TU WIEN Informatics

Vorhersagbare und leistungsfähige Rechnerarchitekturen für zeitkritische Systeme

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Michael Platzer, BSc
Matrikelnummer 1029376

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Diese Dissertation haben begutachtet:

Isabelle Puaut	Michel Schellekens

Wien, 12. Oktober 2022

Michael Platzer

TU WIEN Informatics

Predictable and Performant Computer Architectures for Time-Critical Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Michael Platzer, BSc
Registration Number 1029376

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

The dissertation has been reviewed by:

Isabelle Puaut	Michel Schellekens

Vienna, 12th October, 2022

Michael Platzer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Michael Platzer, BSc

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

Wien, 12. Oktober 2022

Michael Platzer

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. Peter Puschner, who mentored me throughout almost my entire academic career, supervising my bachelor thesis, my master thesis, and now my doctoral thesis, who supported and encouraged all of the ideas and research interests I was pursuing, no matter how far-fetched they were.

I also want to thank Prof. Isabelle Puaut and Prof. Michel Schellekens for reviewing my dissertation, as well as Prof. Christoph Kirsch and Prof. Jens Knoop for serving on my proficiency evaluation committee.

Next, I would like to thank my former colleagues at the Cyber-Physical Systems group, mostly Denise Ratasich, Christian Hirsch, and Emad Maroun, for the many inspiring discussions and for making my time at TU Wien unforgettable.

Many thanks to Birgit Hofreiter and the entire team of the Innovation Incubation Center, particularly Alexandra Negoescu, for teaching me the importance of innovation and how to effectively communicate my ideas and projects.

Thank you also to Laura Flandorfer, Johanna Hover, and Lara Tiefenthaler, among others, for showing me the true meaning of ambition.

Most of all, I would like to thank my family, especially my mother Susanne, for their constant support and encouragement.

Kurzfassung

In Echtzeitsystemen müssen Rechenprozesse innerhalb einer bestimmten Zeitspanne abgeschlossen werden. Andernfalls könnte das System mit möglicherweise katastrophalen Konsequenzen versagen. Daher ist es wichtig, das Zeitverhalten dieser Prozesse zu analysieren, um zu gewährleisten, dass sie immer rechtzeitig abgeschlossen werden. Die Zeitanalyse moderner Systeme wird jedoch durch die mangelnde zeitliche Vorhersagbarkeit sowohl der Software als auch der Hardware eingeschränkt, bis zu dem Punkt, an dem eine Zeitanalyse überhaupt nicht mehr durchführbar ist. Die traditionell in Echtzeitsystemen verwendeten, besser vorhersagbaren Architekturen, sind nicht mehr in der Lage, die Leistungsanforderungen der heutigen anspruchsvollen Rechenaufgaben zu erfüllen. Diese Arbeit befasst sich mit diesem Problem, indem sie angelehnt an aktuelle Trends neue Rechnerarchitekturen für den Einsatz in zeitkritischen cyber-physischen Systemen untersucht, mit dem Ziel, die derzeitige Leistungslücke zwischen vorhersagbaren und leistungsstarken Rechenplattformen zu schließen. Es wird insbesondere ein Instruktionsfilter vorgestellt, der in bestehende Prozessorarchitekturen integriert werden kann und dadurch ermöglicht, zeitlich vorhersagbare Software auf einer Vielzahl von Architekturen auszuführen. Außerdem wird die Verwendung eines zeitlich vorhersagbaren Vektorprozessors in Echtzeitsystemen vorgeschlagen, ein Prozessortyp, der in der Lage ist, datenparallele Rechenaufgaben effizient auszuführen und eine höhere Vorhersagbarkeit ohne Leistungseinbußen bietet. Umfassende Evaluierungen und Vergleiche mit bestehenden Ansätzen zeigen, dass diese beiden Beiträge konkurrenzfähige Alternativen darstellen, die die Auswahl geeigneter Rechenplattformen und die Leistung von Echtzeitsystemen für relevante Rechenaufgaben verbessern.

Abstract

In real-time systems, computing tasks must complete within a certain time limit. Otherwise, the system might fail with potentially catastrophic consequences. Therefore, it is essential to analyze the timing behavior of these tasks in order to guarantee that they will always complete in time. However, the timing analysis of modern systems is complicated by the lack of temporal predictability of both the software and hardware, to the point where a timing analysis is not feasible at all. The more predictable architectures traditionally used in real-time systems are no longer capable of fulfilling the performance requirements of today's demanding workloads. This work addresses the problem by investigating new computer architectures, in line with current trends, for use in time-critical cyber-physical systems, with the aim of closing the current performance gap between predictable and high-performance platforms. In particular, it presents an instruction filter, which can be integrated into existing processor architectures with the aim of executing temporally predictable software on a wide variety of architectures. Also, it proposes the use of a timing-predictable vector processor in a real-time system, a processor type capable of efficiently executing data-parallel worloads and provides increased predictability without compromising performance. Comprehensive evaluations and comparisons with existing approaches show that both of these contributions are competitive alternatives that improve the choice of suitable platforms and the performance of hard real-time systems for relevant workloads.

Contents

CHAPTER 1

Introduction

Real-time computing is different from other types of computing in that the correctness of a task does not only depend on its logical correctness but also on the time within which it terminates. In real-time systems, tasks are usually associated with a deadline, which indicates the time by which the respective task must be completed. Real-time systems are frequently at the core of safety-critical cyber-physical systems, and missing a deadline can lead to catastrophic consequences.

Real-time systems are classified by the impact that a missed deadline can have on the system's operational status [72]:

- In a hard real-time system, a deadline miss leads to total system failure.
- A firm real-time system can tolerate infrequent deadline misses, but any results that are not available in time are useless.
- For soft real-time systems, a deadline miss degrades the quality of service, but results are still useful even if they are only available after the deadline.

A complex system might comprise several tasks with varying degrees of criticality, and a deadline miss could have very different consequences depending on the task. Such systems are referred to as mixed-criticality systems. In a mixed-criticality system, the above definitions apply to the individual tasks, rather than the entire system: a hard real-time task is one for which a deadline miss brings down the entire system, while a deadline miss by a soft real-time task does not impede the system's overall functionality.

