 5.3: Overview of the concept implementation layers on a target platform.

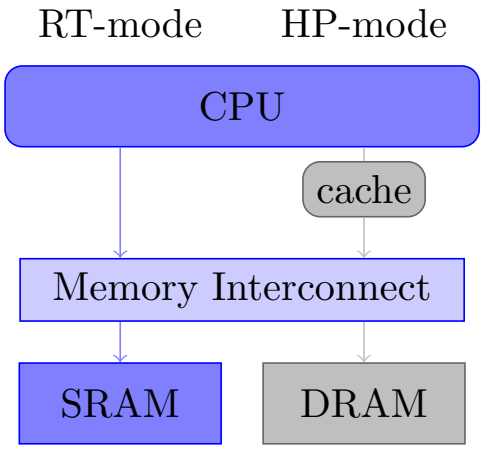

Figure 5.4: Data path of the predictable memory section

5.3.4 Memory hierarchy for non-real-time execution

We had the processor on top, the cache, and the DRAM as available hardware for the high-performance memory section (Figure 5.5). In the evaluation process, we use these different memory configurations to measure the execution time based on the tasks executed from which part of the memory subsystem.

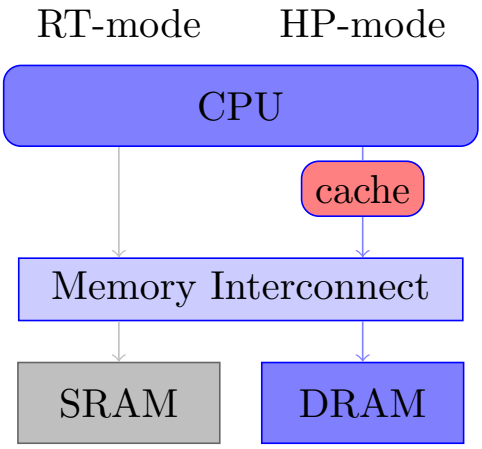

Figure 5.5: Data path of the high-performance memory section

Chapter 6

Design Flow and Implementation

This work aims to propose a processor in the RISC-V ecosystem with two different configurations (HP and RT) with the capability of switching between these two modes. The hardware side of this project consists of a modified Ariane platform by exploiting more predictable components comparing to what currently exists in the Ariane platform. The approach is to benefit from the default platform configuration for the HP mode. Additionally, design and implement a predictable platform for the RT mode in a mode switchable scenario.

Chapter 6 presents hardware, toolchain, and software layers and how to apply the concept to an implemetation from both hardware and software perspectives. The developed measurement framework will be introduced for evaluating execution time behavior for different architecture configurations. The realization of the project by using different layers in order to fulfill the concept requirements here will be illustrated.

6.1 Implementation: Hardware layer

As a proof of concept, our approach has been implemented on the Ariane core. In order to realize the Ariane core, we used an FPGA emulation. By using an

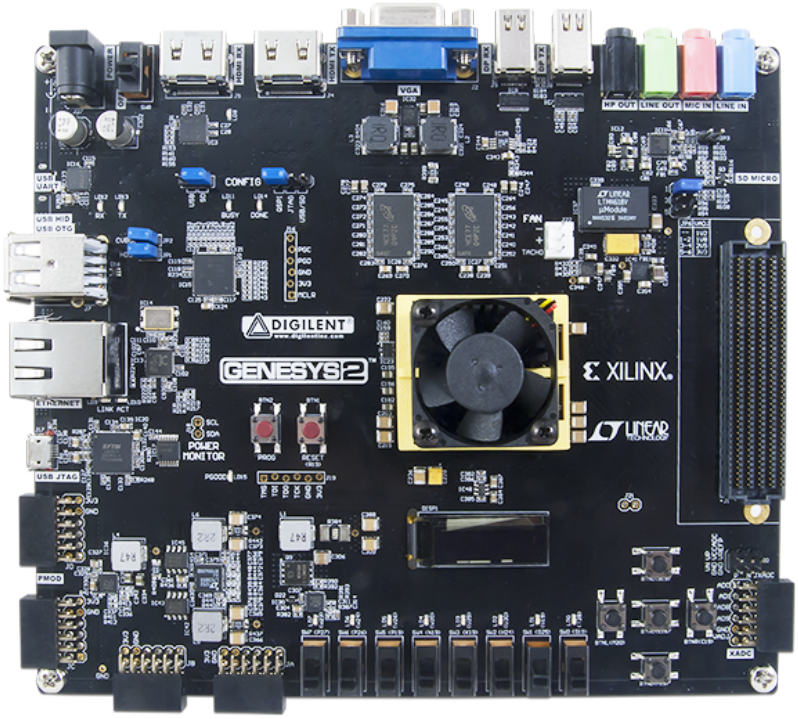

Figure 6.1: The Genesys2 FPGA board (from [47]).

FPGA, we can adapt the implementation to our specific needs. It could even allow us to implement a modified version of the Ariane core with minimal effort. For our implementation, the *Genesys 2* FPGA board by Digilent was used (see Figure 6.1). The *Genesys 2* Board features a *Xilinx Kintex 7* FPGA as well as a variety of peripheral connectors [47]. The bitstream was generated directly from the Ariane source code using the *Xilinx Vivado 2018.3* synthesis suite [48] to program the FPGA. The resulting bitstream is used to program the *Genesys 2* board using *Vivado*.

6.1.1 Target implementation

Ariane is an open-source RISC-V-based processor developed by the PULP group in ETH Zurich. It is a low area application class processor capable of being implemented on the FPGA. Ariane is a 64 bit, single-issue, in-order pipeline processor. It has support for hardware multiply/divide, atomic

memory operations, and a Floating Point Unit (FPU). Moreover, it supports the compressed instruction set extension and the full privileged instruction set extension.

The processor core includes L1 instruction and data caches. These caches are tightly integrated into the core and not considered separate components from a platform perspective. Ariane also provides several performance counters, including Control and Status Registers (CSRs), for different purposes. CSRs control the execution mode of the core. These registers are restricted to the same or a higher privilege level. In addition to the counters for elapsed clock cycles and retired instructions suggested by the RISC-V specification, Ariane supplies cache miss counters for the instruction and data cache. Also, there is a mechanism to enable or disable these caches in software using CSRs [3, 4].

Figure 6.2 shows an overview of the Ariane platform and available peripherals. It depicts how peripherals are connected to the core via an AXI-4 crossbar. There are two different memory interfaces available; the SRAM interface implemented as a simple Advanced eXtensible Interface (AXI) to on device block RAM, and the Double Data Rate (DDR) memory interface, which needs to cross the clock domain boundary between the core clock and the DDR memory clock and connect to the off-chip DDR memory banks. In addition, there is the core level interrupt controler (CLINT), which notably supplies timer interrupts necessary for operating system support that is controlled via memory-mapped registers that are exposed on the AXI bus. Lastly, there is the debug controller that connects to the JTAG port. These peripherals are used for the implementation and evaluation.

All memory accesses by the core pass through the cache interface. This interface consists of two seperate caches (instruction cache and data cache) that share a single connection port to the main bus. The caches are tightly integrated, which is why they are considered part of the core and not seperate components. Also noteworthy are the performance counters used to perform the timing measurements in our evaluation. These counters are also part of the core and concepturally seperate from the memory mapped timer registers.

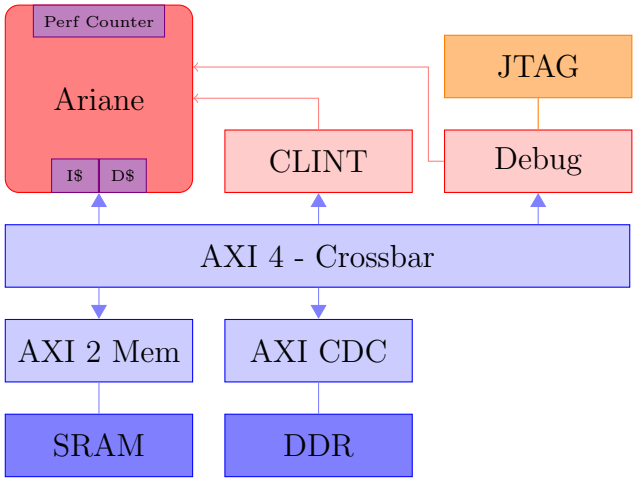

Figure 6.2: Overview of the evaluation platform

6.1.2 Memory system configurations

The results of our experiment show the total variance in execution times caused by non-determinism in three different configurations. We can use the Ariane platform's cache-control mechanism to implement these environments. The mechanism consists of 2 performance counters named CSRs. The CSRs enable or disable the caches, one for the instruction cache and the other for the data cache. Additionally, two performance counters are available to count cache misses when the respective caches are enabled. We divided the possible sources of non-determinism into three categories:

In the first configuration, all memory accesses are served in constant time. This will generate baseline results where only non-determinism introduced by microarchitectural features unrelated to the memory interface will be visible. This environment executes each benchmark once to load all instructions in the instruction cache and all accessed memory in the data cache. Then several runs are executed without flushing the cache in between. If the cache is large enough to hold all required memory (considered separately for instructions and data), the first run will load the cache, and the subsequent runs will then run entirely inside the cache.

In the second configuration, memory accesses are served directly by the DRAM. The variance in the generated results will be caused either by the

internal features or by the memory backend. This environment could be implemented by turning off the cache. To verify that this works as intended, we can use two indicators. First, the cache miss counters do not count when the caches are disabled. Therefore, no misses should be counted in this case. Second, execution times should be significantly larger than with caches enabled because all memory access is served from the main memory, which is naturally much slower than the cache.

In the third configuration, we implemented a Static RAM (SRAM) to the processor implementation to guarantee a predictable execution time for critical tasks and reduce the variations in the memory backend. This environment could be implemented using Xilinx FPGA components and generating the SRAM as an IP. The next step is turning off the cache and forcing the program to be executed from the SRAM. Verification of this implementation is the same as the previous configuration. However, we expect constant execution time with no variation and lower latency than DRAM without cache.

6.2 Implementation: Toolchain Layer and Linking process

The allocated tasks based on the criticality level are mapped to appropriate memory areas during the linking process. Accordingly, the operating system and the real-time tasks are mapped into the memory area served by the SRAM statically to maintain timing predictability execution for the real-time task (without execution time variance). On the other hand, since the FreeRTOS is not large, it is worth keeping it in the predictable memory section to avoid unpredictable artifacts from the operating system side. Similarly, high-performance tasks are mapped into memory areas served by DRAM. For executing the HP code, dynamic allocations are always mapped to DRAM. Since the caches are enabled for these areas, they can execute with maximal performance.

The separation into different memory areas affects both code and static data sections. On the developer side, HP and RT codes are separated into

different compilation units. Afterward, the linker maps these units as necessary. Consequently, it is the programmer's responsibility not to use HP data from RT code, as this would break the real-time guarantees for the RT section. Dynamic memory allocation is usually by design incompatible with real-time requirements. Therefore, we support dynamic allocation only for HP tasks and static allocation for RT tasks.

6.3 Implementation: Software layer

One aspect of the software layer is preparing the system for the execution time measurements and calculating the WCET of benchmarks by measurement. The other aspect, which is an essential part, is the operating system support. The following subsections will explain the mentioned software layer aspects in more detail.

6.3.1 Preparation setup

As mentioned in Chapter 5, for executing a real-time task in a mode switchable architecture, we need static scheduling to guarantee execution within its deadline by interrupting any non-real-time program. For calculating the upper bound and the frame size for each benchmark, we must assess the parameters mentioned in Section 5.3.1. Therefore, we need the size of each benchmark to see whether it could fit within SRAM size and an estimation on the WCET for each benchmark. Table 6.1 indicates the size of applications measured in bytes. The largest size is for the quicksort benchmark, indicating 66350 bytes required memory. The size of available SRAM on the FPGA is 1MB, providing a comfortable margin.

One of the most common methods to determine the WCET is measurement-based WCET Analysis. In this method, the program code is executed on the hardware for 100 iterations. Table 6.2 shows the worst-case execution time we observed during the execution process for each of the executed TACLE benchmark programs on the Ariane platform. The WCET is measured by executing benchmarks using a measurement framework

Table 6.1: Size of static memory for different TACLE benchmarks

benchmark	static memory size in bytes
binarysearch	1658
bitcount	2652
bitonic	2762
bsort	1092
complex_updates	3794
cosf	5282
countnegative	2422
deg2rad	458
fac	508
filterbank	1436
fir2dim	1622
iir	994
insertsort	1160
isqrt	5798
jfdctint	1518
lms	2932
ludcmp	24314
matrix1	2052
md5	8364
prime	796
rad2deg	466
recursion	490
st	10090

iteratively, and we know how long it takes each program to execute in the worst case based on the measurement. Therefore, in this project, we chose the frame sizes appropriately based on the measurements because there is a specific FPGA as a hardware to host the implementation. Therefore, static WCET analysis is not in the scope of this thesis.

6.3.2 Operating system support

The Ariane processor supports executing general-purpose operating systems, notably the Linux operating system. In principle, adding the real-time capa-

Table 6.2: WCET estimate for different TACLE benchmarks

benchmark	WCET estimate in cycles
binarysearch	1022
bitcount	19089
bitonic	16582
bsort	60269
complex_updates	1824
cosf	17567
countnegative	17837
deg2rad	6883
fac	250
filterbank	2928201
fir2dim	5022
iir	1645
insertsort	1364
isqrt	526734
jfdctint	3614
lms	121006
ludcmp	2394
matrix1	7569
md5	9830954
pm	5730584
prime	416
rad2deg	6866
recursion	3328
st	125903

bility does not change this, and it is indeed possible to benefit from a Linux as a strong OS for task scheduling. However, to execute a real-time code with timing predictability requirements, some features in an operating system are required, which are not usually present in a general-purpose operating system. For this reason, FreeRTOS, an open-source real-time operating system, was chosen for this project.

FreeRTOS can be constructed with approximately twenty different compilers and run on more than thirty different processor architectures. In addition, FreeRTOS can be considered a library that provides multitasking capa-

bilities. Every FreeRTOS port comes with at least one pre-configured demo application that should build with no errors or warnings [49, 50]. FreeRTOS datasheet recommends that new projects be created by adapting one of these available projects. However, it is not desirable in this project because, in demos, many other irrelevant components were involved that we did not need most of them. So the project was created from scratch.

6.3.3 Realization of hybrid execution for mixed-Critical applications

Figure 6.3 is constructed to verify the FreeRTOS scheduler. It just used an artificial busy loop for each task running on the spike simulator. It executes three tasks. Two tasks are executed as background tasks that run continuously. The third task is a periodic task (RT) with a higher priority. All tasks collect periodic time samples from the platform clock, separated by a busy wait for a predefined number of instructions. The implementation results confirmed that the operating system would alternatingly schedule the background tasks whenever RT task is not executed. As soon as the RT task is ready to execute, it will be scheduled instead of any background tasks until it has finished. Then, one of the background tasks resumes and computes as much as possible and as fast as possible in the slack period. Regarding the scheduling of two background tasks, the operating system's scheduler alternated between them because we assigned both tasks an equal priority. Therefore, on every interrupt, the OS decides which task to execute.

Figure 6.4 shows a real experiment on the Ariane platform. It replaces the busy loop between time samples in the periodic task of the first experiment by a benchmark program. This real example is executed based on the second execution scenario model introduced in section 5.2.3. Here the results show that the same framework also works with realistic workloads. In this case, execution time measurement was performed with binarysearch benchmark as a periodic task.

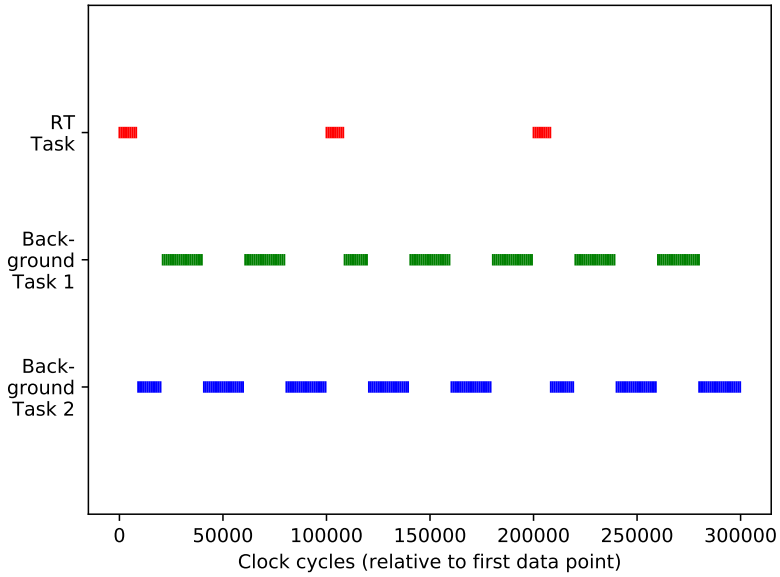

Figure 6.3: Validation of mode switching in spike using two background task and a periodic task

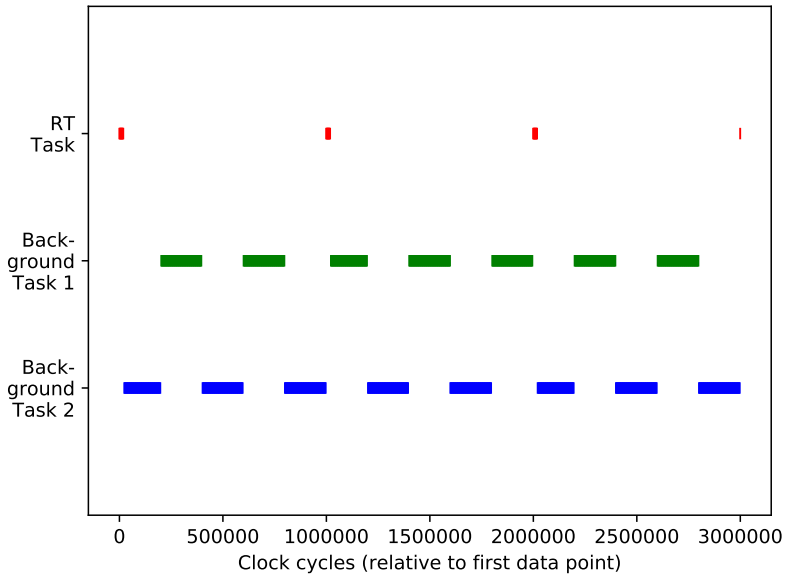

Figure 6.4: Mode switching execution time measurement with *binarysearch* benchmark as a periodic task

6.4 Design a framework for execution time measurment

Figure 6.5 represents different steps that were accomplished to execute the benchmarks and collect the results in the bare metal case. Therefore, in the baremetal case, we want to execute benchmarks from three different memory regions one by one iteratively without involving OS. Accordingly, in this case, the execution was separated into two parts. The first part, which can be seen on top of Figure 6.5 are the stages that trigger the execution on the host side, and the second part, which could be seen at the bottom of the figure 6.5 are the stages that do the execution on the target platform, which is Ariane 64-bit with RISC-V based ISA single-core platform. Hence, we can observe a framework loop on the host computer. So first step is the compilation and linking to a specific memory region. Technically, this is not part of the measurement loop, but conceptually it is in a loop because we perform the compilation and linking process for all benchmarks. Then we load the program (only upload the compiled program to the specific memory region on the platform) and finally execute the program on the target platform. At this point, the host computer waits for the breakpoint from the Ariane platform and then collects the results. These steps were done via gdb (GNU Debugger).

The second part of the execution could be seen on the platform side at the bottom of Figure 6.5. Accordingly, we do the initialization to set up the Ariane platform to execute C code. The execution repeats for some iterations. Subsequently, when the breakpoint signals the end of the measurement, the results are stored on the platform in a memory array. Eventually, we can collect the results when execution is done and return them to the host side inside the gdb. We are now done with one benchmark, and we repeat the experiment for one of the three mentioned configurations in section 6.1.3 for all benchmarks. Consequently, the execution time measurements were performed for all three memory subsystem configurations: cache, DRAM, and SRAM.

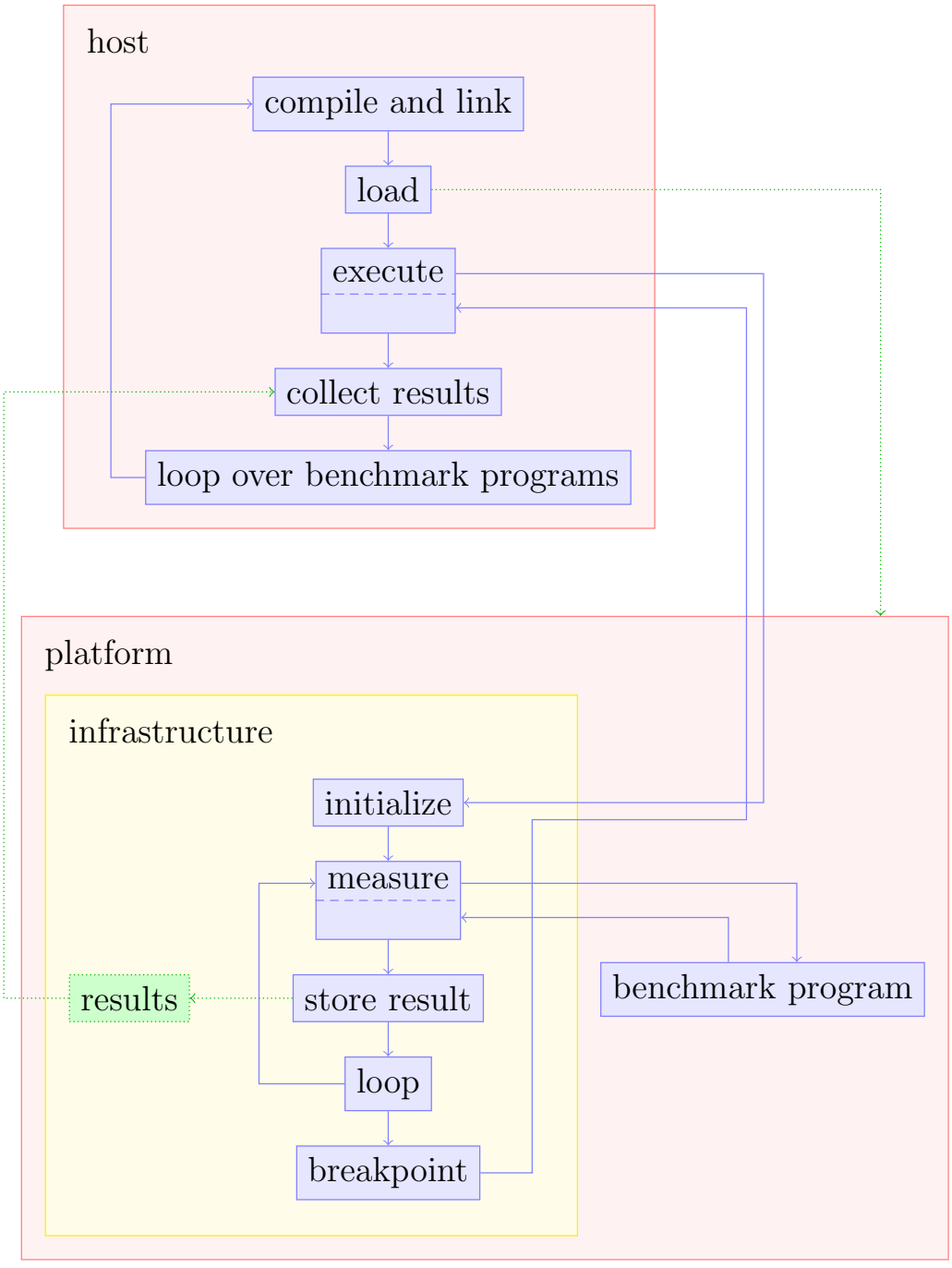

Figure 6.5: Baremetal execution flow

Figure 6.6 depicts different steps that were performed to trigger the execution of different benchmarks using the freeRTOS operating system. So with this setup, the mode switching scenario for executing different tasks with the different real-time requirements will be explained. The host part is the same as the previous figure, so the framework loop on the host computer, compilation, assigning execution to the processor platform, waiting for the breakpoint, and finally collecting the results from the platform with the debugger.

However, we have a periodic task as a real-time program besides two background tasks as two high-performance programs (non-real-time tasks). The operating system also prioritizes the execution of these programs on the platform side. In the previous setup, we had several benchmarks that were required to be executed and measure the execution time, while now there is an entirely different scenario. We have a composition of three tasks that are running according to the real-time requirements of each task. Each benchmark could be executed as a real-time task in this scenario.

Similarly, we have an initialization code on the platform side, but the difference is that it initializes the platform and initializes the operating system. Then we have OS main loop. When we have no real-time task to execute, each of these two background tasks should execute. This process could be seen in the very bottom yellow diagram named "HP-mode". The benchmark task, which is our real-time task, is being executed after getting the priority of the operating system. The real-time benchmark task periodically generates and collects the timestamps. So we collect the time stamp, execute the benchmark, then collect the time stamp again and do this for a group of iterations. The termination task executes once at the beginning, sets the timer, and gives the controller to the kernel. As soon as interrupt execution occurs, it returns the control to the host. Therefore, it does not disturb the measurement. Furthermore, it does not collect any timestamps; this process is managed by freeRTOS.

In the benchmark task and termination task box, the delay functions are operating system calls. The benchmark loop executes uninterrupted by the operating system. So it is required to operate the delay function and then

wait for the OS cycles The example of defining the tasks in Listing 1 also illustrates the process.

Listing 1: FreeRTOS initialization C code assigning static priorities to different tasks

```
    xTaskCreate(
        task_benchmark,           // task function
        "realtime_task",          // task name (debugging only)
        configMINIMAL_STACK_SIZE, // stack size
        NULL,                     // user data pointer
        2,                        // static priority
        NULL);                    // optional, set to NULL
    xTaskCreate(task_background, "background_task_1",
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(task_background, "background_task_2",
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(task_interrupt, "task_interrupt",
        configMINIMAL_STACK_SIZE, NULL, 3, NULL);
```

6.5 Summary

This chapter presented the layers corresponding to design, implementation, and evaluation. First, the hardware is made ready to implement Ariane on the FPGA. Later, the measurement infrastructure is integrated with the hardware implementation to evaluate the execution time for different memory system configurations, including cache, DRAM, or SRAM. To prepare executable binaries from the toolchain perspective for our evaluation platform on the FPGA, we need to choose a compiler and a standard ISA configuration in the RISC-V ecosystem. After that, the linking process enables us to link these binaries to the appropriate memory area based on the real-time requirement of the application. Regarding the software layer, a real-time operating system, FreeRTOS, manages the execution of the tasks based on the priorities on the Ariane single-core platform. The measurement infrastructure could be instantiated in two categories: baremetal and FreeRTOS. In the baremetal case, each benchmark's execution was done in isolation and

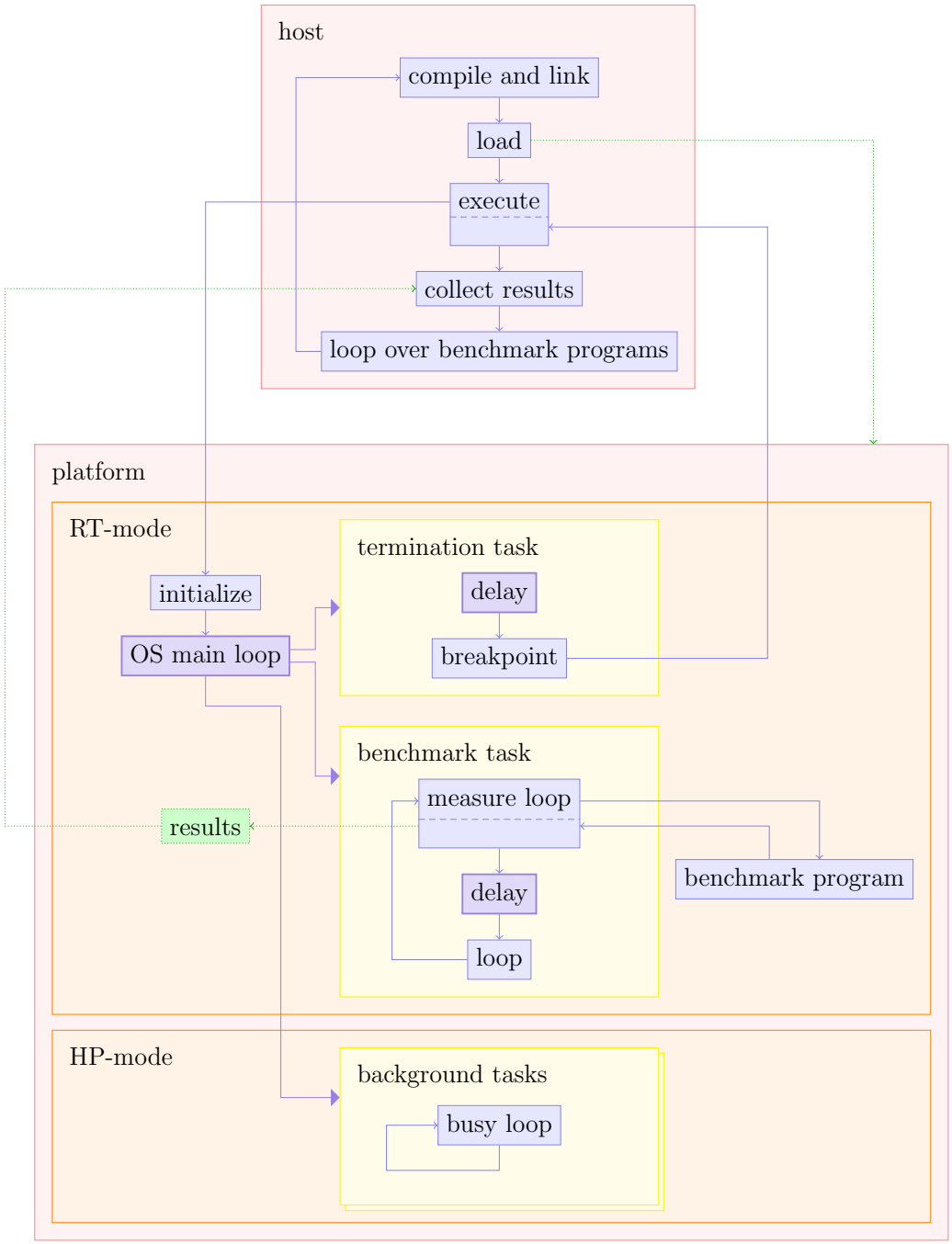

Figure 6.6: FreeRTOS execution flow

without interfering with another task. However, for FreeRTOS, the execution was analyzed in a composition scenario to show if the idea of switching between real-time and high-performance works.

Chapter 7

Evaluation Process and Results

The following sections evaluate the execution time results corresponding to different memory configurations described in Chapter 6. Hence, first section presents the evaluation setup, including a debugger, openOCD, and the benchmark suite. Afterward, Section 7.2 introduces the different memory configurations. Then Section 7.3 describes the evaluation of execution time analysis in composition based on the concept implementation. Finally, Section 7.4 evaluates the whole approach and compares the implemented mode switchable architecture between other available architectures in the RISC-V ecosystem.

7.1 Evaluation Setup

The TACLe benchmark suite [51] as a good collection of embedded benchmarks for real-time systems were selected for our experiments because the TACLe team collected open-source programs, and all of them are self-contained without any dependencies. We need benchmarks to measure the execution times from different memory regions and show the predictable behavior of these different memories. In order to control the execution within the platform, we must devise measurement instrumentation which is not a

trivial task. Therefore, we are inspired by the automated benchmarking framework proposed in [**?**] for the execution time measurement. The measurements were executed using 100 repetitions. Initialization and measurements are performed in a wrapper implemented in an assembler that contains a call to the main function. This function is expected to take no arguments and return a single integer. Furthermore, the results were generated by getting benefits from the built-in Ariane performance counters and transmitting the results using the existing debug link.

7.1.1 TACLE benchmark Programs

To evaluate the platform, we chose to measure the execution of benchmark programs. As a source of benchmark programs, we chose the TACLE benchmark suite. This suite consists of a series of small benchmark programs designed to evaluate embedded systems on a variety of different workloads. The designers' goal was to create benchmarks that closely reflect real world workloads and comprehensively cover different types of load on the target system. Furthermore, the programs are portable and not limited to any particular architecture and do not require the presence of standard library functions. These properties make them well suited for our purpose.

7.1.2 Benchmark Execution process

The execution of benchmarks is driven by an automation framework executing on the host system. In a preparatory phase, it is responsible for building the target memory image. During execution, it initializes the FPGA device, establishes a hardware debug link to the emulated Ariane processor, loads the device with the target memory image, executes the image, and finally extracts the result data to the host system, where it is stored for further analysis. This process is repeated 100 times for each benchmark program. On the target system side, measurement instrumentation is responsible for the benchmark program's initialization, execution, and measurement. The individual steps will now be explained in more detail.

Memory image generation In this step, the platform-independent benchmark program code and the measurement infrastructure code are compiled and linked together into a memory image. The details of linking and memory mapping are described in Chapter 6. The framework builds memory images for all required benchmark programs that are used in the later stages.

Hardware initialization As mentioned earlier, the evaluation was performed on a *Xilinx Kintex 7* FPGA device mounted on the *Genesys 2* development board. Building the bitstream for this device as well as loading it onto the FPGA is performed using the Xilinx Vivado development suite. The task of the automation framework for this step is to pass the correct instructions to Vivado. Listing 3 shows a script for this purpose.

Debug link Establishing the debug link is done using OpenOCD as an intermediary. OpenOCD is a piece of open-source software designed to translate between a high-level software debugger and low-level hardware debug interface. The Ariane distribution includes a configuration file for OpenOCD that connects to the hardware debug interface the Ariane core exposes via a JTAG link. From the host side, OpenOCD acts as a gdb server. This gains us convenient access to the Ariane core using gdb on the host system.

Loading, execution and result extraction All following steps are performed using that gdb interface. Listing 4 shows an abbreviated form of the used gdb automation script. It can be seen that loading and execution can be performed using simple gdb commands. Result extraction is only a little more complex.

Measurement instrumentation The measurement instrumentation executes on the platform. It provides the main entry point for execution. It initializes the platform and executes the benchmark program. It measures this execution and makes the results available to the host system. The mea-

surement instrumentation is implemented in assembler. Listing 5 shows an abreviated form of this code.

The first part of the code consists of platform initialization routines. This code performs the platform-specific steps necessary to prepare the platform to execute the benchmark programs. It also initializes the stack pointer required by the c calling convention. There is also some code specific to the target configuration described below.

The second part consists of the measurement loop. This loop will repeatedly execute the benchmark program and measure the execution time. The measurements are performed by accessing the performance counters of the platform. The results are stored in a memory array that is then accessed by the host system, as explained above.

The instrumentation is able to execute the benchmark multiple times in a loop, but one instance will only execute one benchmark program. In order to execute all benchmarks, the instrumentation has to be built and executed for each benchmark program separately.

7.1.3 Changes for FreeRTOS

The previous section presented the process for executing and measuring programs in the bare-metal configuration. The process for executing and measuring programs in the FreeRTOS configuration is mostly similar. The differences are during linking and in the measurement infrastructure.

Differences in linking During linking, three different components are linked together to form the target memory image: the benchmark program, the operating system core, and the infrastructure code. The benchmark program is unchanged from the bare-metal configuration. The operating system core is not present there. Furthermore, the infrastructure code is heavily modified in the presence of the operating system.

Apart from this, the linker must now support targeting different memory areas depending on the execution mode. In the bare-metal case, the entire memory image was either put in SRAM or DRAM, depending on whether

high-performance or real-time measurement should proceed. However, the core motivation in supporting the operating system is to support tasks in both modes in one execution environment. To support this, since we use static allocation for the tasks, the linker must be responsible for allocating the tasks to the appropriate memory type.

Differences in instrumentation The instrumentation now separates into conceptually different stages. There is code that executes before the operating system takes control, and there is code that is executed as tasks under operating system control.

The first stage contains the platform initialization code as before. In addition, it now needs to contain initialization routines for the operating system itself. As a part of this, it is responsible for initializing the statically defined tasks. The stage finishes by executing the operating system scheduler.

The second stage consists of 4 tasks. The tasks are a watchdog task, the benchmark execution loop, and two background tasks.

The watchdog task's only purpose is to determine when to signal to the host environment that execution has terminated. This is implemented using an operating system wait structure. The signal itself is emitted using the previously explained "ebreak" instruction.

The benchmark execution loop is similar in purpose and implementation to the benchmark execution loop in the bare-metal instrumentation. The difference is that it does not execute all requested iterations back to back. Instead, it executes a few program iterations and then yields control back to other tasks. Measuring and collection of results are performed as above.

The background tasks work by executing an infinite busy loop. The operating system will schedule them alternating for execution. It will also interrupt them whenever a higher priority task (the benchmark loop or the watchdog task) is ready to execute.

7.2 Benchmark Execution on different configurations on the target

In Section 7.1 the evaluation process for bare-metal and FreeRTOS experiments are demonstrated. This section clarifies different configurations utilized to evaluate the behavior of the memory subsystem in terms of timing predictability. The first three subsections cover bare-metal configurations for measuring the execution times in isolation. Finally, the last subsection will explain the configuration to measure the execution time in the composition scenario with the support of FreeRTOS as a management layer.

7.2.1 DRAM with cache

In this configuration, all memory accesses are served in constant time. DRAM with cache configuration will generate baseline results where only non-determinism introduced by microarchitectural features unrelated to the memory interface will be visible. The environment is implemented by executing the benchmark once to load all instructions in the instruction cache and all accessed memory in the data cache. Then several runs are executed without flushing the cache in between. If the cache is large enough to hold all required memory (considered separately for instructions and data), the first run will load the cache, and the subsequent runs will then run entirely inside the cache.

7.2.2 DRAM without cache

In this configuration, memory accesses are served directly by the DRAM. The variance in the generated results will be caused either by the internal features or by the memory backend. This environment can be implemented by turning off the cache. To verify that this works as intended, we can use two indicators. Firstly, the cache miss counters do not count when the caches are disabled. Therefore, no misses should be counted in this case. Secondly, because all memory access is served from the main memory (which is naturally much

slower than the cache), execution times should be significantly larger than with caches enabled. Listing 2 shows a part of the script for enabling or disabling the cache.

Listing 2: Assembler instructions for enabeling or disabeling the first level caches of the Ariane core

```
        # turn off caches
        csrwi 0x700, 0
        csrwi 0x701, 0
        # turn on caches
        csrwi 0x700, 1
        csrwi 0x701, 1
```

7.2.3 SRAM without cache

In this configuration, we injected SRAM into the processor implementation to guarantee a predictable execution time for critical tasks and reduce the execution time variations affected by the memory backend. This environment could be implemented using Xilinx FPGA components and generating the SRAM as an IP. The next step is turning off the cache and forcing the program to be executed from the SRAM. Verification of this implementation is the same as the previous configuration. However, we expect constant execution time with no variation and lower latency than DRAM without cache.

7.2.4 SRAM embedded in FreeRTOS

In this configuration, the aim is to realize the concept of executing real-time and non-real-time tasks. Therefore, the same benchmark programs were executed as real-time tasks on the target platform to have a fair comparison. In addition, FreeRTOS as the management layer was employed to schedule these real-time tasks by assigning higher priorities than the non-real-time tasks in a scenario, including one real-time task and two non-real-time tasks. Like the previous SRAM configuration, we expect constant execution time

with no variation and, in the best case, with minimum timing overhead produced by freeRTOS compared to SRAM without cache, which was just an execution time analysis in isolation.

7.3 Results

This section presents the benchmark execution time results provided by executing from the different memory subsystems. In section 7.2, different target configurations were introduced. We analyze the execution time distribution for each benchmark based on the memory configuration. Furthermore, we assess the efficiency impacts of mapping the switchable architecture design to the implementation. We use the benchmark programs as tasks in RT mode.

7.3.1 Average Execution Time Comparison

Figure 7.1 shows the average execution times of the different benchmark programs executed in clock cycles on a logarithmic scale. It is clearly visible that the execution time of benchmarks varies significantly. However, for all benchmarks, the configuration for DRAM without cache takes significantly longer to execute than other configurations. The DRAM with cache (hot cache) configuration is always faster than the other configurations. The SRAM configuration represents better execution time results than DRAM without cache but is still slower than hot cache. In the end, the benchmarks executed within freeRTOS show very close results to SRAM, which is the best case we expected when running a benchmark in the composition scenario.