In order to guarantee that a hard real-time task will not miss its deadline, its temporal behavior has to be analyzed and its maximum execution time must be determined. The analysis of the temporal behavior of real-time systems is called timing analysis. Timing analysis is closely related to functional verification in that both attempt to give certain

guarantees about the properties of a computer program. However, while functional verification is concerned with the correctness of the results produced by a program, timing analysis is interested in the time it takes an algorithm to terminate.

In the domain of functional verification, it is possible to divide the verification of a computer system into independent verification of the hardware and verification of the software. The Instruction Set Architecture (ISA) of a computer precisely specifies the operation of the underlying hardware and provides an abstraction layer to software running on this machine [68]. While hardware verification attempts to prove that the execution platform correctly implements the ISA [115], software verification builds on this foundation to demonstrate that software correctly uses the available machine instructions w.r.t. a given specification [70, 11]. However, the ISA only specifies the results that a machine instruction must produce, but does not define any timing properties. Therefore, timing analysis has to simultaneously consider the timing properties of a program as well as of the underlying hardware on which that program executes.

Static Timing Analysis (STA) attempts to derive a safe bound of the maximum time it takes a program to terminate, which is called the worst-case execution time (WCET). This requires solving two problems: modeling the timing behavior of the execution platform and determining the possible execution paths of a program [129]. While the first problem depends on the temporal predictability of the execution platform, the complexity of the second problem increases as the number of execution paths in a program grows.

The timing predictability of a computing system is a qualitative measure of the degree to which the execution time of a program can be accurately predicted. High timing predictability is essential to determine tight WCET bounds: a timing behavior that is hard to predict requires a pessimistic analysis, which typically leads to a gross over-estimation of the WCET. For instance, if the caching policy of a processor makes it hard to determine whether memory accesses will result in cache hits or cache misses, then the worst-case behavior must be assumed, which is a cache miss. This can lead to considerable over-approximations. Wilhelm et al. [130] concluded that the predictability of the processing platform is of such importance that it determines whether a timing analysis is feasible at all.

Besides the predictability of the hardware, the predictability of the software (i.e. the number of possible execution paths) also has consequences on the complexity of WCET analysis. Every data-dependent control-flow branch contributes to an exponential growth of the number of execution paths. Thus, exploring all possible paths becomes intractable. The situation becomes particularly complex for loops: if the loop condition depends on input data, then a special annotation that bounds the maximum number of loop iterations is required. Otherwise, it might be impossible to bound the execution time of a loop. A similar problem arises for recursive function calls, which equally require bounding.

Real-time systems are currently facing a compromise between performance and predictability: The low predictability of more performant systems leads to over-approximations

of the WCET. Hence, the performance can not be fully utilized in a time-critical application. Attempts to increase the temporal predictability of either hardware or software tend to impact performance. Yet, the performance requirements of real-time systems continue to grow with the emergence of new computationally intensive applications, such as advanced driver assistance systems and self-driving vehicles [69].

It seems that real-time systems are confined to inferior processors with weaker performance and simpler programs in order to retain the temporal predictability required for timing analysis, which prohibits their use in demanding applications. However, two paradigm shifts that are currently taking place have the potential to change this situation and allow real-time platforms to catch up. One of these paradigm shifts affects computing hardware, while the other one is taking place in the software domain.

For several decades computer architects have aimed at increasing the performance of processors, first by exploiting the combination of Moore's Law (the doubling of transistor densities every $18 - 24$ months) and Dennard Scaling (constant power densities due to the reduced power consumption of individual transistors) to increase the clock frequency, then via increased micro-architectural optimizations. However, as the race for better performance is slowed down by the increasing power requirements of modern processors, the attention shifts towards improving the energy-efficiency instead [29]. This new quest for reduced power consumption is reviving interest in architectures that are more amenable to timing analysis, such as hardware accelerators, which are inherently more predictable [85].

At the same time, a paradigm shift is taking place in the software world: the advent of machine learning is profoundly changing several domains of computer science, such as computer vision or natural language processing [2, 45]: Traditional algorithms are being replaced by neuronal networks, thus shifting software design towards more declarative programming. Neuronal networks are usually devoid of data-dependent control-flow branches. Instead, a fixed network topology determines the operations that need to be carried out [78]. As such, many programs based on machine-learning algorithms are inherently more temporally predictable.

This work explores new architectural concepts in the domain of real-time systems with an emphasis on improving the predictability of both hardware and software while taking inspiration from architectural concepts explored in other domains in an attempt to bridge the performance gap between time-critical processing and platforms without real-time constraints.

In particular, the aim of my research work is to answer the following questions:

1. Is it possible to improve the predictability of software by eliminating data-dependent control-flow branches without impacting its performance and without severely limiting the available processing platforms?
2. Which processing architecture is able to handle the emerging massively data-parallel workloads while retaining sufficient predictability to be usable in a hard real-time

system and also remaining flexible enough to execute a broad set of potential workloads?

The overall goal of my work is to advance the current state-of-the-art in real-time systems design by investigating architectures that allow to increase the temporal predictability of both software and hardware in time-critical systems and hence ease the timing analysis. For this purpose, architectural extensions to existing platforms as well as new processing architectures are considered and evaluated.

First, Chap. 2 gives an overview of the existing timing-predictable computer architectures. It also discusses the requirements of processor suitable for real-time systems. Further, the concept, the requirements, and the shortcomings of static WCET analysis as the currently most popular mean to achieve guaranteed execution times are explained.

Chap. 3 introduces an alternative approach to timing-predictability that attempts to forego some of the inconveniences of traditional WCET analysis by eliminating data-dependent control-flow branches from the program code. This approach is known as the single-path paradigm, since the execution traces of a program are all merged into a single path. In particular, the chapter shows that single-path code can be applied to the state estimation and control algorithm of a quadcopter, a highly dynamic real-time control system.

Single-path code makes use of predicated execution to replace data-dependent control-flow branches, which means that individual instructions can be enabled or disabled based on the value of a predicate. Therefore, single-path code can only be executed on a hardware platform that supports predicated execution. The demonstrated application uses the timing-predictable T-CREST architecture that has been designed with single-path code in mind and allows the predication of all instructions.

Besides some exceptions, such as the T-CREST platform and most notably the ARM ISA, predicated execution is not a common feature in most computer architectures. Therefore, Chap. 4 proposes a lightweight processor extension that adds support for fully predicated execution to existing processors. This so-called single-path filter is fitted to the hardware design of a LEON3 and an ARM Cortex-M0 processor and a detailed evaluation compares the performance of single-path code on these extended processors with that of regular code and the WCET bounds determined using traditional timing analysis.