7.3.2 Result interpretation

In order to interpret the variance introduced by the memory subsystem, the comparison was performed using the execution times collected as histograms and as a Cumulative Distribution Function (CDF) for each benchmark program. CDF of the observed execution times maps execution time onto the probability of an execution time less than or equal to that time to occur. As

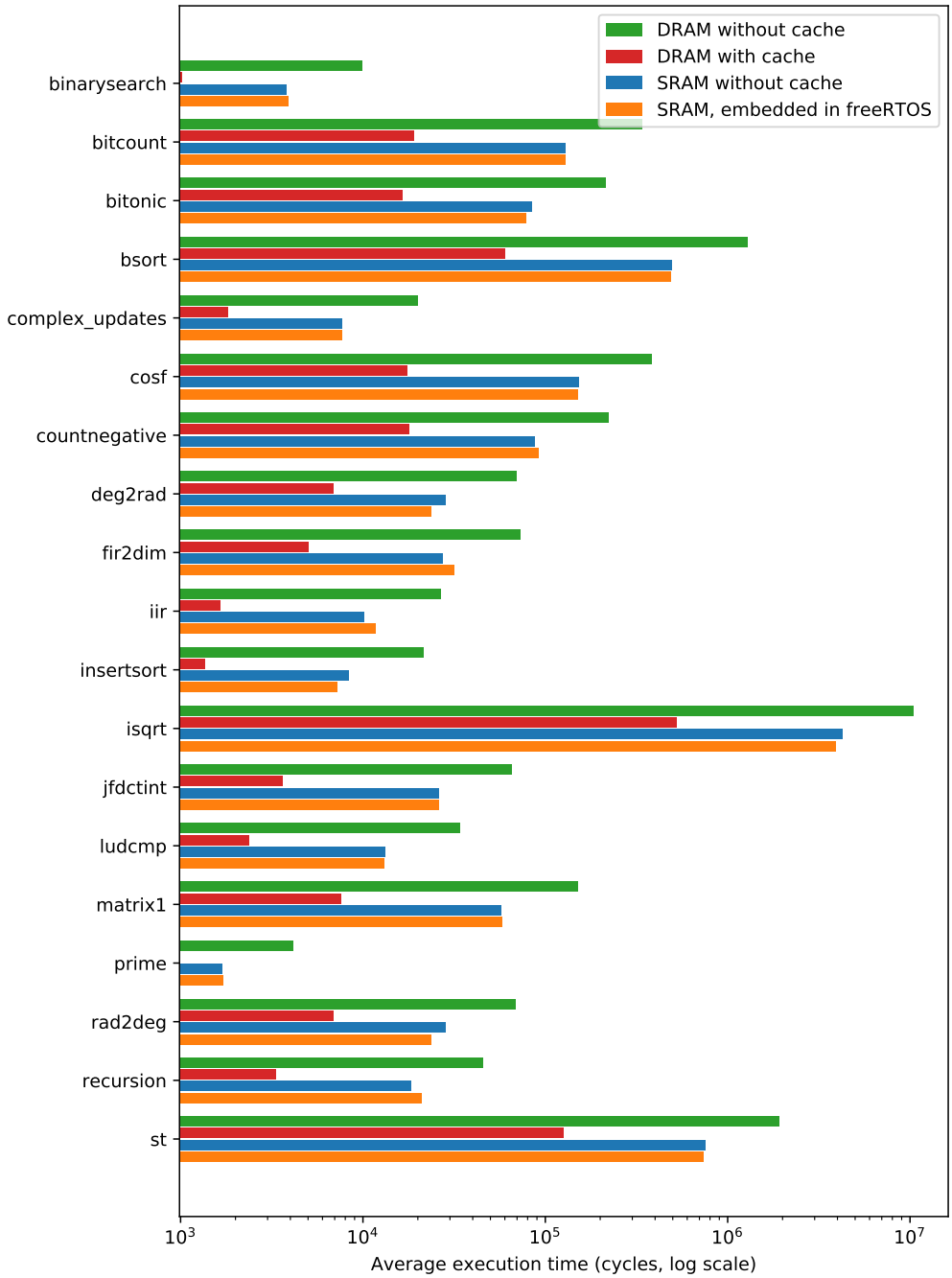

Figure 7.1: Average execution times in clock cycle for kernel benchmarks of TACLE benchmark suite.

a result, the CDF raises from 0 to 1 in an interval from the lowest to the highest observed execution time. If these measurements are distributed over many different execution times in an interval, the resulting curve will appear relatively smooth. However, if the results are concentrated in a few distinct times, the curve will have significant steps and be very non-smooth. Because it is not possible to represent all plots, we provide two benchmark figures as examples. *Binarysearch* as a low latency benchmark and *st* as a high latency benchmark is selected. However, all benchmark results are provided in the appendices.

7.3.3 EXECUTION TIMES FOR BINARYSEARCH BENCHMARK

Figure 7.2 shows the measurement results for the *binarysearch* benchmark. The figure consists of four plots representing the results of DRAM with cache (hot cache), DRAM without cache, and two SRAM-involved configurations, respectively. Each of these plots contains a histogram and CDF as the data representations. It should be noted that the y-axis uses different scales for the histogram on the left side and the CDF on the right side. In addition, the x-axes use different scales in the different plots due to different benchmark binary sizes in the histograms. The hot cache and SRAM configurations run in constant time, so they consists of a single step and has maximum steepness. Thus, we can observe predictable behavior for the last three configurations. However, the DRAM without cache configuration creates a pattern of some narrow spikes in the histogram that is broadly distributed over a large range, so the CDF is smooth. For the last configuration, freeRTOS was introduced as the management layer and executed a mixed workload model, but this does not change the predictability of the real-time workload in comparison with the SRAM in isolation.

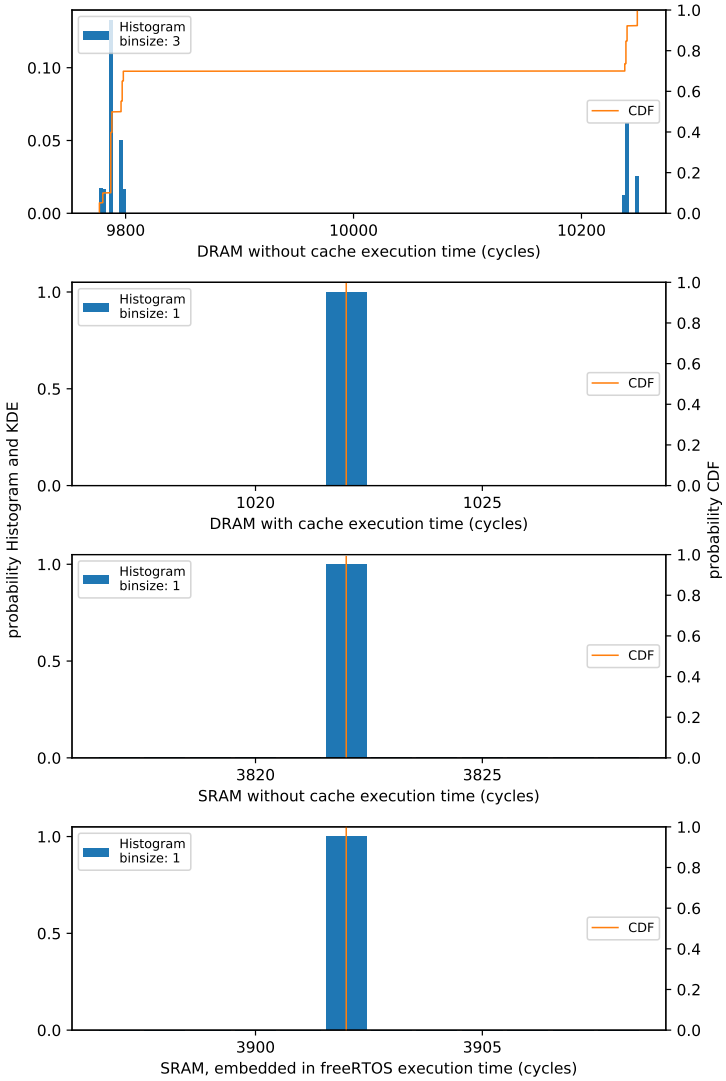

Figure 7.2: Execution time analysis on Ariane processor for binarysearch benchmark

7.3.4 execution times for Cosf benchmark

Figure 7.3 depicts the measurement results for the *cosf* benchmark. It is visible that keeping the cache hot for this benchmark is not possible due to the large binary size. Therefore, the histogram creates a pattern of some spikes, and the CDF of this configuration has a few large steps that prooves the unpredictable execution time for DRAM with and without cache configs. The SRAM configurations run in constant time, so the CDF consists of a single step and has maximum steepness. So these results show that the introduced approach eliminates the unpredictability of caches and guarantees the predictable execution time. Additionally, execution time from SRAM without cache is faster than DRAM without cache.

7.3.5 execution times for st benchmark

Figure 7.4 shows the measurement results for the *st* (Statistics calculations) benchmark. The figure consists of four plots representing the results of DRAM with cache (hot cache), DRAM without cache, and two SRAM-involved configurations, respectively. For this measurement result, the hot cache and SRAM configurations run in constant time, so the CDF consists of a single step and has maximum steepness. However, the DRAM without cache configuration is broadly distributed over a large range, so the CDF is smooth. We can still observe a little unpredictability for the SRAM configuration due to a single OS interrupt. This can happen for some large benchmarks. However, these are only 10 clock cycles.

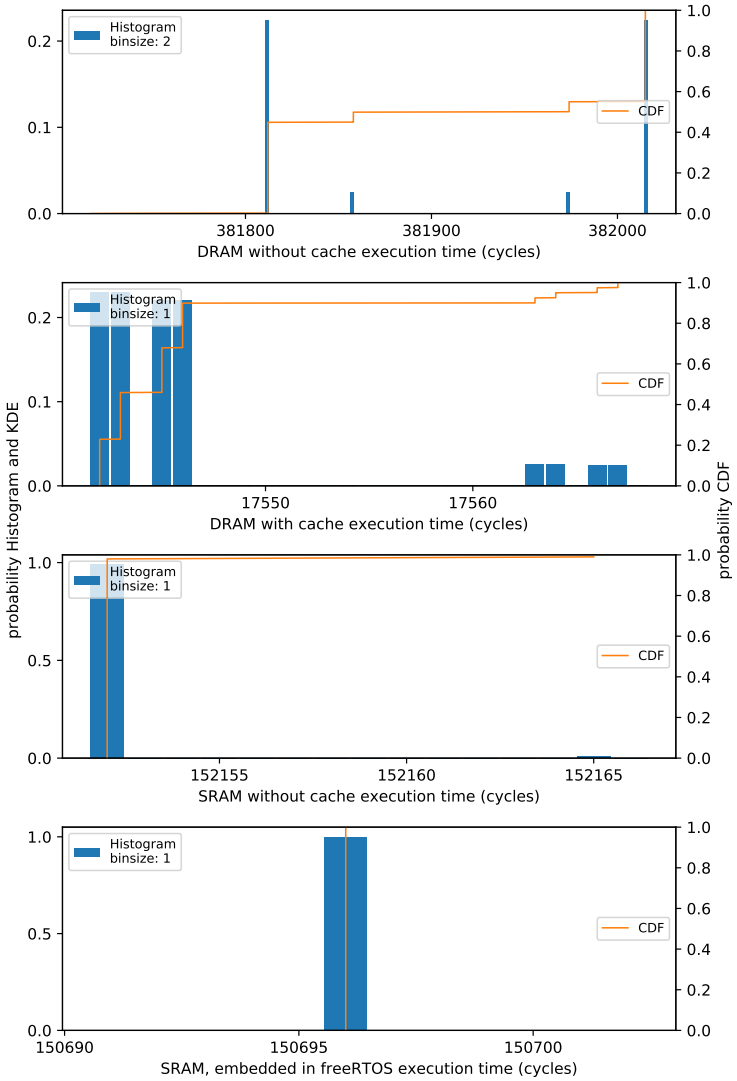

Figure 7.3: Execution time analysis on Ariane processor for Cosf benchmark

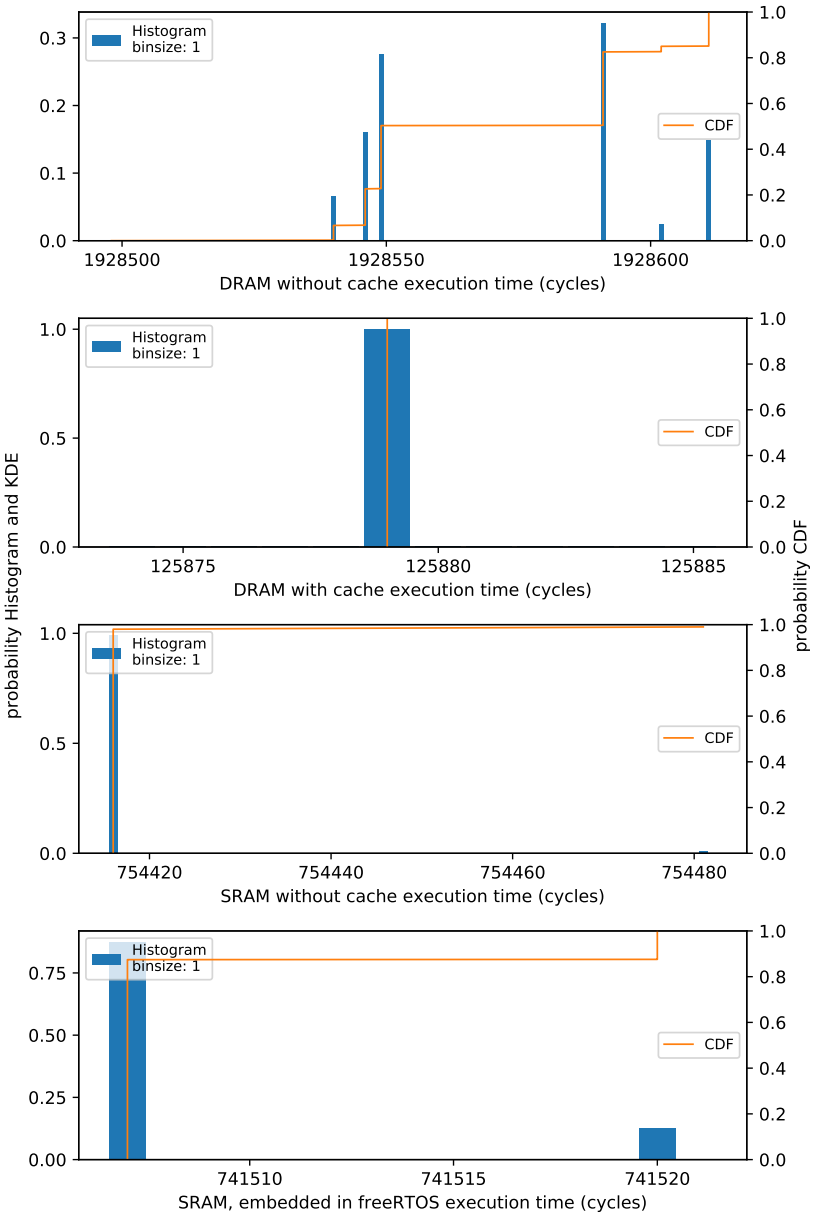

Figure 7.4: Execution time analysis on Ariane processor for st benchmark

7.4 Discussion

In this thesis, we had some limitations, such as picking a single-core platform in the RISC-V ecosystem, the capability of supporting an operating system, and not focusing on scheduling policies intensely. Therefore, analyzing state of the art played an essential role in finding a suitable solution based on the limitations. In addition, for our RT mode, we need enough SRAM memory which is heavily dependent on the FPGA. The *Genesys 2* FPGA board by Digilent provides 1MB SRAM, which is a comfortable margin for executing our real-time tasks in the RT mode.

The results presented in the previous section and other results in the appendices prove that the idea of executing tasks with different timing requirements on a single-core platform in the RISC-V domain works. In the early results, we observed some unpredictability due to some compiler optimization artifacts during compilation and some operating system artifacts during execution. The first issue was solved by modifying the in-lin assembly block. The second issue was solved by configuring three individual configs with different clock rates to extend the ticks for the OS interrupts based on the execution speed of programs in the benchmark suite. Then we observed entirely predictable execution times for 90% of benchmarks except two of them. These two are *insertsort* and *st*.

Although these two benchmarks show predictable behavior in SRAM configuration, we have observed a small unpredictability in SRAM embedded in the FreeRTOS config. The reason for this behavior is a single operating system interrupt during execution. We observed this issue for some other slow benchmarks, but the problem was solved by increasing the intervals in which the OS interrupts the program. Therefore we do not have any interruptions during execution. However, for these two benchmarks, we still have a single interrupt during execution, which is why we do not have a single bar in the resulting plot. Altogether this unpredictability is in the margin of 20 clock cycles.

In Chapter 3, the state-of-the-art comparison was introduced. Two of those platforms (Ariane and FlexPRET) are more noteworthy because, on

the one hand, they use the RISC-V ISA, where this project implementation is based on. On the other hand, one of them is the platform implemented and extended in this work. Therefore, to sum up, in this section, we compare these two platforms with the proposed implementation based on the defined scope to check in case of functioning the idea of mode switchable architecture design how well the proposed work could be comparable with these architectures in the RISC-V ecosystem. More detail information could be found in Table 7.1 and Figure 7.5

The FlexPRET supports a kind of mode switching in hardware but just for hard-real-time and soft-real-time applications and is not suitable for embedded high-performance applications without supporting any operating system. Ariane is not designed as a timing predictable processor, so it could not support mode switching at all. However, by modifications performed in the Ariane hardware, supporting real-time applications is now possible. Although there is still some room for improvement, my proposed architecture can support real-time workloads and switch between two modes thanks to operating system support.

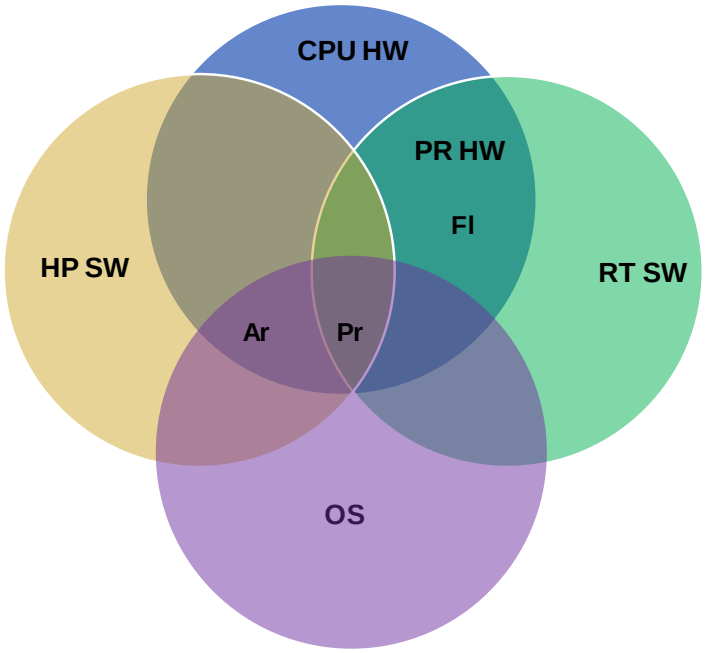

Figure 7.5: Comparison of proposed work with other processors in the RISC-V ecosystem.

Table 7.1: Comparison of RISC-V based processors based on the requirements of switchable architecture design with the proposed work

requirements / **Platform**	**Pred. execution times**	**General OS support**	**HP feature**	**Execution mode switching**
CVA6 (Ariane)	-	+	+	na
FlexPRET	+	-	+	+
proposed	+	+	+	+

7.5 Summary

In this chapter, we have evaluated the application and execution models proposed in this thesis. First, Section 7.1 discussed the evaluation setup to assess the execution times for four different configurations on the FPGA board with and without FreeRTOS. The evaluation considered various TACLE benchmarks for the real-time mode in our proposed switchable architecture in the RISC-V ecosystem. In Section 7.2 different configurations for measuring the execution times from DRAM, SRAM, cache, and in at last composition scenario from different memory regions (SRAM and DRAM) were introduced. Next, the corresponding results to the configuration introduced in Section 7.2 have been evaluated, and we have compared the results by showing the execution time distributions for all memory subsystem configurations explained before. Section 7.4 then represents that the idea works and describes some challenges that we faced to realize the idea. In addition, we explained the reason for a couple of exceptions that the predictability did not completely fulfill. To end up, we have compared the proposed architecture with comparable available other RISC-V single-core platforms to show the significance of such a design in the RISC-V domain and the strength of our platform in supporting both real-time and non-real-time workloads in a single-core system.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis started with observing the significance of considering the execution of applications with different timing requirements on a unique processor in the RISC-V ecosystem. The contributions of this thesis address the challenges of running distinct embedded software programs on a platform, where the functionality can be categorized along with their criticality into real-time or non-real-time.

When several software applications with different levels of criticality are integrated into a device, multiple challenges need to be considered and managed. In order to answer ***RQ1***, we identified the memory subsystems that make timing predictability difficult in Section 2.2.1 and 5.3. We have proposed several requirements that a processor and/or platform should meet to support both real-time and high-performance workloads in Section 2.2.2 to address ***RQ2***. Based on these requirements, we have assessed different platforms and processors to recognize to what extent current architectures can potentially support the handling of mixed workloads. Then, we have proposed a switchable single-core processor based on the Ariane processor in the RISC-V ecosystem, which can handle mixed workloads at runtime. This switchable processor meets predictable processor design requirements while providing respectable performance for high-performance applications. The

main design features are avoidance of interferences in the memory subsystem and predictable execution time behavior while maintaining acceptable performance on the processor side. Moreover, operating system support is integrated into the system to handle the interruption and mixed timing workloads in Section 6.3 as a solution to ***RQ3***. The execution time measurement was illustrated entirely in Section 7.3 to address the ***RQ4***.

8.2 Future work

There is some room for improvement of this work in the future. We have defined some constraints that can be removed to extend this work.

Extension to multi-core implementation

Although this work has advantages concerning the cost compared to multi-core implementation, making this approach applicable as a multi-core platform could be considered a promising future extension. From the hardware perspective, on the one hand, one challenge could arise in memory management for different shared caches, SRAMs, and DRAMs. On the other hand, benefiting from an isolated processor for executing real-time or non-real-time software could be another fascinating, challenging aspect.

Tightly integrated SRAM to the processor

The current SRAM configuration is designed to solve the problem of unpredictability execution inside DRAM. Accordingly, it is now connected to the processor through the generic peripheral bus. However, there would have been better optimization in SRAM implementation to provide a dedicated high-speed bus like the caches to improve the future execution duration to something near cache rather than something between DRAM and cache.

Improvement in operating system support and scheduling complicated scenarios

In this thesis, we introduced two possible scenarios that fit into the concept of this work. In addition, we tried to implement one of the mentioned scenarios in Section 6. However, it is possible to extend the features in FreeRTOS in different ways, which was not possible due to a shortage of time within this Ph.D. project. For instance, in another experiment, we should be able to add

larger benchmarks with longer execution times. This may cause some unpredictability due to the operating system artifacts on the software side or the amount of available predictable memory on the platform side. Furthermore, it should be possible to schedule more benchmark programs as an extension in the scheduling part that was not in this project's scope.

References

[1] J. Bin, S. Girbal, D. G. Pérez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core cots architectures," in *Embedded Real Time Software and Systems (ERTS2014)*, 2014.

[2] P. Ittershagen, K. Grüttner, and W. Nebel, "An integration flow for mixed-critical embedded systems on a flexible time-triggered platform," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 4, pp. 1–25, 2018.

[3] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[4] P. P. Project. (2021) Ariane github. [Online]. Available: https://github.com/openhwgroup/cva6

[5] A. Alsheikhy, S. Han, and R. Ammar, "Delay and power consumption estimation in embedded systems using hierarchical performance modeling," in *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. IEEE, 2015, pp. 34–39.

[6] Y. Jung and L. P. Carloni, "σvp: Host-gpu multiplexing for efficient simulation of multiple embedded gpus on virtual platforms," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[7] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez, "Energy efficient hpc on embedded socs: Optimization techniques for mali

gpu," in *2014 IEEE 28th International parallel and distributed processing symposium*. IEEE, 2014, pp. 123–132.

[8] R. Hegde, G. Mishra, and K. Gurumurthy, "Software and hardware design challenges in automotive embedded system," *International Journal of VLSI Design & Communication Systems*, vol. 2, no. 3, p. 165, 2011.

[9] G. Buttazzo, "Research trends in real-time computing for embedded systems," *ACM SIGBED Review*, vol. 3, no. 3, pp. 1–10, 2006.

[10] G. Chen, "Resource management in real-time multicore embedded systems: Performance and energy perspectives," Ph.D. dissertation, Technische Universität München, 2016.

[11] A. Alsheikhy, R. Ammar, and R. Elfouly, "An improved dynamic round robin scheduling algorithm based on a variant quantum time," in *2015 11th International Computer Engineering Conference (ICENCO)*. IEEE, 2015, pp. 98–104.

[12] J. A. Stankovic and K. Ramamritham, "What is predictability for real-time systems?" 1990.

[13] D. Grund, J. Reineke, and R. Wilhelm, "A template for predictability definitions with supporting evidence," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[14] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange *et al.*, "Building timing predictable embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, pp. 1–37, 2014.

[15] R. Kirner and P. Puschner, "Time-predictable computing," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2010, pp. 23–34.

[16] L. M. Pinho, E. Quinones, and A. Marongiu, *High-performance and time-predictable embedded computing*. River Publishers, 2018.

[17] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk *et al.*, "The hipeac vision," *Report, European Network of Excellence on High Performance and Embedded Architecture and Compilation*, vol. 12, 2010.

[18] R. Giorgi, "Scalable embedded systems: Towards the convergence of high-performance and embedded computing," in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. IEEE, 2015, pp. 148–153.

[19] A. A. Alsheikhy, "High performance embedded systems," 2016.

[20] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in cots-based multicore systems," in *2013 8th IEEE international symposium on industrial embedded systems (SIES)*. IEEE, 2013, pp. 39–48.

[21] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 269–279.

[22] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 183–192.

[23] K. K. Chang, "Understanding and improving the latency of dram-based memory systems," Ph.D. dissertation, Carnegie Mellon University, 2017.

[24] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[25] (2021). [Online]. Available: https://riscv.org/about/

[26] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA," University of California at Berkeley Berkeley United States, Tech. Rep., 2019. [Online]. Available: https://riscv.org/specifications/

[27] L. Project. (2019) LLVM Download Page. [Online]. Available: http://releases.llvm.org/download.html#10.0.0

[28] lowRISC Community Interest Company. (2019) lowrisc home page. [Online]. Available: https://www.lowrisc.org

[29] M. Poorhosseini, W. Nebel, and K. Grüttner, "A compiler comparison in the risc-v ecosystem," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–6.

[30] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[31] "RISC-V Exchange: Cores & SoCs," accessed FILL ME IN. [Online]. Available: https://riscv.org/exchange/cores-socs/

[32] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020.

[33] "BOOM RISC-V CPU," accesssed 2022-02-21. [Online]. Available: https://github.com/riscv-boom/riscv-boom

[34] datasheet. (2021) Raspberry pi 4. [Online]. Available: https://datasheets.raspberrypi.org/cm4/cm4-datasheet.pdf

[35] S. Mittal, "A survey on optimized implementation of deep learning models on the nvidia jetson platform," *Journal of Systems Architecture*, vol. 97, pp. 428–442, 2019.

[36] M. O. Aboelhassan, O. Bartik, and M. Novak, "Embedded multi-core systems for mixed-critical applications with rpmsg protocol based on xilinx zynq-7000," in *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. IEEE, 2017, pp. 162–167.

[37] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad *et al.*, "Virtual execution platforms for mixed-time-criticality systems: The compsoc architecture and design flow," *ACM SIGBED Review*, vol. 10, no. 3, pp. 23–34, 2013.

[38] B. Akesson and K. Goossens, *Memory controllers for real-time embedded systems*. Springer, 2011.

[39] S. Goossens, B. Akesson, M. Koedam, A. B. Nejad, A. Nelson, and K. Goossens, "The CompSOC design flow for virtual execution platforms," in *Proceedings of the 10th FPGAworld Conference on - FPGAworld '13*. Stockholm, Sweden: ACM Press, 2013, pp. 1–6.

[40] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[41] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: A time-predictable microprocessor," *Real-Time Systems*, vol. 54, no. 2, pp. 389–423, 2018.

[42] F. Kluge, M. Schoeberl, and T. Ungerer, "Support for the Logical Execution Time Model on a Time-predictable Multicore Processor," p. 7.

[43] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 101–110.

[44] M. Fletzer, "SPEAR2 - an improved version of SPEAR," Thesis, 2008, accepted: 2020-06-30T09:53:10Z.

[45] P. Ittershagen, "Application modelling and performance estimation of mixed-critical embedded systems," Ph.D. dissertation, Universität Oldenburg, 2018.

[46] P. Giusto, G. Martin, and E. Harcourt, "Reliable estimation of execution time of embedded software," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001.* IEEE, 2001, pp. 580–588.

[47] "Genesys 2," accessed 2020-08-21. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/genesys-2/start

[48] "Vivado Design Suite," accessed 2020-08-21. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[49] R. Barry. (2016) Mastering the FreeRTOS Real Time Kernel- A Hands-On Tutorial Guide. [Online]. Available: https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

[50] A. W. Services. (2017) The FreeRTOS Reference Manual-API Functions and Configuration Options. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

[51] "TACLe Benchmarks," accessed 2020-08-21. [Online]. Available: https://github.com/tacle/tacle-bench