The results of this evaluation show that single-path code is a competitive alternative to traditional WCET analysis. Although single-path code essentially forces the worst-case behavior for every execution, it facilitates the analysis to the point where determining the exact WCET becomes trivial. While single-path code cannot generally reduce the WCET of a program, it frequently performs better than the WCET bound found using traditional timing analysis. The reason for this seemingly counter-intuitive observation is that the WCET bound determined by STA is typically an over-estimation of the actual

WCET, which allows single-path to perform better despite always executing all program paths at once, including the worst-case path.

For instance, most processor architectures (even relatively simple ones designed for use in embedded systems) feature some form of branch prediction, which attempts to predict the outcome of a conditional branch instruction in order to speculatively start executing instructions at the predicted branch target address before the branch condition has been evaluated. The branch prediction is commonly based on the outcome of prior branch instructions. Keeping track of all possible control-flow branches that may lead to a certain point in the program's execution flow is intractable. Hence, a WCET analysis tool will have to conservatively assume that the branch predictor will mispredict the outcome of the branch instruction in all but the most basic scenarios and thus add the penalty incurred by flushing the pipeline to its WCET estimate. Single-path code, on the other hand, does not use any branch instructions. Therefore, branch mispredictions followed by stall cycles for flushing the pipeline cannot occur. As a result, for relatively short conditional statements the single-path code is faster than the conservative estimate of the WCET analysis. Fig. 3.2 in Chap. 3 illustrates in detail how single-path code can be faster for such a statement.

Chap. 5 extends the concept of predicated execution to parallel computer architectures by proposing a timing-predictable vector coprocessor. Vector processing is a form of parallel processing that is suited for data-parallel applications. A vector instruction processes a whole vector of elements rather than just a single value. Special vector masks are used as a form of predication, where vector instructions can be enabled or disabled individually for each element of a vector based on a vector mask.

The timing behavior and performance of the proposed vector coprocessor are analyzed in detail and compared to other vector processors and to existing parallel timing-predictable execution platforms. It turns out that the inherent efficiency of vector processing renders many of the optimizations used in modern architectures ineffective. Therefore, these optimizations can be removed without significant reductions in performance. Hence, removing optimizations that typically impede timing-predictability has much less of a performance penalty for a vector processor than for other processors. Therefore, the performance and scalability of the proposed timing-predictable vector coprocessor are similar to that of other vector processing architecture and significantly better than that of timing-predictable multi-core platforms for data-parallel workloads.

Chap. 6 concludes that leveraging predicated execution allows to improve the timing predictability of computer programs in time-critical systems. In particular, the parallelized predicated execution offered by vector processors appears to be well suited for demanding data-parallel real-time applications. As such, the proposed single-path filter, that dramatically improves the availability of predicated execution, and the proposed timing-predictable vector coprocessor are well positioned to advance the state-of-the-art in timing-predictable computer architectures.

CHAPTER 2

WCET Analysis and Timing-Predictable Computer Architectures

Hard real-time systems are dependable computer systems used in various safety-critical applications. The characteristic of a real-time system is that correct system behavior requires not only the results of computations to be correct but also that these results are produced within a certain time span [72]. Therefore, verifying the correctness of such a system involves WCET analysis to determine the maximum execution time of tasks [129]. This, in turn, requires modeling the timing behavior of the execution platform.

WCET analysis benefits from a processor architecture with easily predictable instruction timings. However, optimizations frequently used in modern processors, such as caches, branch prediction, and out-of-order execution, reduce the timing-predictability and complicate the timing analysis to the point where the timing-predictability of the architecture determines whether WCET analysis is feasible at all [130].

This chapter introduces the current state-of-the-art in timing-predictable computer architectures as well as in WCET analysis and discusses the associated challenges.

2.1 Timing-Predictable Computer Architectures

Early computer architectures typically had very predictable timing behavior that could be easily analyzed [122]. Shallow pipelines and flat memory hierarchies meant that the execution time of instructions had low variability, if not being constant. Therefore, timing models for these architectures were simple, and determining the WCET of a sequence of instructions was straightforward.

For several years the combination of Moore's Law (the doubling of transistor densities roughly every 18 – 24 months) and Dennard Scaling (constant power densities due to the reduced power consumption of individual transistors as a bi-product of reduced feature size) allowed processor performance to scale almost linearly with transistor count by increasing the operating frequency [29]. During this period, there was little incentive to complicate processor design by adding complex optimization features, which meant that the timing behavior of many processors remained relatively predictable. However, in the mid-2000s, power dissipation limits put an end to the acceleration of processor clock frequencies, and further performance gains required increased micro-architectural optimizations. While improving average-case performance, these speed-up mechanisms (e.g., caches, branch prediction, out-of-order execution) came at the expense of increased timing variability and reduced predictability.

Timing-predictability is usually not a concern for the designers of processor architectures. Instead, the main focus is to improve average performance [122]. However, this meant that new processors became increasingly less predictable and thus less suitable for use in time-critical applications.

Engblom and Jonsson [37] analyzed the timing behavior of single-issue in-order pipelines and identified so-called *long timing effects* (LTEs) that occur when one instruction affects the timing of other instructions. The presence and duration of these LTEs are used as a metric to classify processor pipelines and characterize their timing predictability.

Wilhelm et al. [130] described *timing anomalies*, i.e., variations in the timing behavior of a sequence of instructions resulting from interactions between the individual instructions as well as the micro-architectural state of the processor that are hard to predict. Timing anomalies jeopardize the timing-predictability of an architecture.

Thiele and Wilhelm [122] also identified threats to the timing-predictability of a system and proposed design principles to support better predictability. Their recommendations are, among others, to favor static rather than dynamic decisions, to use scratchpad memory as an alternative to cached memory, to take advantage of parallel execution instead of speculation in VLIW designs, or to use static scheduling in multi-core architectures.

Based on these findings, Berg et al. [15] established the following design principles for predictable processor architectures:

- Minimal variation in the timing of instructions.
- Non-interference between processor components.
- Deterministic processor behavior.
- Comprehensive documentation, including timing behavior of all instructions.

- Recoverability: Information about any processor state that affects instruction timing must be eventually recoverable (i.e., given an initial unknown state it must be possible to eventually gain full knowledge about the processor's state).