Appendices

Appendix A

Timing Comparison Plots

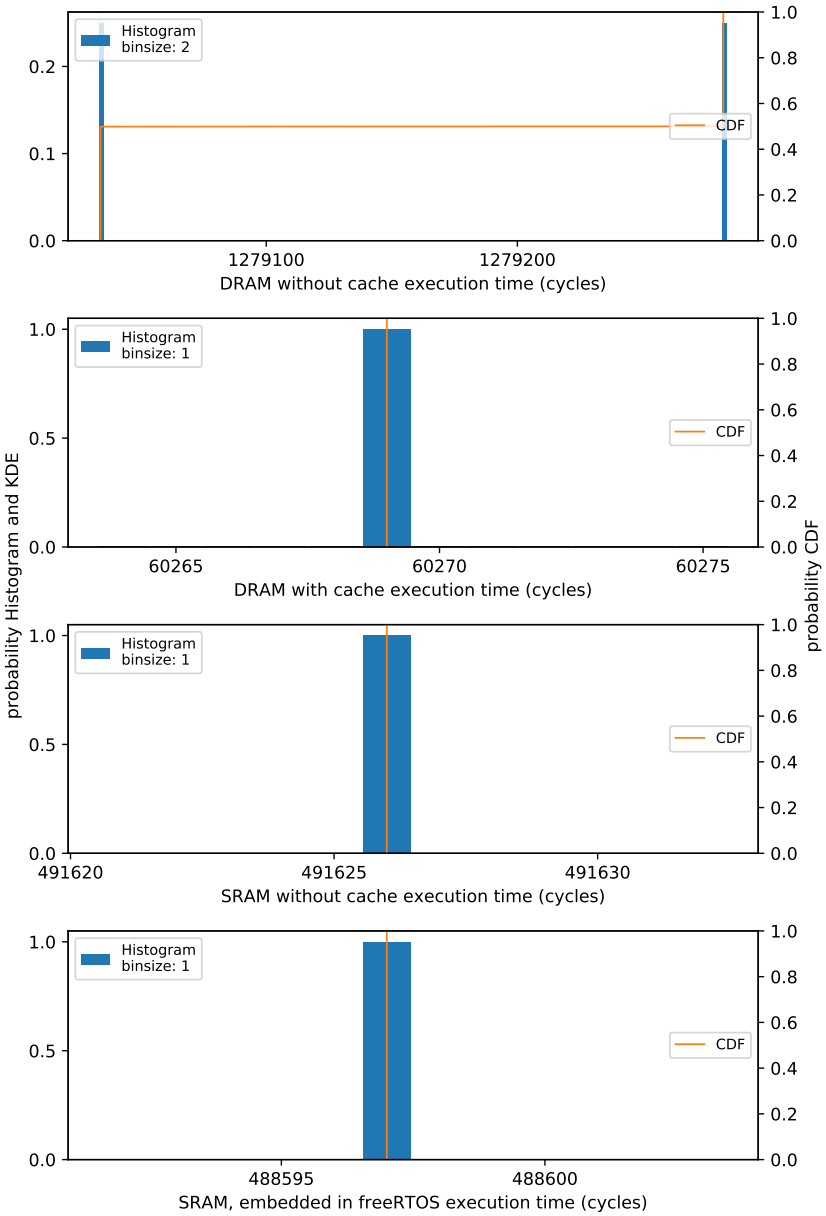

Figure A.1: quad_bsort

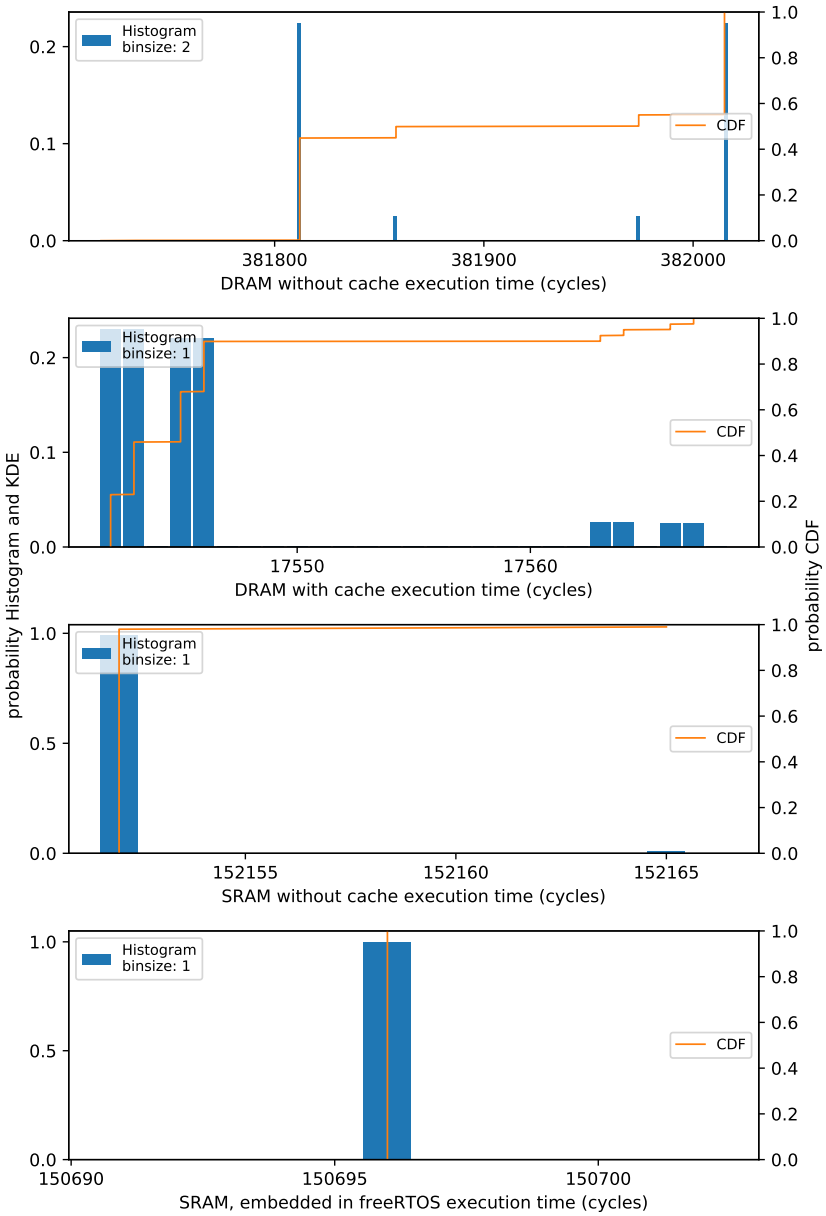

Figure A.2: quad_cosf.pdf

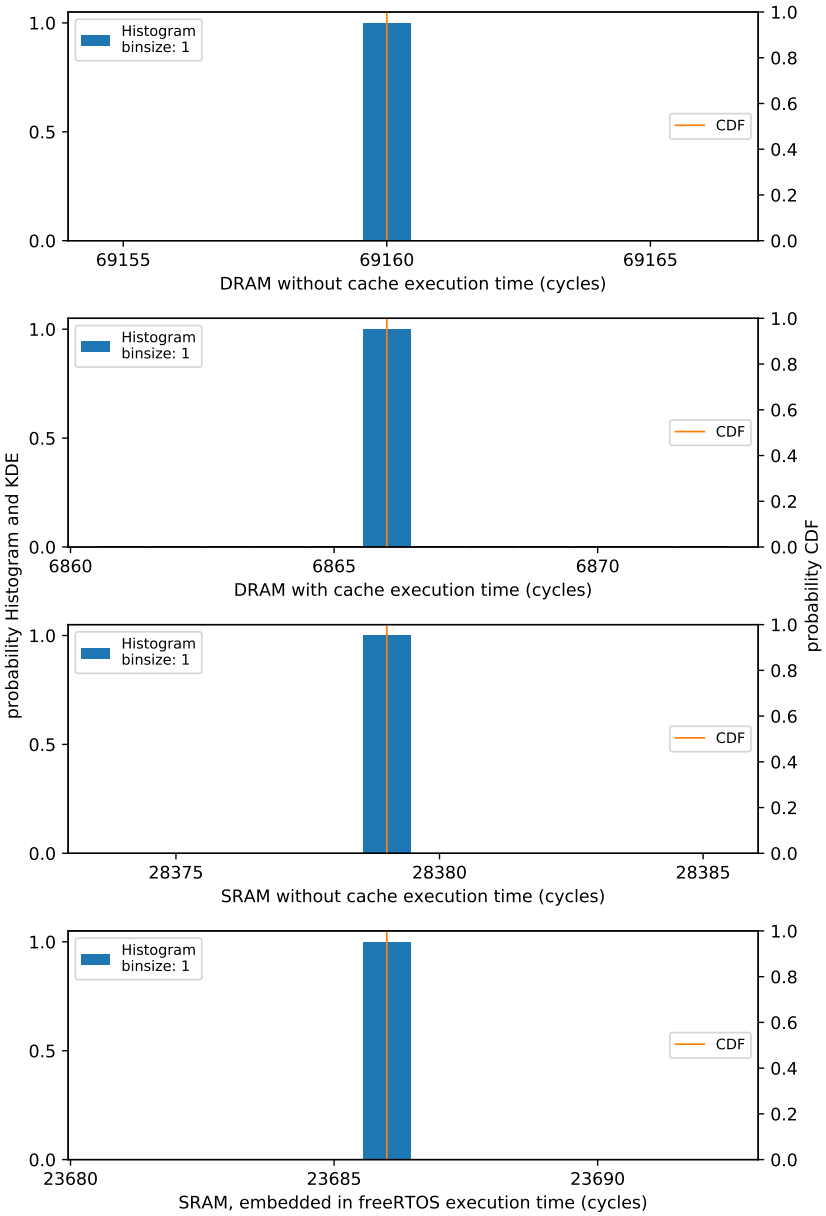

Figure A.3: quad_rad2deg.pdf

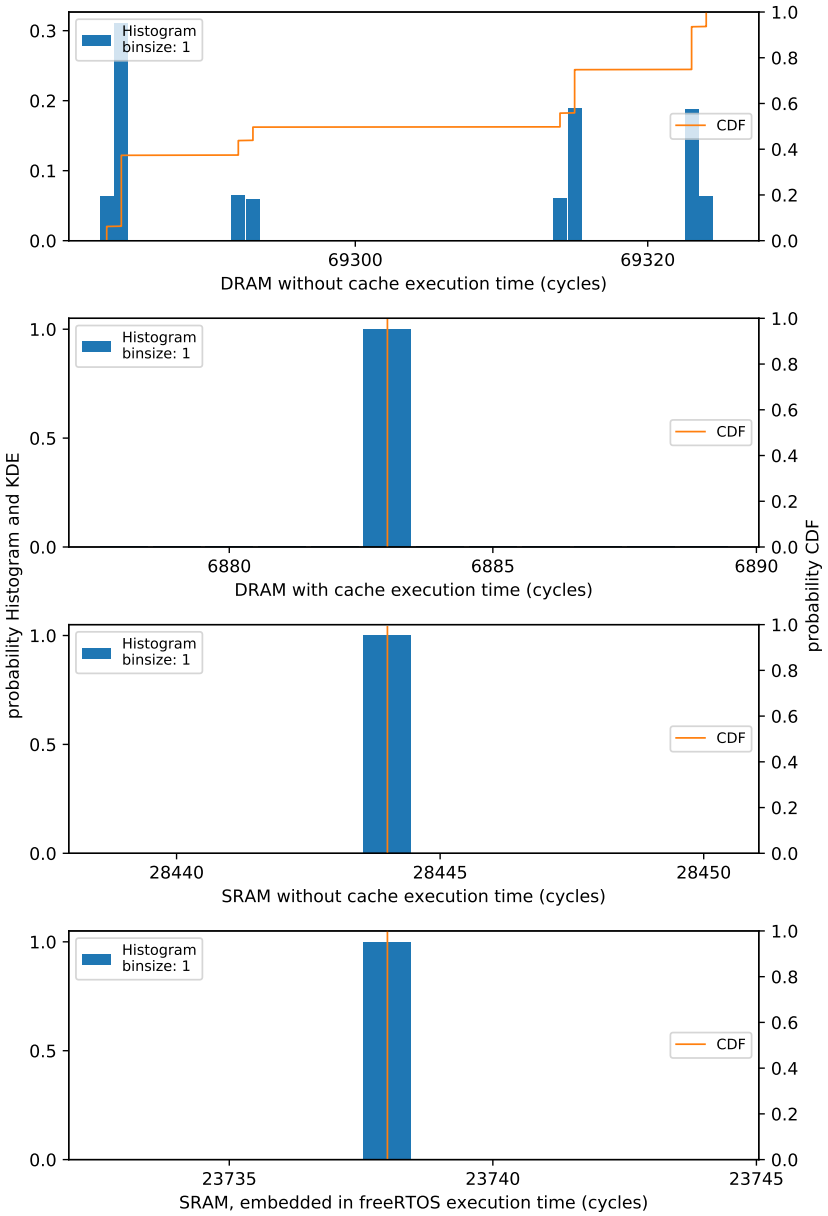

Figure A.4: quad_deg2rad.pdf

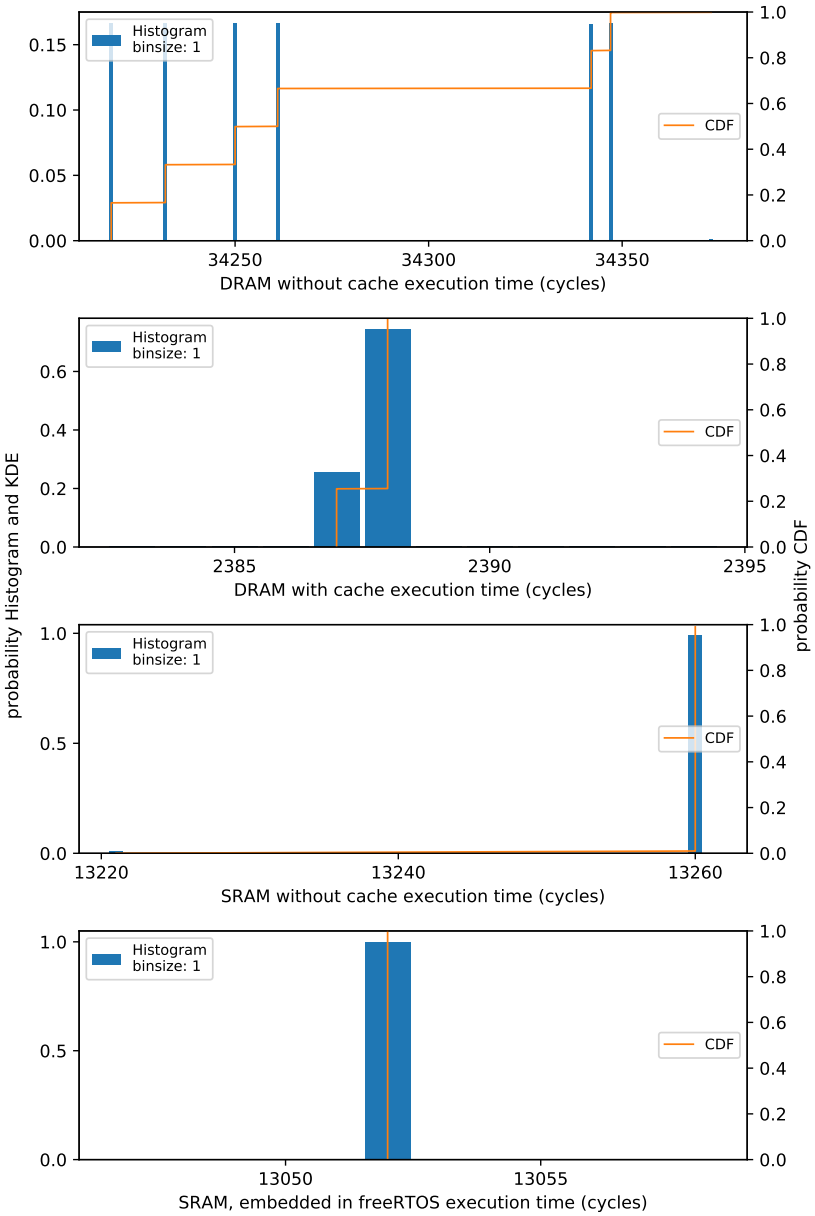

Figure A.5: quad_ludcmp.pdf

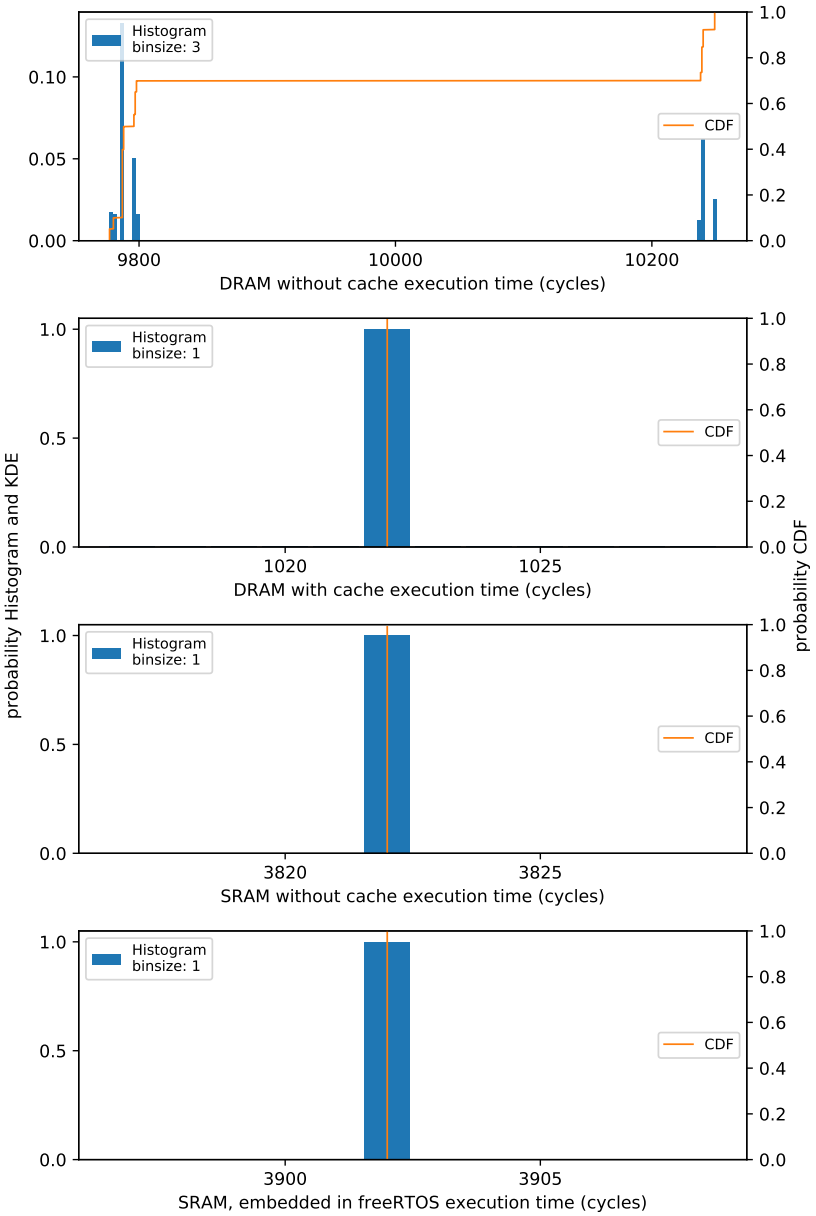

Figure A.6: quad_binarysearch.pdf

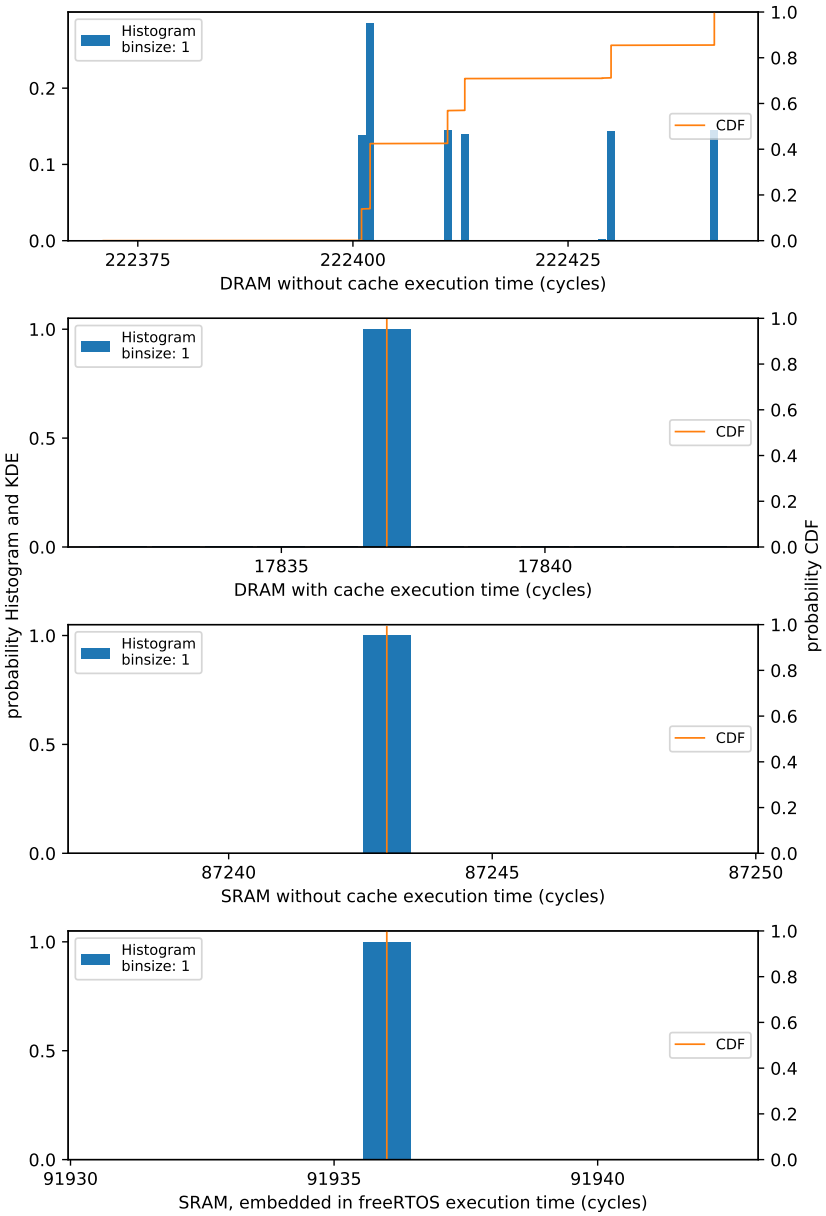

Figure A.7: quad_countnegative.pdf

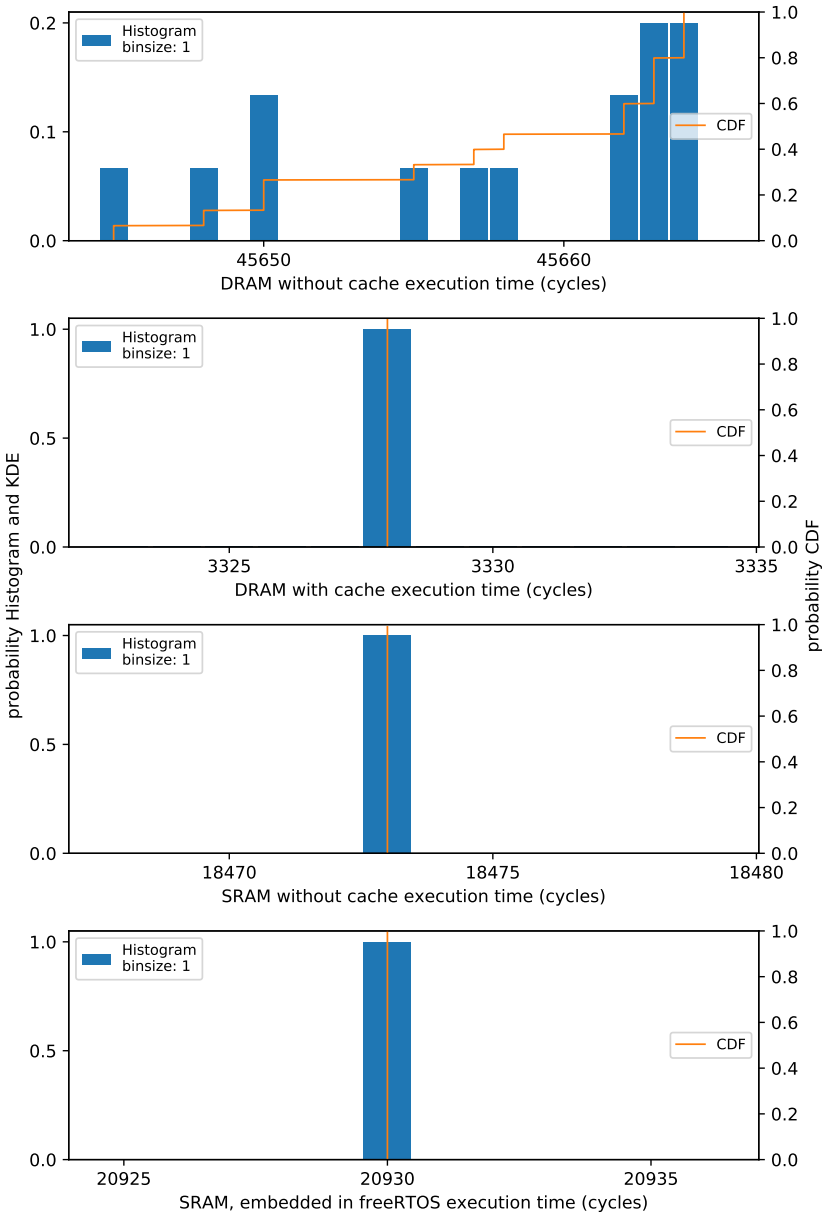

Figure A.8: quad_recursion.pdf

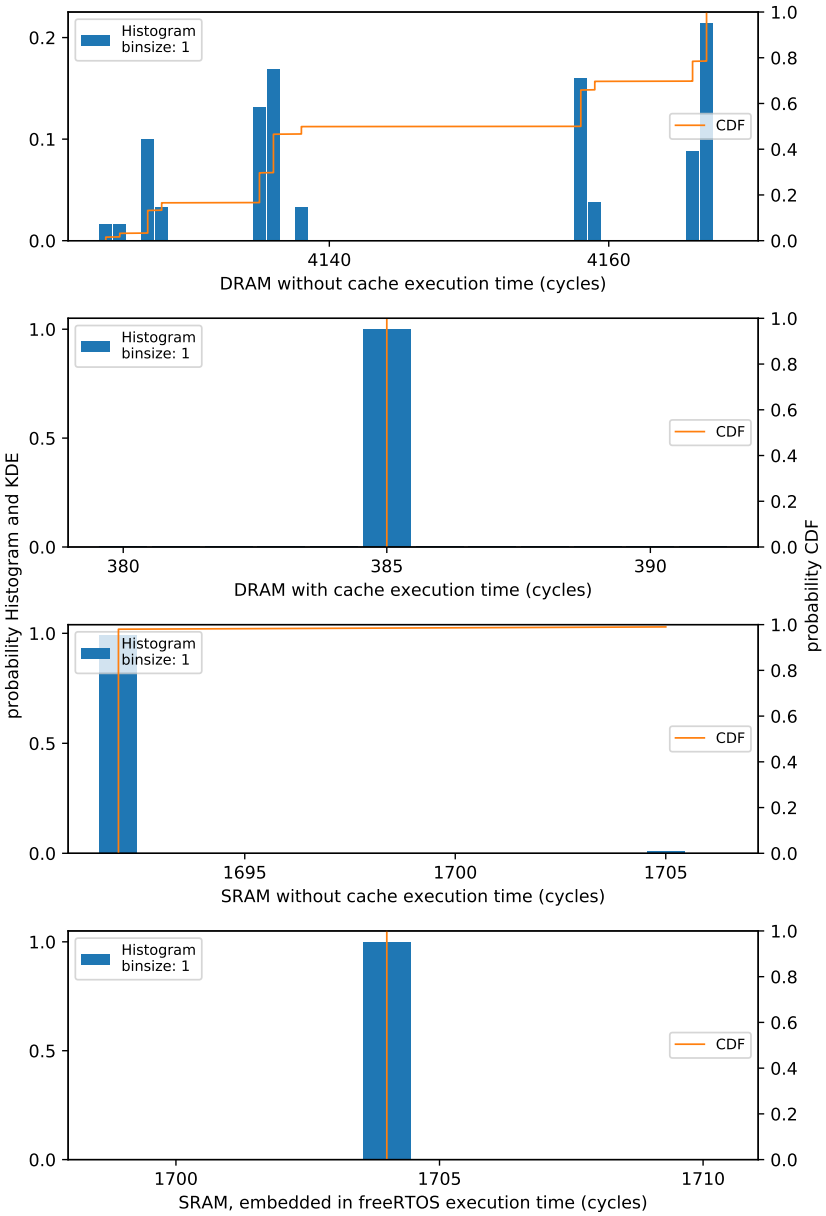

Figure A.9: quad_prime.pdf

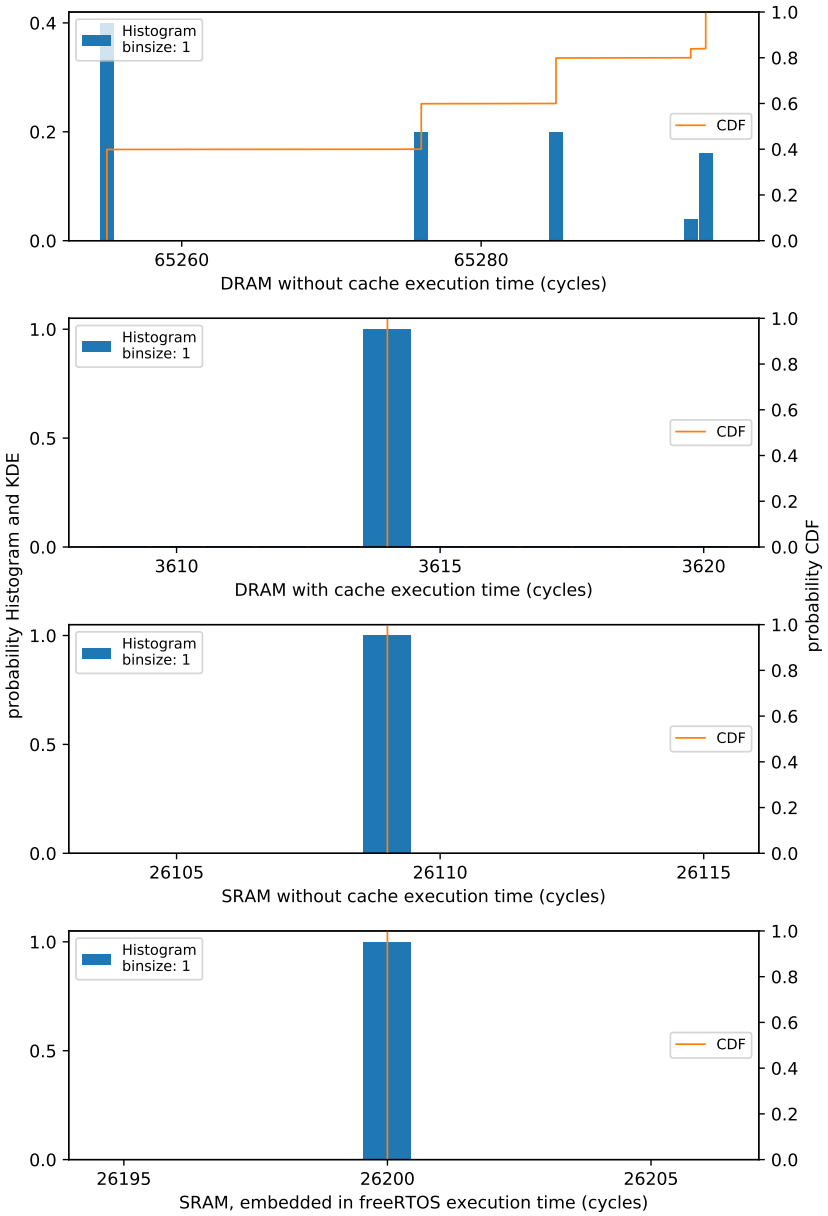

Figure A.10: quad_jfdctint.pdf

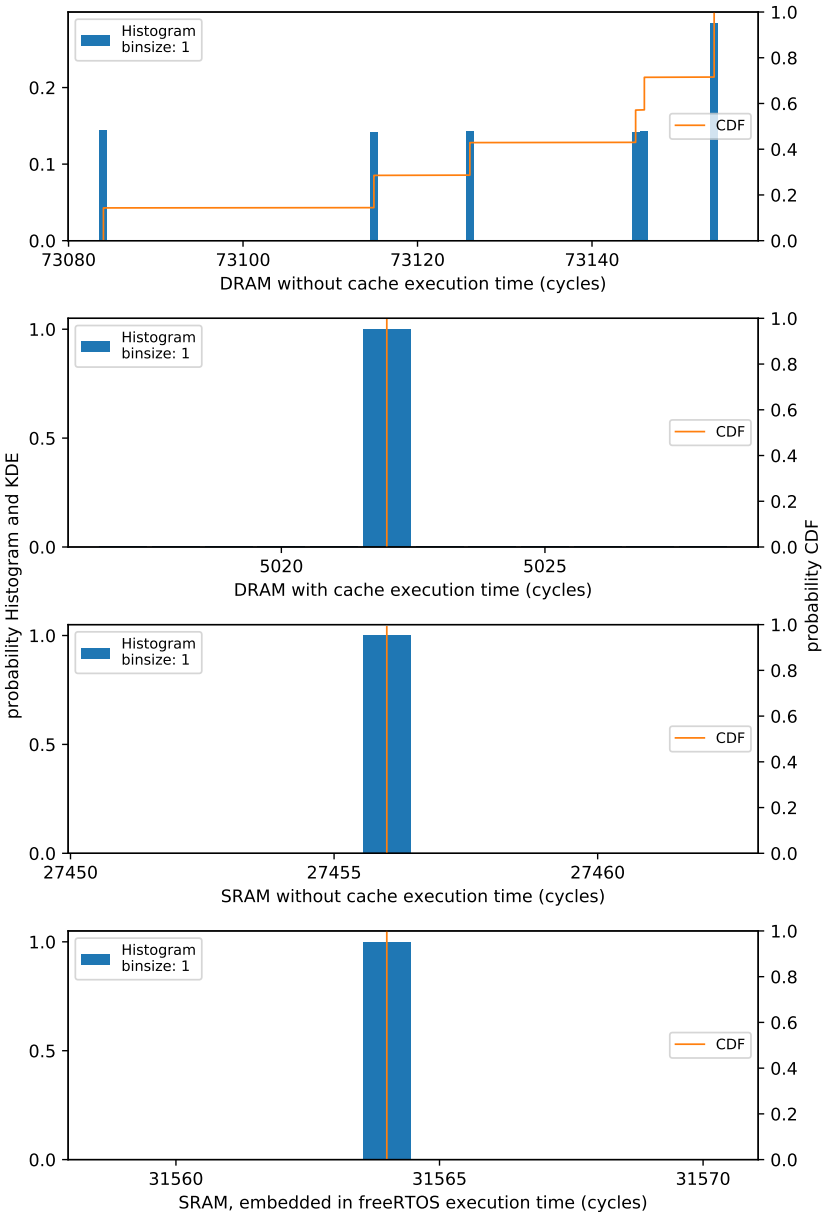

Figure A.11: quad_fir2dim.pdf

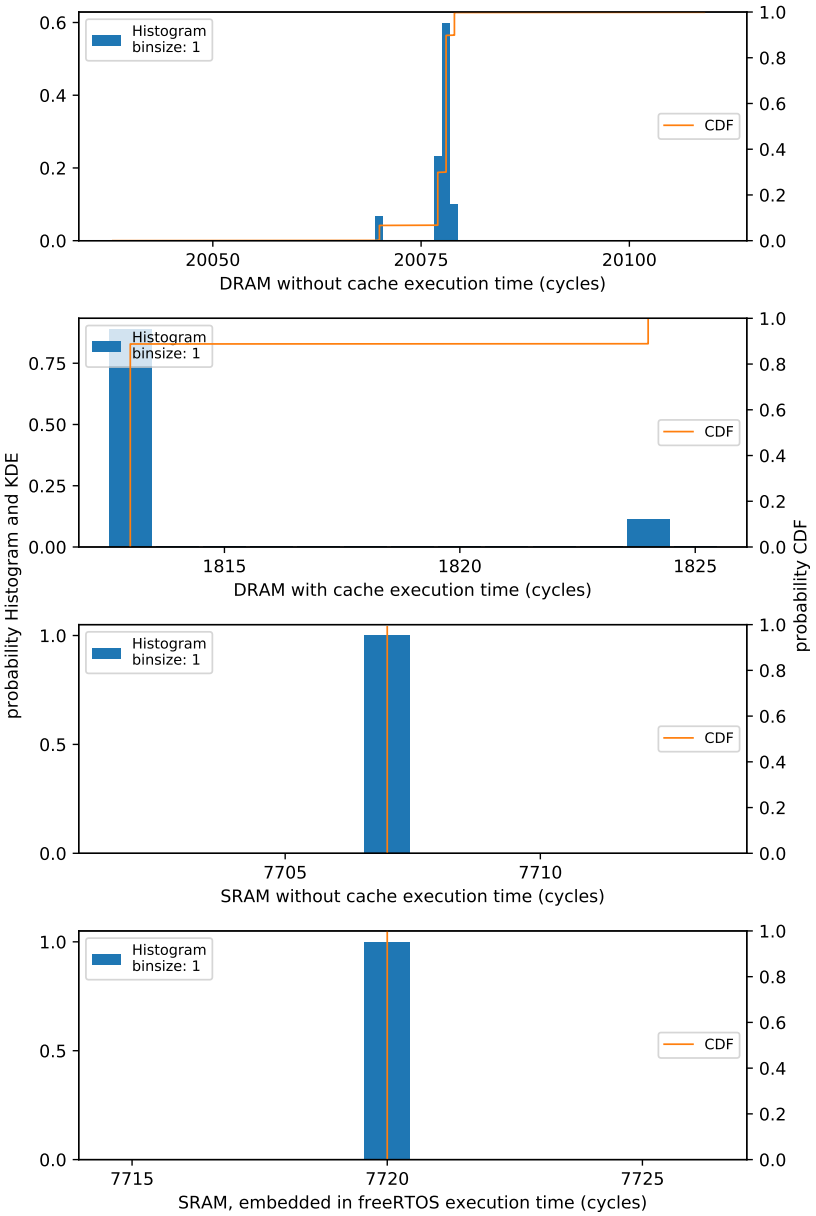

Figure A.12: quad_complex_updates.pdf

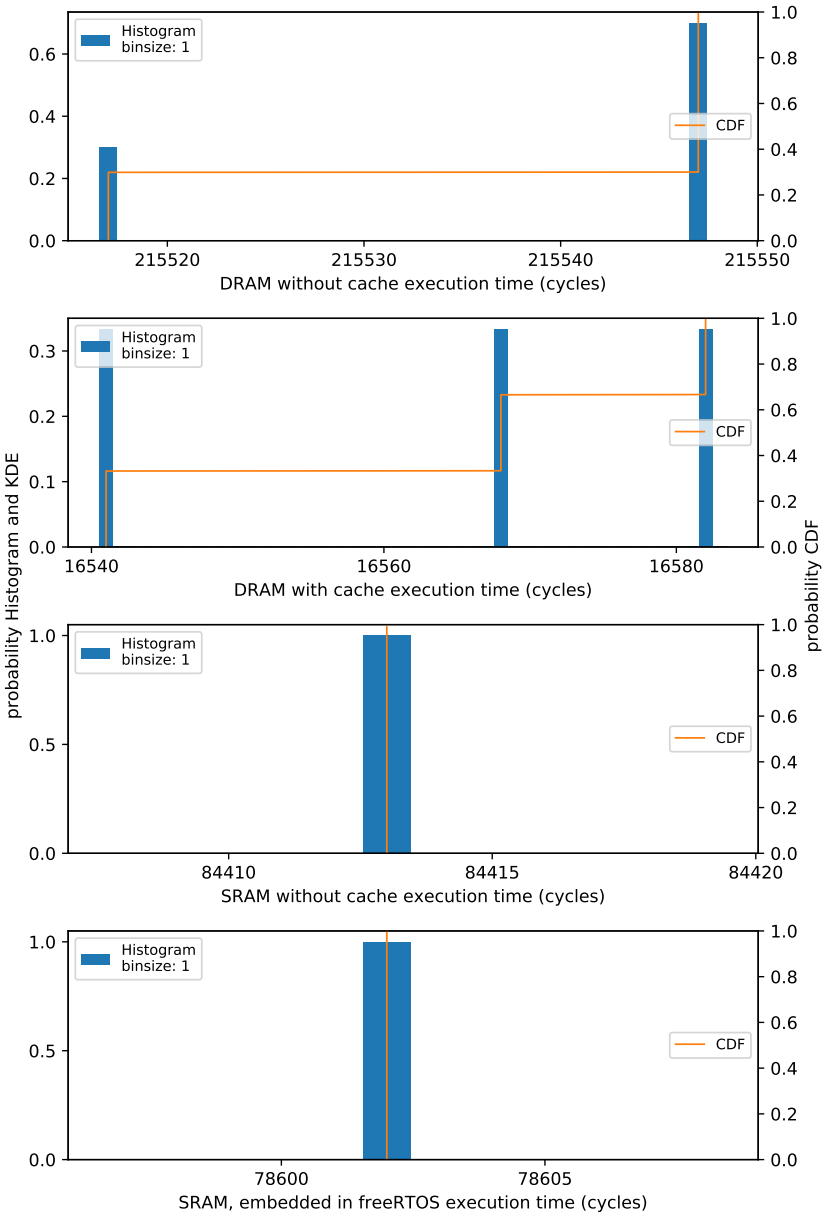

Figure A.13: quad_bitonic.pdf

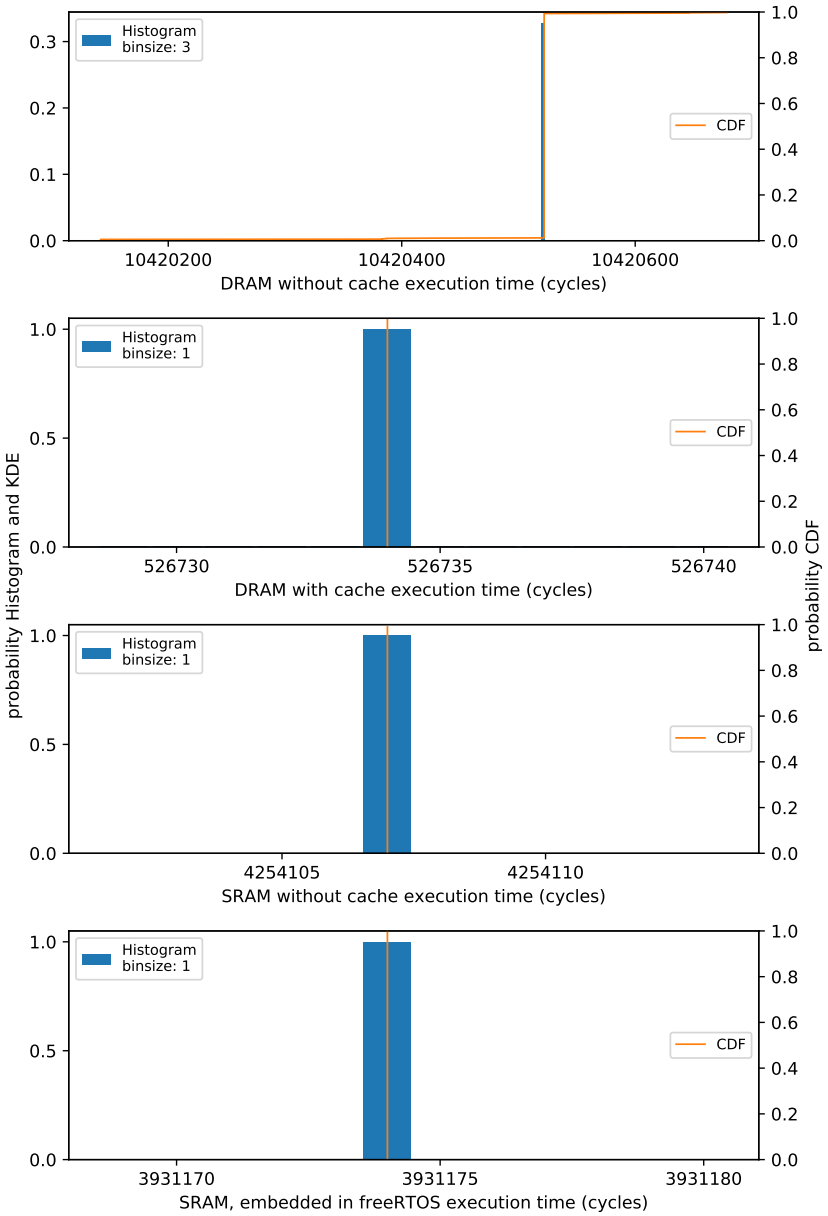

Figure A.14: quad_isqrt.pdf

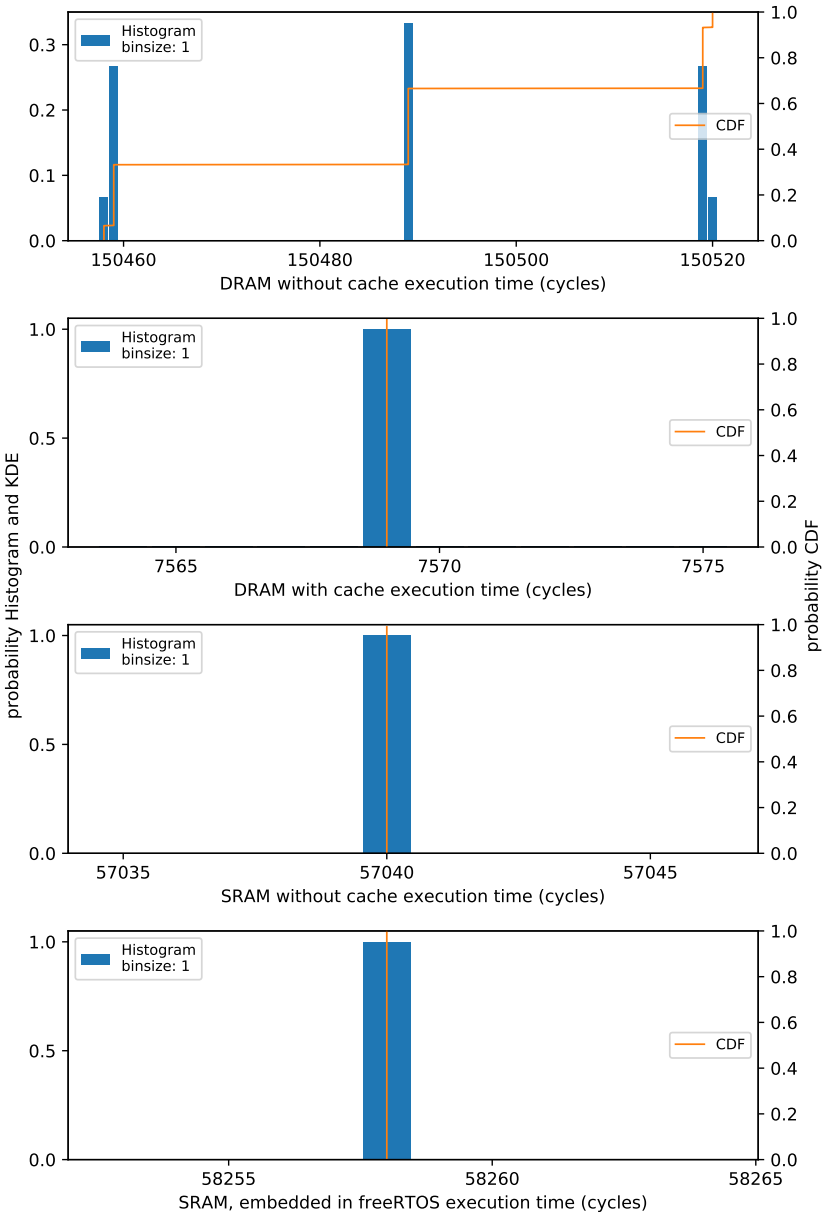

Figure A.15: quad_matrix1.pdf

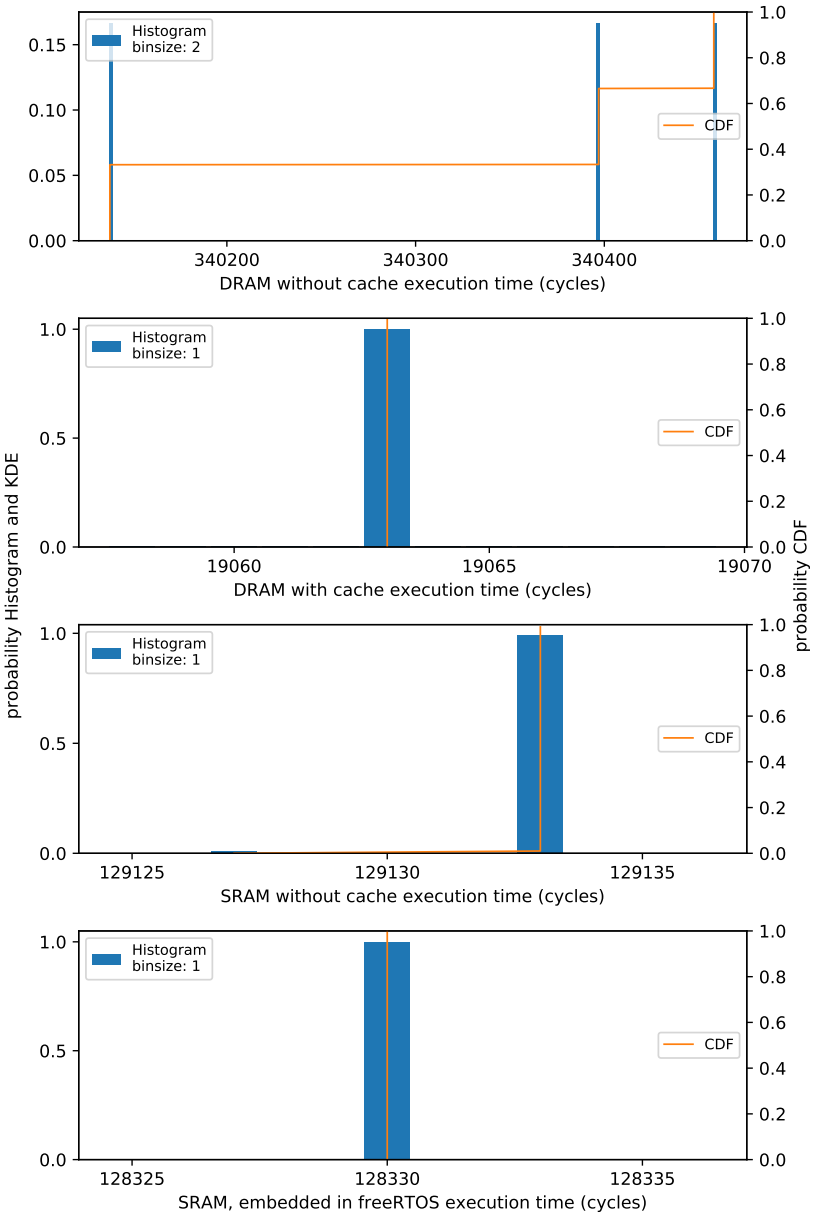