Based on these principles, a number of architectures for use in real-time systems were proposed that restored the predictability and analyzability of early processors while attempting to retain some of the speed gains achieved through micro-architectural optimizations.

Within the Merasa project, Ungerer et al. [123] developed a processor that runs several hardware threads on a single core, one of which is a hard real-time thread, while the other threads are available for tasks with less stringent timing requirements. While this improves the performance of the non-real-time threads, time-critical computations cannot take advantage of these gains since they are confined to the slower real-time thread. They followed up their research with the parMERASA multi-core architecture [124], which runs parallel hard real-time applications on up to 64 Merasa cores, organized in clusters and connected with a timing-analyzable network-on-chip (NoC). This architecture allows multiple time-critical tasks to run in parallel or to distribute the workload of a task across several cores. While increasing the number of cores significantly increases the overall performance, the real-time NoC quickly becomes the main bottleneck of the whole processing system and limits its scalability.

As part of the T-CREST project, Schoeberl et al. [113] presented the timing-predictable processor Patmos, which can be used in a multi-core configuration where individual Patmos cores are connected among each other via a message-passing NoC and to a real-time memory controller using a second NoC. A statically scheduled dual-issue pipeline potentially doubles the performance of each core, while using multiple cores allows parallelization similar to the parMERASA architecture. Schoeberl et al. demonstrated versions with up to 15 cores. However, they found that their NoC suffers from the same shortcomings as the one of the parMERASA platform. In particular, they report that the worst-case performance of parallel tasks only grows logarithmically in the number of processing cores, which drastically limits the scalability of the architecture.

Another category of fully timing-compositional processors is the precision-timed (PRET) machines [32], which feature predictable timing by completely eliminating any variability in the execution time of instructions. Liu et al. [77] proposed a PRET machine called Precision Timed ARM (PTARM), which interleaves four hardware threads in a five-stage pipeline such that the previous instruction of a thread exits the pipeline as the next one enters it, thus avoiding pipeline hazards. This greatly simplifies WCET analysis at the expense of slowing down the performance of each individual thread. As such, the PTARM is suitable for workloads that can utilize four or more independent threads. Zimmer et al. [136] followed up with a more flexible processor called FlexPRET, for which the number of pipeline stages simultaneously used by each thread is configurable on a per-thread basis. This allows having a hard real-time thread that utilizes only one pipeline stage at once, while another thread can use all the remaining pipeline stages for

less critical computation. However, while this improves performance for non-real-time tasks, time-critical tasks do not benefit.

Wilhelm et al. [130] suggested using processor architectures that are free of any timing anomalies, which they refer to as *fully timing-compositional architectures*. Timing anomalies usually result from branch prediction and speculative execution [104]. However, Hahn et al. [55] went on to show that even simple single-issue in-order cores are prone to timing anomalies. Because pipelined processors keep fetching instructions while the previous ones are still executing, the memory access of a load or store instruction can be delayed by the fetching of a subsequent instruction. This re-ordering of memory accesses potentially affects all processors with a von Neumann memory model (combined instruction and data memory) and more than just a few pipeline stages.

Hahn et al. followed up their findings by presenting the Strictly In-order Core (SIC) [52], a processor which delays memory accesses resulting from instruction cache misses until any memory accesses from preceding instructions are complete. They proved that SIC is free of timing anomalies, thus enabling compositional timing analysis. Their work has been followed up by proofs showing that the T-CREST and the PRET architectures are equally free of timing anomalies [64].

De Dinechin et al. [28] proposed using the Kalray MPPA®-256, a many-core processor with 256 cores, for timing-predictable massively parallel computation and argued that its processor cores are fully timing-compositional. However, Hahn et al. [52] conjecture that the buffers used in those cores are a likely source of timing anomalies. Hence, while this platform would offer parallel processing performance surpassing any current timing-predictable computer architecture, it appears to be unsuitable for hard real-time applications.

Timing-predictability is an essential property of a processing platform to allow its use in real-time systems. Since the optimizations used in most modern processors compromise this property, only a few processing architectures, most of which have been specially designed for that purpose, are available for time-critical applications. Yet, the choice of architectures is further limited by the need for a timing model and support by WCET analysis tools in order to be able to actually carry out a timing analysis for hard real-time tasks executing on a given processor. Moreover, even if tool support and a timing model are available, the analysis is not trivial.

2.2 WCET Analysis

In hard real-time systems, the WCET of tasks must be determined to guarantee the correctness of the system. This is frequently done via STA, i.e., using a static analysis tool to inspect the machine code of a program and calculate its maximum execution time based on a timing model of the execution platform.

The choice of processing architectures suitable for use in real-time systems is already limited by the stringent requirements regarding the timing-predictability of the archi-

tecture. However, in order to carry out a timing analysis, the selected processor must also be supported by any of the available timing analysis software, unless one desires to develop or extend such a software oneself. At the time of writing, the website of AbsInt GmbH, the company behind one of the major STA programs, lists less than 30 supported processor models for the latest release of their WCET analyzer [1] (and several of those that are supported appear to be close variants of the same architecture).

Even if a processing architecture is supported by timing analysis software, determining the WCET of programs remains a complex task. There is no general solution for bounding the execution time of any program, as that would implicitly solve the halting problem [129]. WCET analysis inspects the execution paths in a program and attempts to identify the worst-case path, which leads to the longest execution time. Unfortunately, the number of potential execution path grows exponentially as the number of conditional statements and other programming constructs introducing control-flow alternatives increases. Therefore, investigating every possible execution path and the various processor states through which it runs quickly becomes intractable. Most STA tools abstract the concrete state of a processor in an attempt to mitigate this problem. As a consequence, it is generally no not possible to determine the exact WCET of a program. Instead, the analysis gives a safe bound that is at least as high as the actual WCET. However, while that bound might be relatively accurate, it is often exaggerated.

2.2.1 Challenges in the application of WCET analysis

Although determining the WCET of real-time system tasks is essential, in practice, the process of performing WCET analysis of production code turns out to be a complex and cumbersome process requiring significant manual intervention [129].

The practical difficulties of STA of industrial code were investigated by Sehlberg et al. [114]. They did a static WCET analysis of production code used on construction vehicles and found that while it is relatively easy to obtain loose WCET bounds using analysis tools, significantly more effort is required if tighter WCET estimates are desired. In particular, code that happens to include many if-statements with exclusive conditions yields many infeasible paths, a fact that is difficult to capture in manual annotations and which leads to large over-estimates of the WCET.