Figure A.16: quad_bitcount.pdf

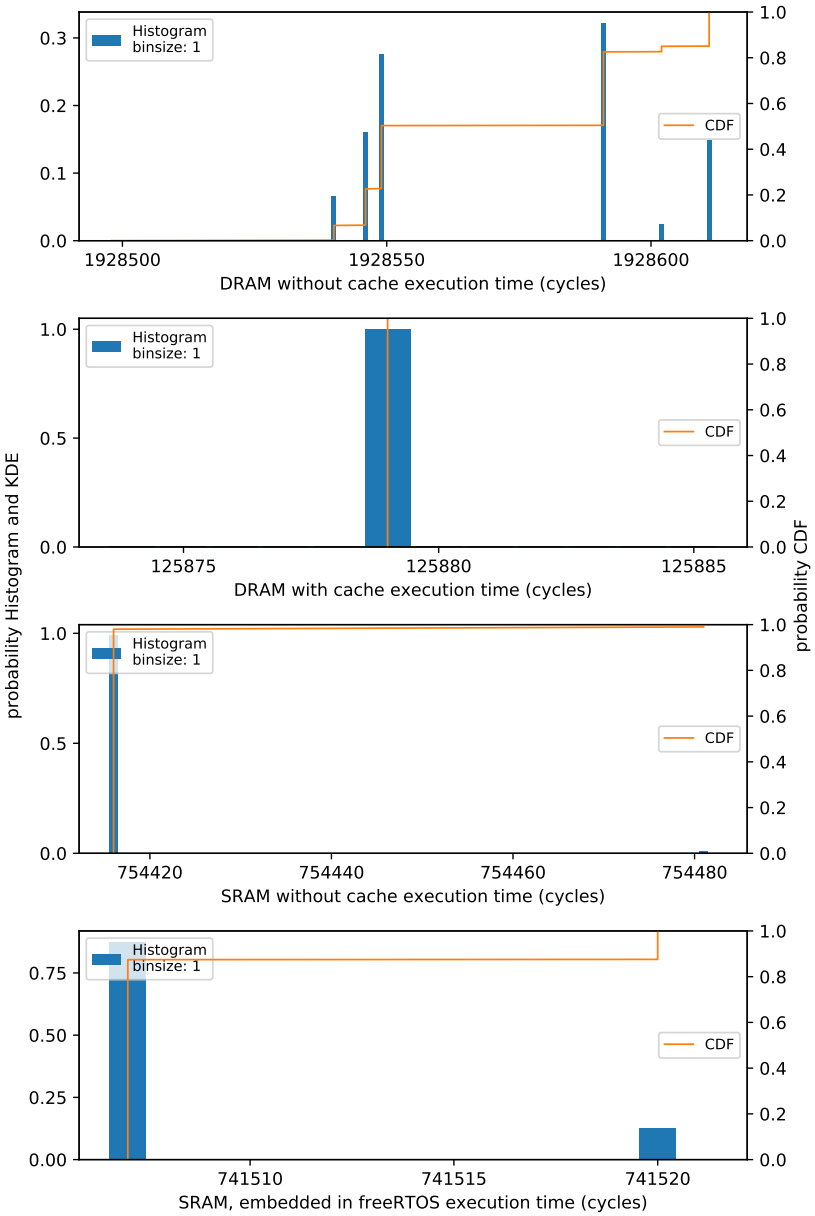

Figure A.17: quad_st.pdf

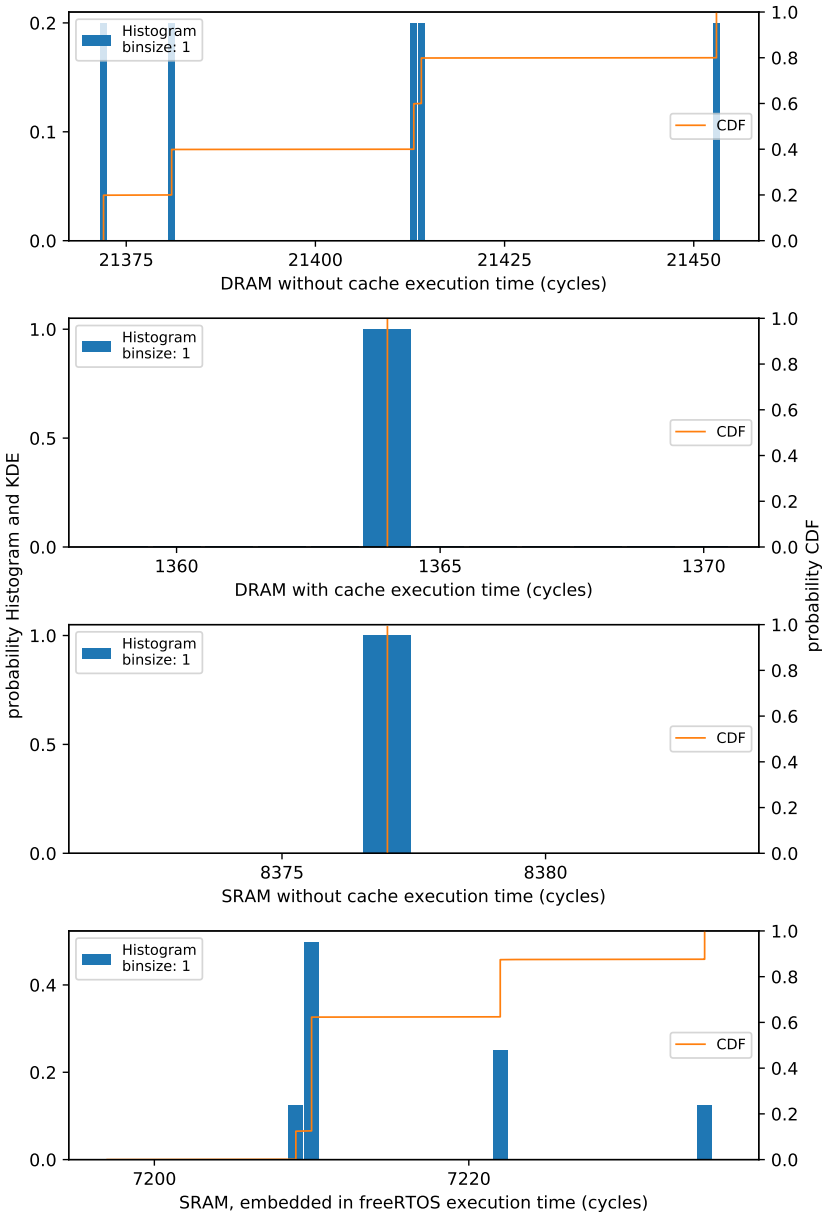

Figure A.18: quad_insertsort.pdf

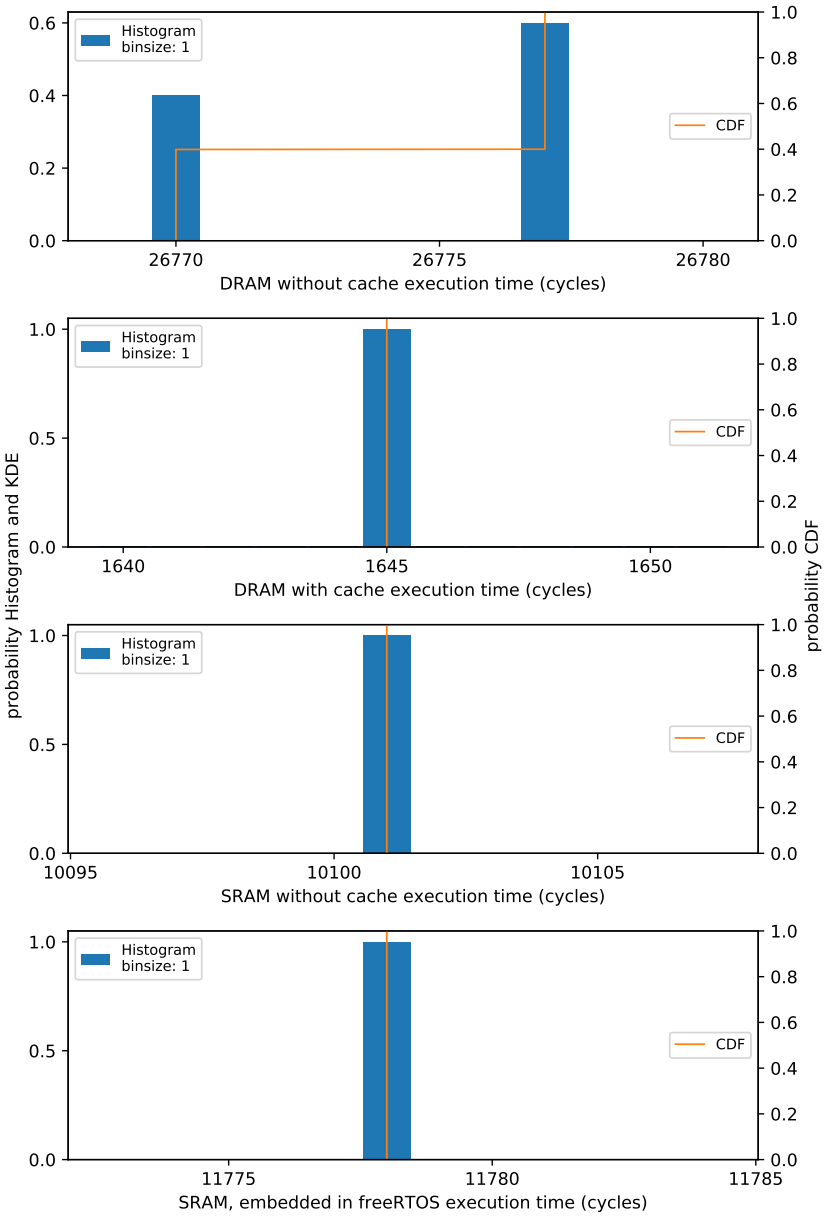

Figure A.19: quad_iir.pdf

Appendix B

Execution Visualization Plots

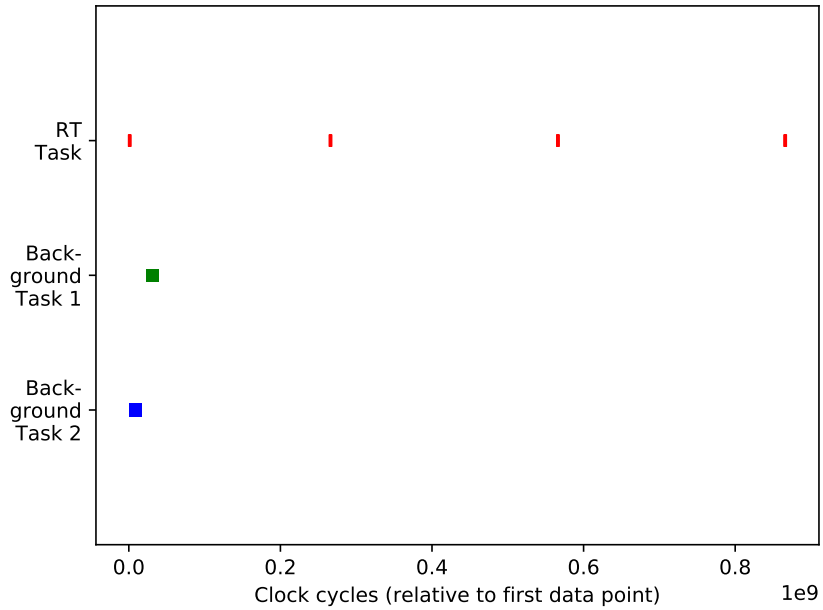

Figure B.1: bsort

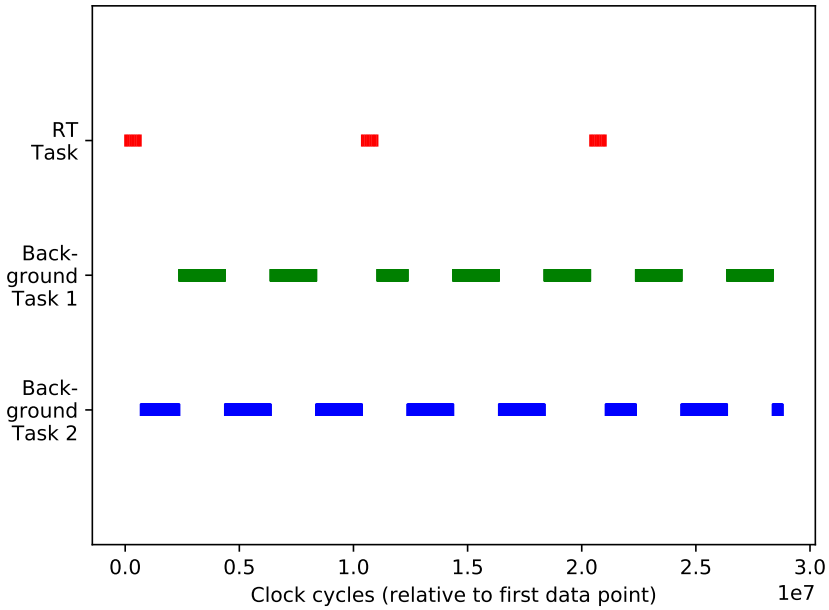

Figure B.2: cosf.pdf

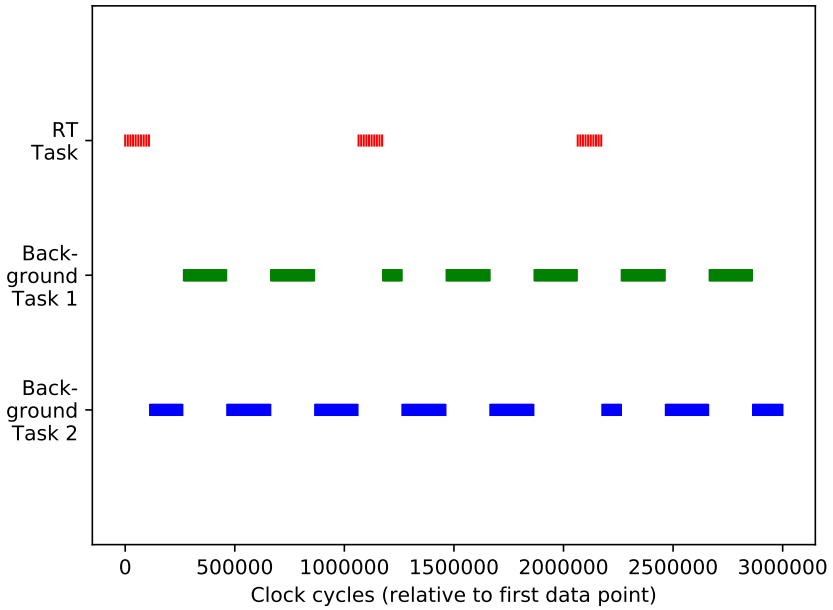

Figure B.3: rad2deg.pdf

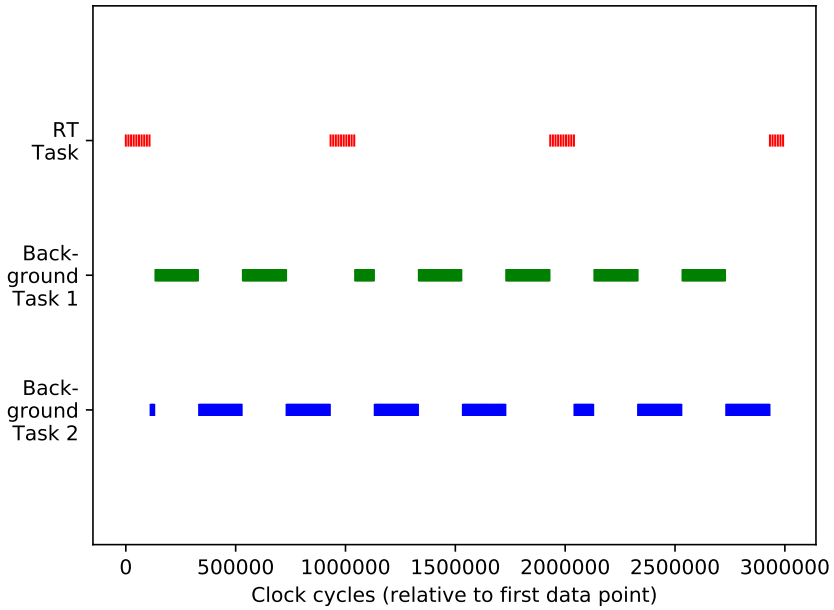

Figure B.4: deg2rad.pdf

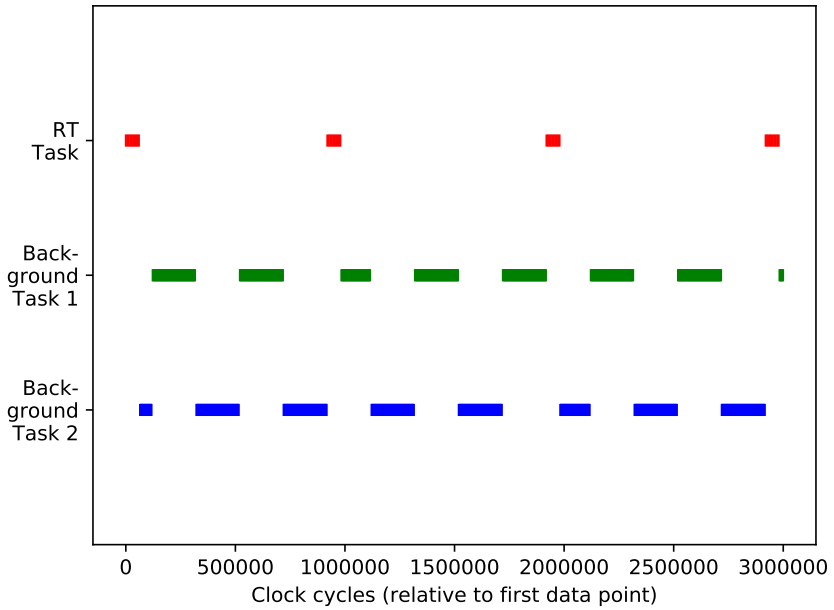

Figure B.5: ludcmp.pdf

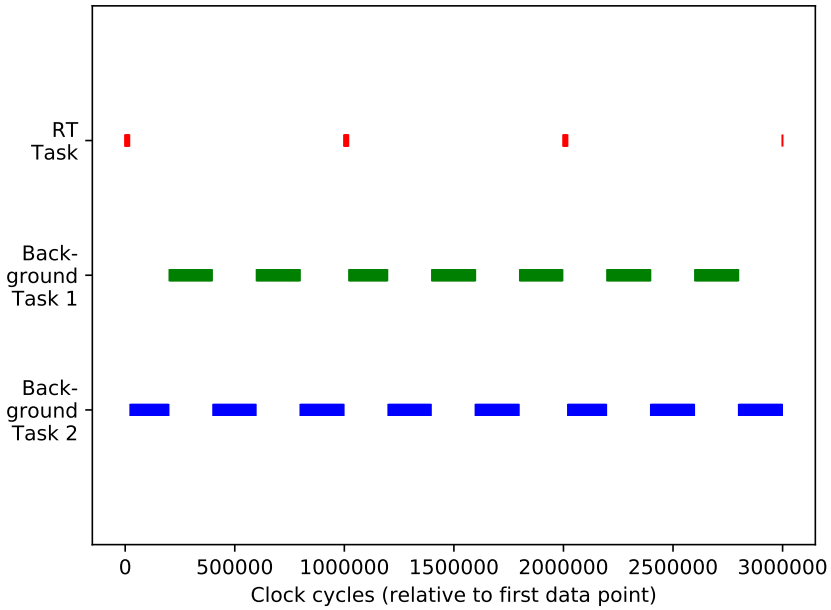

Figure B.6: binarysearch.pdf

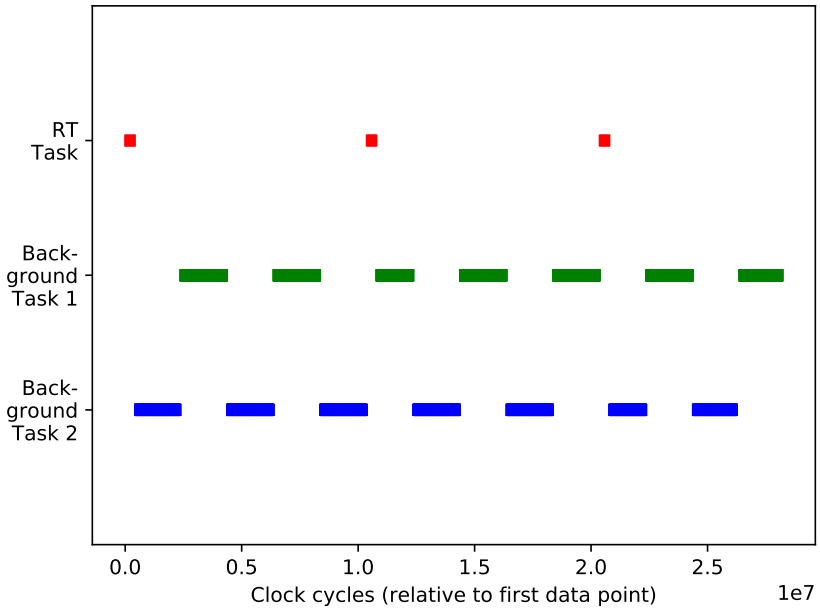

Figure B.7: countnegative.pdf

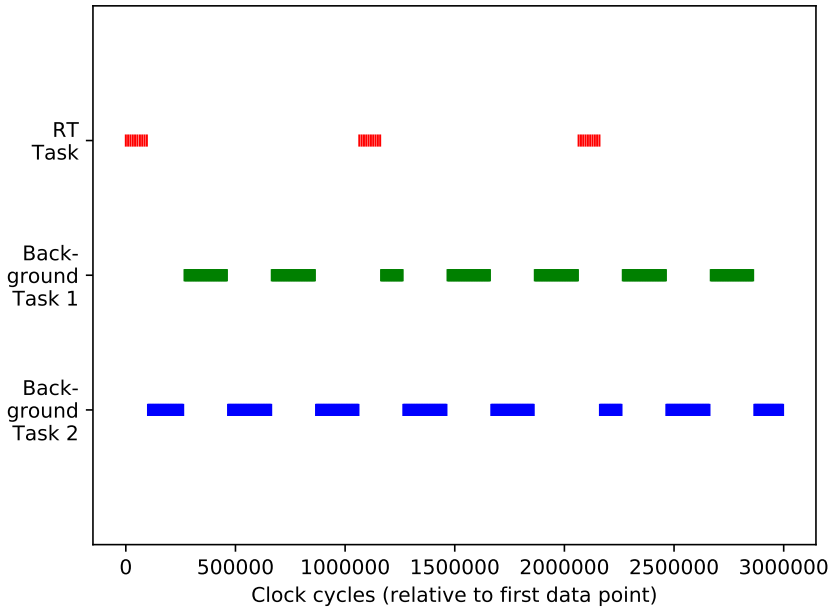

Figure B.8: recursion.pdf

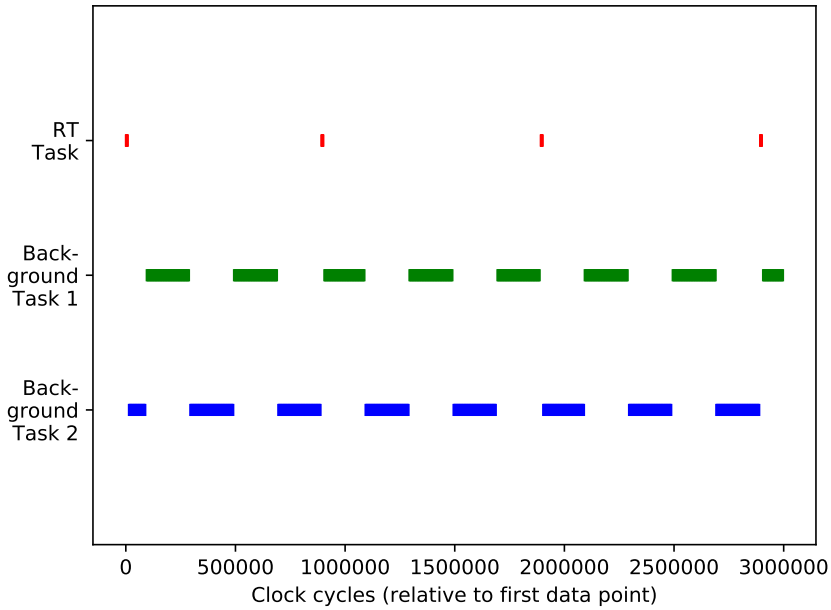

Figure B.9: prime.pdf

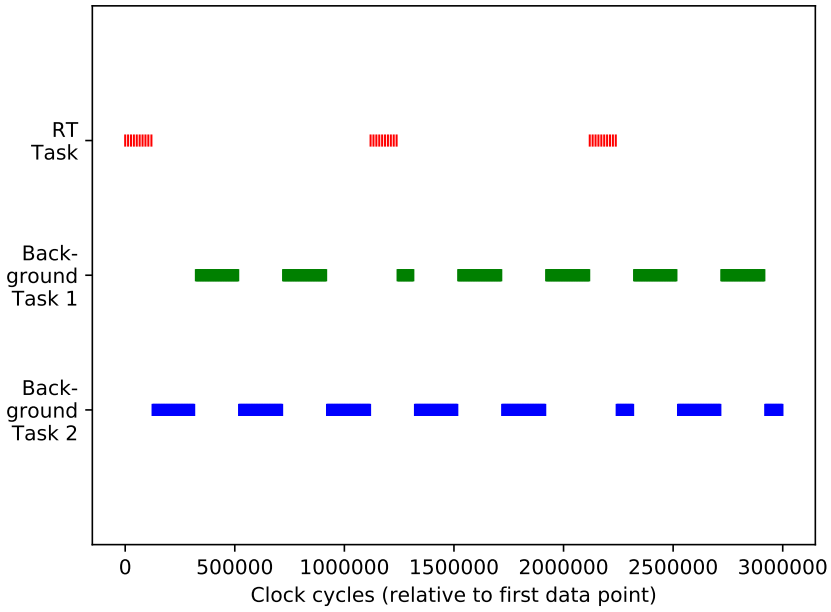

Figure B.10: jfdctint.pdf

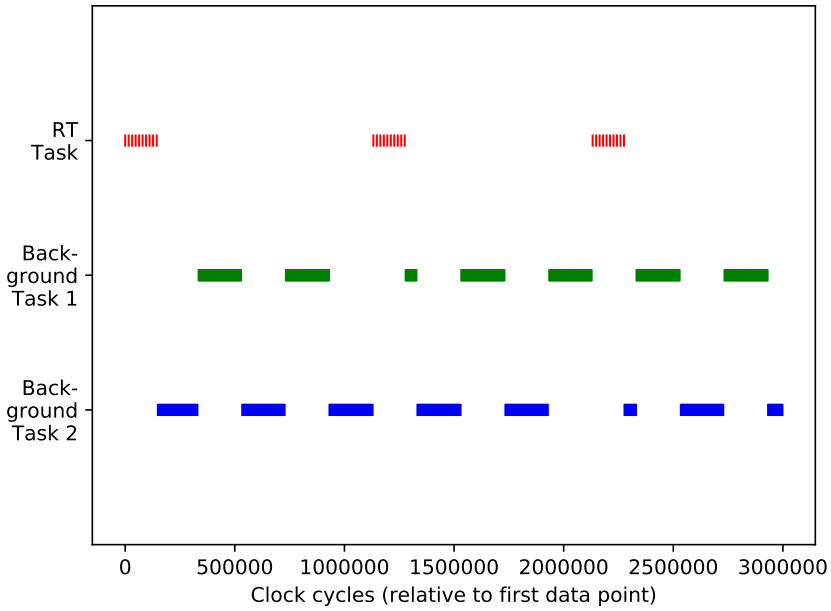

Figure B.11: fir2dim.pdf

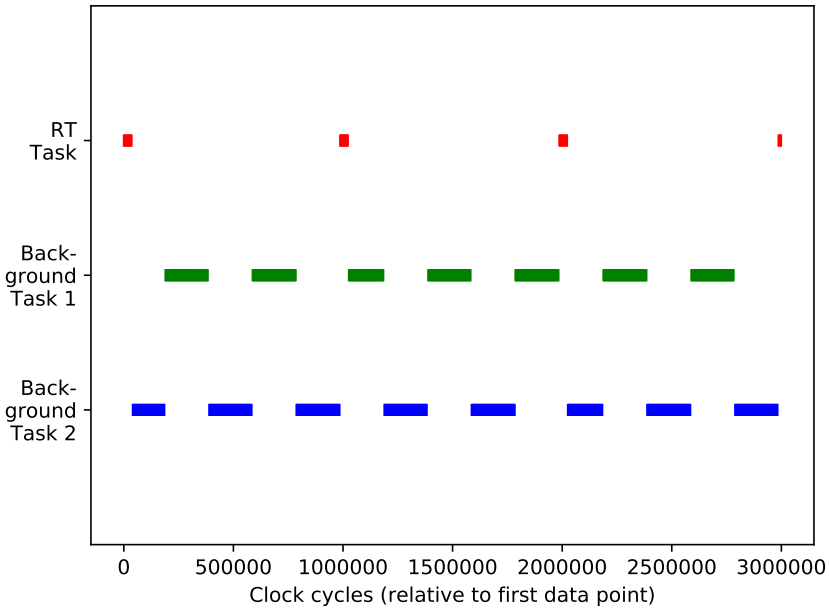

Figure B.12: complex_updates.pdf

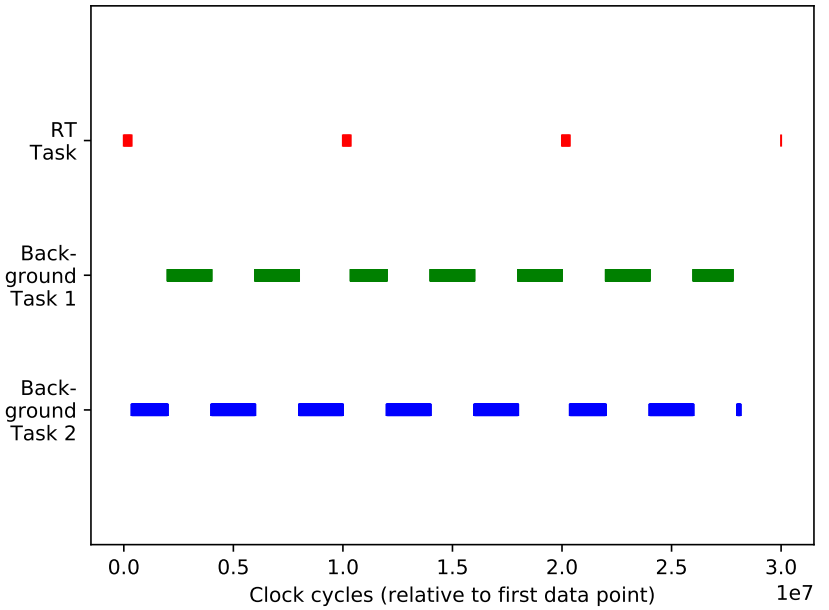