These conclusions are supported by earlier work by Ermedahl et al. [38], who carried out three case studies where static WCET analysis was done on production code. They also concluded that good WCET bounds can be obtained by static analysis but that the analysis is labor intensive due to the extensive amounts of annotations required for tight WCET bounds.

Byhlin et al. [17] performed static WCET analysis on software in time-critical automobile communication networks. They conclude that the quality of the bounds of WCET analysis can be greatly improved by precise code and system knowledge, in particular about the range of possible input values. They criticize that the analysis tools have insufficient support for specifying input data ranges.

Wilhelm et al. [130] investigated which factors are enlarging the gap between WCET estimates and the effective upper bound of execution time. They found that the lack of predictability of the execution platform and the resulting uncertainties are the main factors leading to overestimated WCETs, to the point where the hardware architecture determines whether a WCET analysis is feasible at all.

Another factor that contributes to the overestimation is the exponential growth of the possible execution paths of a program as the number of data-dependent conditional control-flow instructions increases. As a result the exhaustive analysis of all paths becomes intractable even for moderately sized programs. Hence, abstractions are required, which must take into account the worst-case behavior of the abstracted component. Therefore, the structure and complexity of a program's control-flow graph contribute to overly pessimistic execution time bounds.

2.2.2 Alternatives to static WCET analysis

Because of the issues with STA, researchers have proposed alternative approaches to determine or at least estimate the WCET of a program. One of the main goals is to reduce the overestimation of the WCET that plagues the traditional static analysis.

Measurement-Based Probabilistic Timing Analysis (MBPTA) [109] uses a combination of measurements and static analysis to determine an execution time bound that holds with some probability. Optimistic WCET bounds would have a higher violation probability, while more pessimistic WCET bounds are more likely to hold.

Wenzel et al. [128] proposed a hybrid approach where timing measurements are used as a substitute to a hardware timing model and combined with STA. A similar approach is used by Bernat et al. [16], who compute probabilistic bounds on the WCET, again using a mix of measurement-based and analytical approaches.

Quinton et al. [102] introduce the concept of Typical Worst-Case Analysis (TWCA), which derives a formal bound for the number of violations of a typical worst-case execution time within a certain time frame.

While probabilistic approaches have proven to be useful in many cases, their application is not without challenges. In particular, hardware systemic effects need to be carefully considered [48], and appropriate test coverage is essential for the accuracy of the violation probabilities of the execution time bounds [74]. Additionally, the execution time of code must have a probabilistic behavior, which requires adequate support from hardware and software, such that the timing behavior can be modeled with probabilistic and statistical methods [84].

Therefore, probabilistic timing analysis has some similar issues as STA:

- While STA requires the processor to be timing-predictable, probabilistic alternatives require that it has a probabilistic timing behavior. In particular, the modeled

features must be statistically independent of each other, which is generally not the case in most processor architectures.

- STA can only be done for processors for which tool support and an accurate timing model are available. Similarly, probabilistic methods need to model probabilistic behavior and systemic effects of the execution platform.
- While the huge number of possible execution paths of a typical program forces the use of abstractions in STA, it requires careful analysis and planning of measurement runs to ensure a sufficiently high test coverage for measurement-based techniques.

Hence, probabilistic timing analysis also requires dedicated hardware, and obtaining WCET bounds remains complicated. Although probabilistic analysis may avoid the overestimation of the actual WCET that is typical for STA, the violation probability of the obtained bound may not be acceptable for tasks that require more reliable estimates.

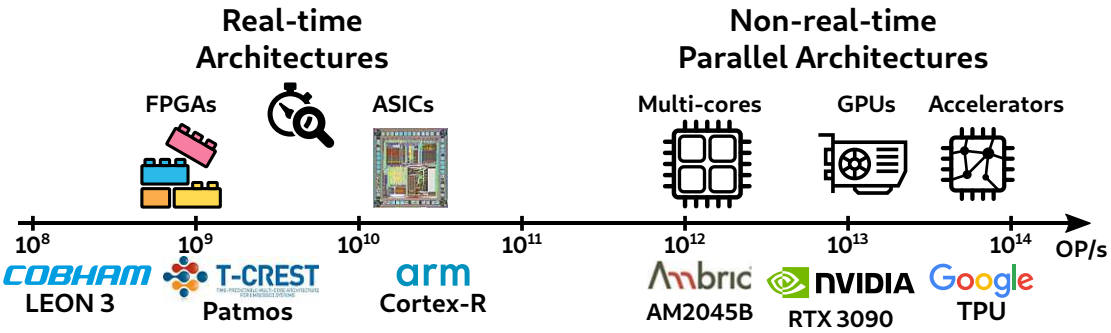
2.3 Considerations

Designers of hard real-time systems have only a very limited choice of execution platforms at their disposal. The processing system needs to be timing-predictable, and it must be supported by one of the available timing analysis programs. Even if that is the case, timing analysis remains a complicated process requiring extensive source code annotations to obtain tight WCET bounds. Measurement-based probabilistic methods that have been proposed as alternatives to STA are plagued by similar problems and are not suitable if a guaranteed execution time bound is required.

These rigid limitations have led to a significant performance gap between processing platforms that are appropriate for real-time systems thanks to their timing-predictability and the majority of computer architectures that do not consider the needs of time-critical applications, particularly when looking at parallel architectures, which are becoming increasingly important to tackle nowadays workloads [69]. Fig. 2.1 visualizes this gap by placing both real-time and non-real-time architectures on a performance scale based on the hardware's peak performance in operations per second. Note that the real-time architectures are further subdivided into soft-core processors synthesized in configurable logic and application-specific integrated circuits (ASICs). While a certain performance disparity between soft-core and ASIC processors is natural, the gap between the best-performing timing-predictable multi-cores (e.g., the 8-core ARM Cortex-R) and high-performance parallel computer architectures is still a couple of orders of magnitude.

Timing-predictability is an essential requirement for processing platforms in time-critical applications. Therefore, it is unlikely that the performance gap can be closed by attempts to shed or weaken the predictability of the execution platform. However, the restrictions imposed by current STA methods might be avoidable by continuing the search for alternatives. Freeing real-time systems from the limitations of current WCET analysis practice could simultaneously increase the choice of computer architectures

suitable for real-time systems, ease the process of obtaining execution time bounds for programs, and improve the accuracy of the resulting WCET estimates. This work attempts to do exactly that in the hope of identifying ways to reduce the performance gap by extending the choice of processors available for real-time computation and exploring the predictability of computer architectures that until now have not been considered for time-critical applications. The single-path filter proposed in Chap. 4 enables a wide range of processors to execute timing-predictable code, thus greatly increasing the choice of processors suitable for time-critical tasks. The vector coprocessor presented in Chap. 5 extends the concept of predicated execution to a parallel architecture and shows that vector processing is a performant mean to execute data-parallel real-time workloads.



Attribution: This figure contains a photograph of a Motorola 68HC908JB8, licensed under CC BY-SA 3.0 by Antoine Bercovici. The logos of Cobham Limited, the T-CREST project, Arm Limited, Ambric, Inc., NVIDIA Corporation, and Google LLC are trademarks owned by the respective companies or organisations, and are included here for identification purposes only.

Figure 2.1: Performance gap between processing architectures suitable for use in hard real-time systems and those optimizing for average performance, based on the theoretical peak performance in operations per second. The real-time architectures are subdivided into soft-cores implemented in configurable logic, such as the LEON 3 or the T-CREST platform, and ASICs, such as the ARM Cortex-R series. The performance numbers are for parallel variants (i.e., combined performance of the maximum number of available cores for multi-core architectures).

CHAPTER 3

A Timing-Predictable Real-Time Application

The content of this chapter is based on and extends a conference paper titled *A Real-Time Application with Fully Predictable Task Timing* [**?**].

A real-time control system remains stable by reacting to changes in the system's state within a certain time limit that depends on the dynamics of the system. However, the stability of the system does not only depend on the worst-case time within which the controller reacts but may also be compromised if the jitter (i.e., variability) of the response time is large. Traditional WCET analysis covers the first aspect by deriving an upper bound on the controller's response time but usually does not analyze the timing variability of an application. Additionally, static WCET analysis depends on the availability of tool support and a timing model for the execution platform. The determined bounds tend to grossly over-estimate the actual WCET unless significant manual intervention and annotations are employed.

This chapter serves to motivate an alternative approach that achieves fully predictable task timing by leveraging predicated execution to completely eliminate variability in the execution time of a task and to show the feasibility of that approach by applying it to a highly dynamic control problem. The chosen approach is known as the single-path paradigm, which eliminates any data-dependencies in the control-flow graph of an application by replacing conditional code sections with predicated execution (i.e., conditional instructions are always executed, but a predicate selects whether instructions actually change the state of the processor).

The use of single-path code allows to completely forego any static WCET analysis. Since the control-flow graph of single-path code has no data dependencies, the execution time is constant w.r.t. input data. Therefore, a single measurement is enough to determine that constant execution time, which simultaneously also corresponds to the WCET. Since

the execution time can be measured on the actual hardware, this approach also avoids any potential inaccuracies in the timing model of a processor that is used for traditional static WCET analysis. Further, since the execution time is constant, any variability in the controller's response time is completely eliminated, effectively removing the stability issues associated with response time jitter.

This chapter assesses the feasibility of a single-path control system for a highly dynamic system by compiling the control algorithm of a quadcopter to single-path code and evaluating its performance and stability. It also features a performance comparison with non-single-path code to quantify the performance penalty incurred by the predicated execution of all code branches.

3.1 Mitigating response time variability in real-time control systems

The stability of a real-time control system may be compromised if the controller fails to counteract deviations from the targeted system state within a certain time. Additionally, the system's stability could be threatened if the controller has a large variability in its response time, which introduces jitter into the control loop.

Ovaska et al. [89] proposed a predictive compensation scheme to mitigate the effects of response time variability on system stability. A predictor estimates the execution time of the control algorithm based on timing measurements during previous executions. The outputs of the controller are held back if the control task finishes early until the predicted time has elapsed. Response-time jitter is therefore reduced by smoothing.

A more drastic approach is to completely eliminate the variability in the response time by applying the control signals only after a delay equal to the WCET of the control algorithm. This approach was proposed by Henzinger et al. [56], who introduced the time-triggered language Giotto, which enforces strictly time-triggered task invocation as well as time-triggered application of control actions. This separation of reactivity from schedulability guarantees constant reaction times but requires a tight WCET bound. Otherwise, the delay becomes unnecessarily large, which might compromise stability more than the jitter.

Frehse et al. [42] further expanded on this by suggesting to reduce the delay of the actuator inputs to the upper bound determined by TWCA instead of the more conservative WCET bound. They show that this approach can improve the response of control systems which aim to optimize various parameters such as responsiveness, limited overshoot or stability, and can tolerate deadline misses.

Duggirala et al. [30] presented verification methods for real-time linear control systems that take into account the infrequent deadline misses resulting from TWCA. They reduce the analysis of linear control systems governed by linear ODEs to software verification with computation over reals and show that several methods from software verification can then be used to verify their correctness.

Using single-path code for the control algorithm effectively removes any variability in the response time. While this is similar to the approach by Henzinger et al., the need to determine the WCET of the control algorithm is avoided by single-path code, and there is no need to implement a mechanism to hold back the controller outputs since single-path code has constant execution time on timing-predictable hardware, which naturally guarantees that there is no variability in the controller's response time.

3.2 Single-Path Paradigm

The single-path paradigm eliminates execution time variability by eliminating any data-dependent control-flow branches, thus effectively merging all execution traces of a program into a single execution path [100]. STA is limited, among others, by the exponential growth of possible program execution paths as the number of control-flow alternatives increases, which renders the analysis of each possible path intractable. Single-path code eliminates this issue, requiring determining the execution time of a single execution path only. As a result, timing analysis becomes trivial. The execution time of single-path code is constant on a timing-predictable processor, requiring only a single measurement to determine this constant execution time and hence the *exact* WCET of the code [98]. As such single-path code follows the concept of repeatable timing [33].

In order to convert a program to single-path code, all data-dependent control-flow instructions need to be eliminated. This is achieved by using predicated execution as a replacement for conditional execution. Predicated execution allows enabling or disabling individual instructions based on the truth value of a predicate [26]. A disabled instruction is still executed (i.e., it is fetched by the processor just as any other instruction). However, it does not modify the state of the processor and thus has no effect. Therefore, when executing a single-path program, the same sequence of instructions is executed by the processor each time. Yet, whether these instructions take effect is controlled by predicates, which may vary from one execution to the next. The predicates capture the truth values of data-dependent conditions, thus replacing conditional execution governed by data-dependent control-flow changes.