Figure B.13: bitonic.pdf

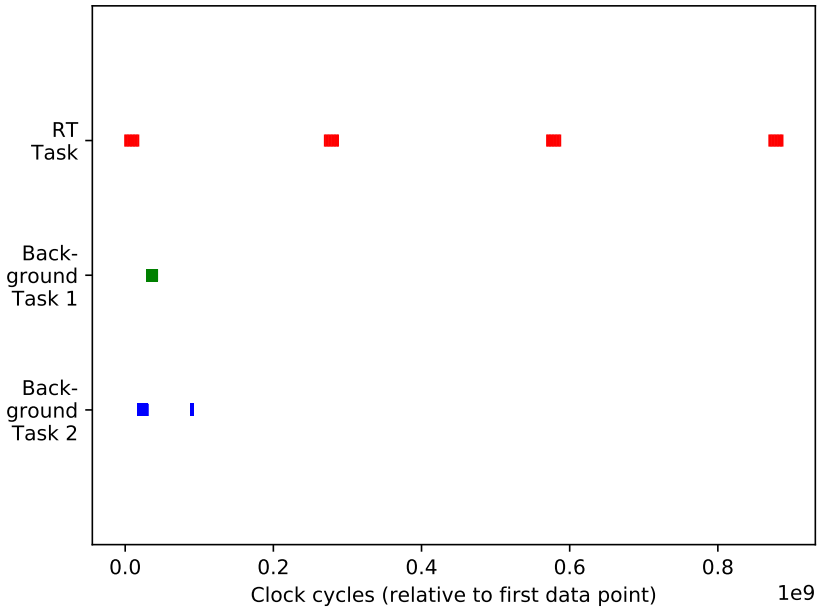

Figure B.14: isqrt.pdf

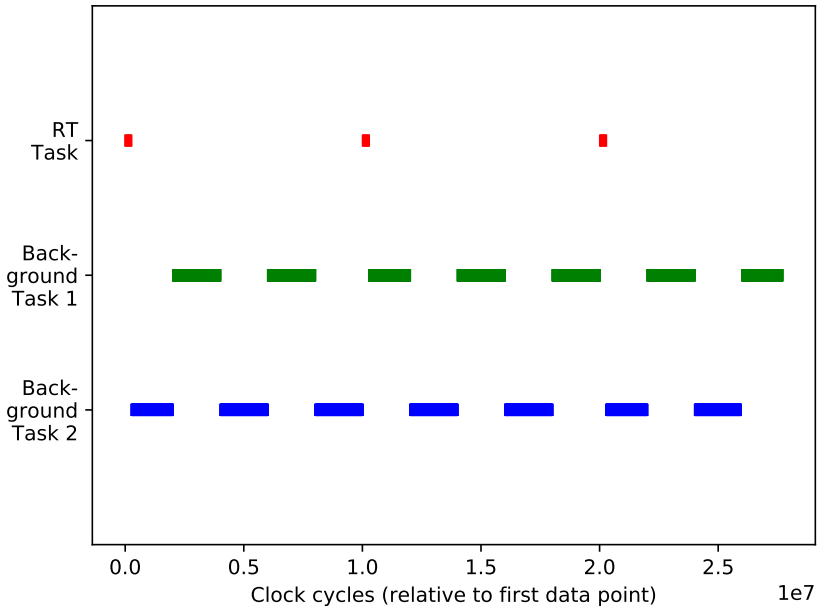

Figure B.15: matrix1.pdf

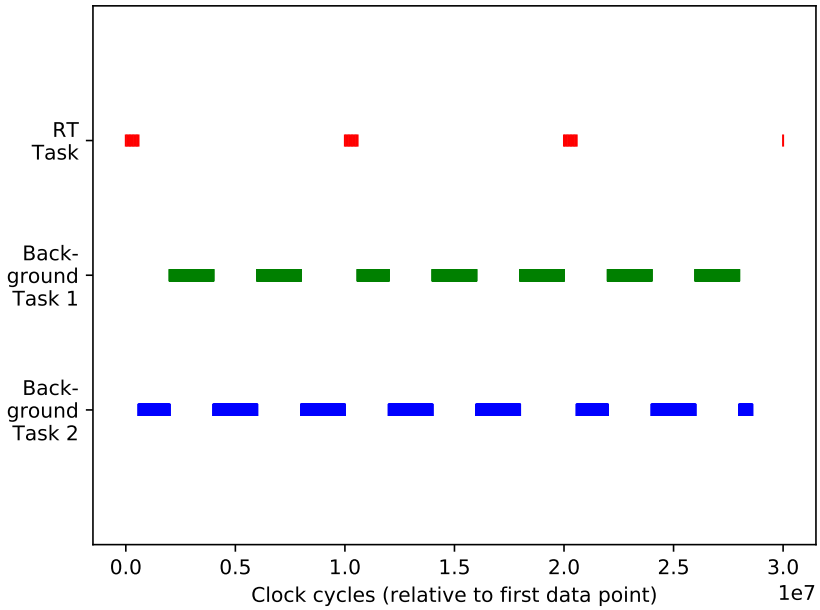

Figure B.16: bitcount.pdf

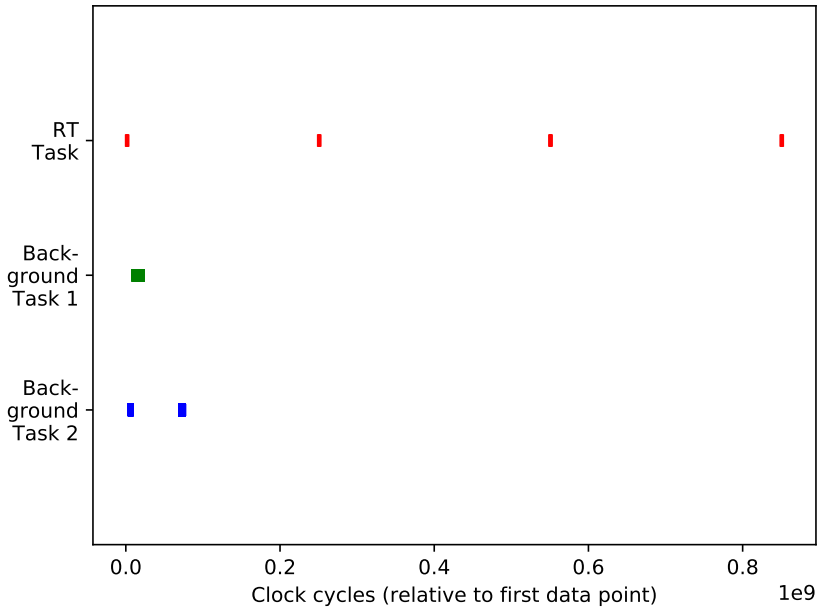

Figure B.17: st.pdf

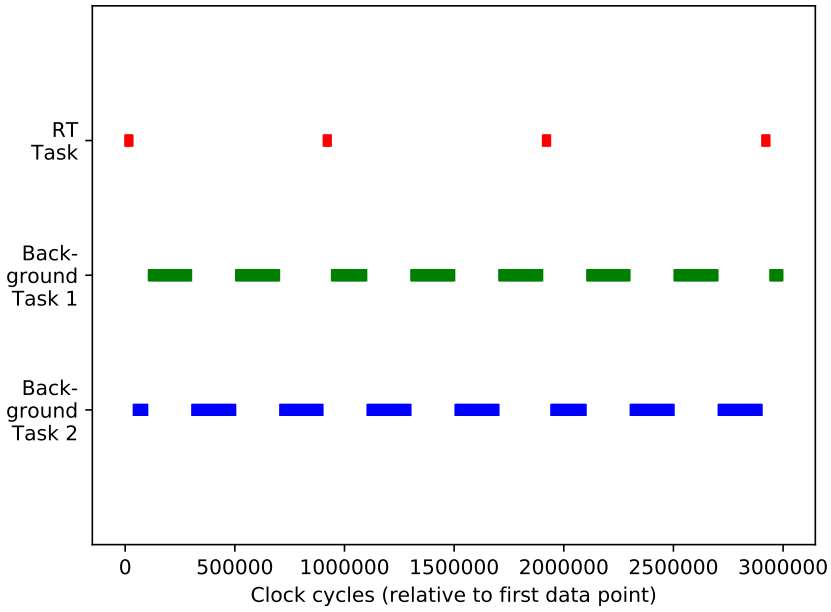

Figure B.18: insertsort.pdf

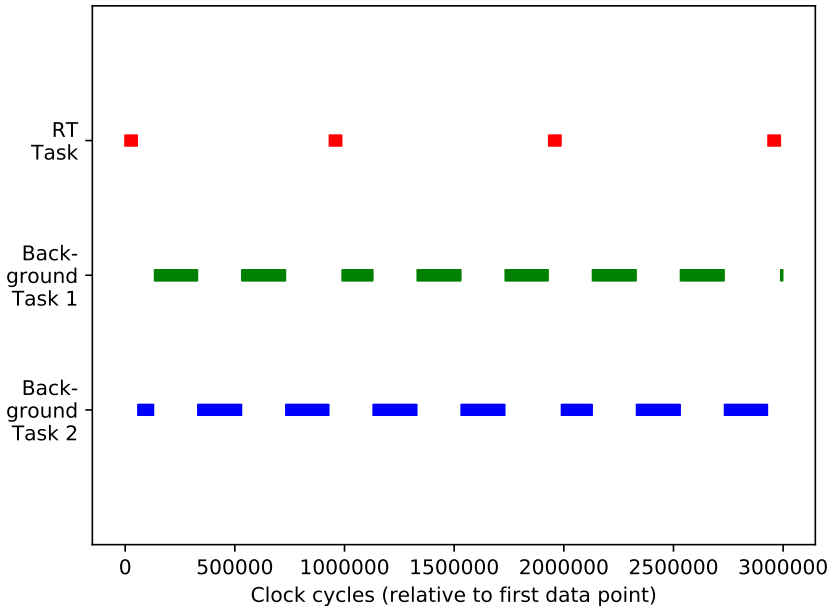

Figure B.19: iir.pdf

Appendix C

Code listings

Listing 3: Vivado tcl script to program the FPGA with a bitstream

```
set bitstream $::env(BITSTREAM)
open_hw
connect_hw_server
open_hw_target [lindex [get_hw_targets] 1]
set_property PROGRAM.FILE $bitstream [current_hw_device]
program_hw_device
```

Listing 4: Gdb script for loading, execution and result extraction

```
target remote localhost:3333
restore image.elf
set $a0 = 10
j _init
p $a0
p $a1
set $i = 1
while $i <= $a1
    p *((signed long *)(0x80100000) - $i)
    set $i = $i + 1
end
```

Listing 5: Measurement instrumentation asm code for initialization the Ariane as the target platform

```
        # entry point into the program
_init:
        # enable the floating point unit
        li t0, 0x2000
        csrrs zero, mstatus, t0
        # initialize the stack pointer
        li sp, 0x90000000
        # flush caches, by turning them off and on again
        csrwi 0x700, 0
        csrwi 0x701, 0
        csrwi 0x700, 1
        csrwi 0x701, 1

        # loop index in s1 runs from n-1 downto 0
        mv s1, a0
.L1:
        addi s1, s1, -1

        # capture the current value of the mcycle performance counter
        csrr s0, mcycle
        # call the main function of our benchmark
        call main
        # calculate the number of cycles elapsed during execution of main
        csrr t0, mcycle
        sub s0, t0, s0

        # store the result
        la t0, _results
        slli t1, s1, 3
        add t0, t0, t1
        sd s0, 0(t0)

        # end loop
        bgt s1, null, .L1

        # signal the debugger that execution has terminated
        ebreak
```

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis an der Carl von Ossietzky Universität Oldenburg und den DFG-Richtlinien festgelegt sind, befolgt habe. Des Weiteren habe ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste in Anspruch genommen.

_____

Mehrdad Poorhosseini