Fig. 3.1 serves as an example of the single-path transformation, replacing a conditional statement by predicated execution. The pseudo-code on the left corresponds to the machine instructions emitted for a simple conditional statement with two mutually exclusive code paths (i.e., an *if-then-else*-statement). Based on the truth value of the condition `COND` (which is assumed to be the result of some boolean expression depending on input data), the control-flow either continues right away with the first assignment of the variable `x`, or is redirected to the *else*-branch, thus executing the second assignment. By contrast, the pseudo-code on the right illustrates how the same result is achieved using predicated execution. This time, the truth value of the condition `COND` is captured in a predicate, which enables either the first or the second assignment of `x`, with the other assignment being disabled. Both branches of the conditional statement are executed.

However, the predicate captures the value of the condition and governs which instructions actually take effect.

Substituting conditional execution with predicated execution is not exclusive to single-path code. Some ISAs, such as the 32-bit ARM ISA, support predicated execution as a means to improve performance. When supported, the intent is often that predicated execution is used for short conditional statements where the overhead of executing both branches is less than the performance loss resulting from frequent branch misprediction. Predicated execution is also sometimes used in cryptography, where constant execution times serve to defeat timing attacks that attempt to (partially) recover secret data by analyzing the execution time of a program [71].

However, in single-path code, predicated execution is not only used for simple conditional statements but also to remove any data-dependency in the control-flow graph. This also extends to function calls within conditional statements. A function that is called depending on a data-dependent condition is always executed in single-path code. However, if the associated condition is false and thus the function would not have been executed in regular code, then all instructions within that function are disabled by a predicate and thus have no effect. This also applies to recursive functions, which therefore require a recursion bound to avoid infinite recursion. STA equally requires a recursion bound to be able to determine a WCET bound for recursive functions.

The concept of avoiding any data-dependent control-flow changes also applies to loops in single-path code. Loops are always executed for the same constant number of iterations, which corresponds to the loop bound of a loop. If the loop condition becomes false before this maximum number of iterations is reached (which would correspond to exiting the loop in regular code), then a predicate disables the instruction of all subsequent iterations. Note that a loop bound is equally required for STA to determine the WCET of a loop. Single-path code always forces the worst-case (maximum) number of iterations of a loop.

```
  goto else if ¬COND                    eval COND
  x = 1                           ( COND) x = 1
  goto end
else:
  x = 2                           (¬COND) x = 2
end:
  ...                                     ...
```

Figure 3.1: A conditional statement as implemented in regular machine code using conditional control-flow instructions on the left and the equivalent single-path version which instead uses predicated execution on the right.

The main disadvantage of single-path code is that all branches of conditional statements must be executed. Therefore, the single-path version of a program is expected to execute slower than its regular version. The reduced performance is, however, traded for increased predictability. Also, the WCET, which is the most relevant metric for real-time systems, might not be increased all that much by the single-path transformation.

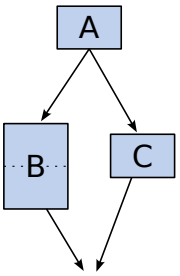
In regular code, the WCET of a conditional statement is the maximum of the execution times of the individual branches, while in single-path code, it is the sum. However, if the WCET of the individual branches is short or there is only one branch with a large execution time, then the difference can become negligible. For loops, single-path code always forces the maximum number of executions. Yet, that means the execution time of a loop in single-path code is not worse than its WCET for regular code.

The execution time overhead created by the single-path transformation is further diminished by the fact that the STA of regular machine code typically over-estimates the actual WCET. For single-path code, there is only one execution path to analyze, which renders the timing analysis trivial and allows to determine a much more accurate execution time bound [99]. As a result, the constant execution time of single-path code may even be lower than the WCET bound of the equivalent regular machine code [92].

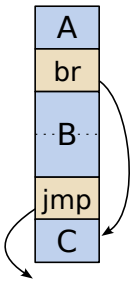
Fig. 3.2 illustrates how the single-path code of a simple conditional statement can execute in a shorter time than the WCET of the equivalent regular code, despite executing both alternatives. The control-flow graph of a conditional statement with two alternatives is depicted in (a), and (b) shows the sequence of regular program code with a conditional branch and an unconditional jump that a compiler would typically generate for such a statement. When executing such a conditional statement on a processor with dynamic branch prediction, then the best case behavior is that the branch is correctly predicted and the shorter alternative is executed, as illustrated in (c). However, the worst-case behavior shown in (d), which occurs if the branch is mispredicted and the longer alternative executed instead, takes significantly longer to execute as it also includes several cycles of delay due to the pipeline flush caused by the mispredicted branch. By contrast, the equivalent single-path code sequence presented in (e) requires no branches and has a constant execution time that is lower than the WCET of the regular code.

The example in Fig. 3.2 illustrates that single-path code may execute faster than the WCET of the equivalent regular program code using a simple example of a conditional statement with two very short alternatives. However, note that this may still be true for larger conditional statements as long as one of the two alternatives is shorter than the overhead caused by a mispredicted branch, since then the execution of the longer alternative after a mispredicted branch still takes longer than executing both alternatives in sequence, as done in single-path code. Chap. 4 contains a detailed comparison of the performance of single-path code w.r.t. the WCET bounds of regular code for a set of benchmark applications.

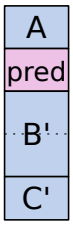
Besides rendering the timing analysis trivial, other advantages favor the use of single-path code in real-time systems. The execution time of single-path code is constant w.r.t.



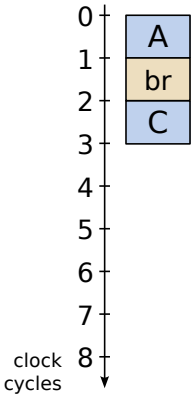
(a) Control flow graph of a simple conditional statement with two alternatives



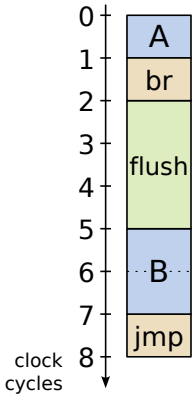
(b) Regular program code of that conditional statement which redirects the control flow using a conditional branch and a jump instruction



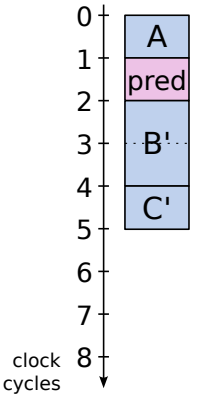
(c) Single-path code of that statement where blocks B' and C' are the predicated variants of B and C, respectively



(d) Execution sequence for regular code when the branch is correctly predicted as taken



(e) Execution sequence when the branch is *not* taken but was mispredicted to be taken



(f) Execution sequence for single-path code which uses predicated execution instead

Figure 3.2: Comparison of the execution time of various execution sequences of regular code and the execution of single-path code for a simple conditional statement. The conditional statement in this example has two alternatives, labeled B and C, which take two and one cycles to execute, respectively. We further assume that the conditional branch instruction takes one cycle to execute if it is correctly predicted, but causes an additional 3-cycle pipeline flush if mispredicted. An unconditional jump and the instruction that sets up a predicate for predicated execution are assumed to always execute in one cycle. Regular machine code can execute the statement in as little as 3 cycles if the shorter alternative is executed and the branch is correctly predicted. However, due to the penalty incurred by a mispredicted branch the WCET of the regular code is 8 cycles. Note that depending on the possible execution traces that lead to this conditional statement within a larger program this worst-case might actually not be reachable (e.g., because the branch predictor cannot be in a state that causes it to predict the branch as taken when in fact it is not). Yet, in general the WCET analysis has to conservatively assume that this local worst-case fully contributes to the global worst-case timing. By contrast, the single-path code always executes in 5 cycles and is thus faster despite executing both alternatives.

input data on timing-predictable hardware. This constant execution time (which also corresponds to the WCET) can be determined with a single measurement on the actual hardware. Therefore, there is no need for a timing model of the execution platform, which allows deploying single-path code on platforms that are not supported by analysis tools (e.g., due to the lack of a timing model). In addition, potential mismatches between the timing model and the actual hardware, which may lead to a violation of WCET bounds for regular code, are completely avoided.

3.2.1 Hardware Requirements

In practice, the adoption of single-path code is complicated by the limited availability of predicated execution in current processor architectures.

Single-path code relies on predicated execution to execute all the code branches of a program (merged into a single path) while discarding the results of instructions that would have been skipped by control-flow changes in regular machine code. Predicated execution allows to conditionally disable instructions (thus discarding their results) based on the truth value of a predicate.

Most processing architectures support a limited form of predicated execution covering some instructions. For instance, the *conditional move* instruction, which moves the content of one register to another register if a condition is true, is part of several ISAs [81]. The availability of a conditional move instruction is already sufficient to transform any WCET-analyzable program into single-path code [97, 101]. The resulting single-path code speculatively executes all conditional branches of a program and uses the *conditional move* to either keep or discard their results, based on the condition that would have triggered control-flow changes in regular code. The timing-predictable Java Optimized Processor (JOP) implements a *conditional move* instruction specifically to allow the execution of single-path code [112].

The presence of conditional variants of a small subset of the instructions of an ISA is referred to as *partially predicated execution*. While it is sufficient to enable the execution of single-path code, the efficiency of that single-path code is frequently compromised due to the need to avoid potential side-effects when executing the instructions of a conditional branch (or, more specifically, the need for these side effects to manifest only if the condition associated with the branch is actually true). For instance, a speculatively executed branch might need to avoid exceptions (e.g., a division by zero). Therefore, a single-path transformation that relies on a *conditional move* only, or a similar form of partially predicated execution, usually adds a significant amount of complexity to the code.

In order to execute single-path code *efficiently*, the processing hardware must support *fully predicated execution*, where all available instructions are predicated (except for control-flow changes, which by definition are always unconditional in single-path code). However, fully predicated execution is a rare feature. A notable example is the 32-bit ARM instruction set, which sets aside four bits in the encoding of every instruction for a

condition field that allows enabling or disabling the instruction based on the condition flags in the status register [63].

Among the timing-predictable processing architectures, the ISA designed as part of the T-CREST project supports fully predicated execution, and the compilation toolchain developed specially for it is capable of generating single-path code for execution on the timing-predictable processor Patmos [113]. As such, the T-CREST architecture is currently the only timing-predictable processor architecture with native support for the efficient execution of single-path code. Patmos was designed as a fully timing-predictable processor with a statically scheduled dual-issue RISC pipeline, a deterministic branch prediction mechanism, and predictable cache and memory access latency. Therefore, the execution of single-path code is effectively constant w.r.t. to input data on this architecture. Therefore, the T-CREST platform is used for the present evaluation.

3.3 System Description

This section details the control system used to motivate and evaluate the use of single-path code in a real-time control system. A quadcopter (i.e., a helicopter with four propellers rotating around the vertical axis) serves as the system to be controlled.

The T-CREST architecture with its timing-predictable processor Patmos is used as the execution platform for the controller. An Inertial Measurement Unit (IMU) serves as the only sensor of the system, measuring acceleration forces and rotational speeds, which allows estimating the *attitude* (i.e., the orientation within 3D space) of the quadcopter. Four motors, each spinning one of the propellers of the quadcopter, are the actuators.

Fig. 3.3 shows a schematic depicting the hardware setup. The measurements from the IMU are supplied to the processor, which executes algorithms for state estimation and control of the quadcopter. The control outputs are the desired speeds of the four propellers, which are then fed to the motor controllers, which in turn take care of regulating the power supplied to the motors such that the propellers turn at the desired rotational speeds. Patmos is a soft-core processor that is synthesized in configurable logic on an Field-Programmable Gate Array (FPGA).

3.3.1 Overview of Quadcopters

A quadcopter is a helicopter that generates lift with four horizontally spinning propellers. In contrast to conventional helicopters, which have a large main rotor in the center and control their attitude by adjusting the pitch of the propeller blades, the orientation of a quadcopter is controlled by regulating the rotational speed of its propellers [60, 18].

A quadcopter usually consists of the following major components [31]:

- Sensor hardware to detect the current state of the quadcopter, such as the current attitude, i.e., the orientation relative to the ground.

- A propulsion system consisting of four propellers, each of which is spun by a motor to which it is attached. The speed of each motor (and hence of the attached propeller) is regulated by an electronic speed controller (ESC).

- A flight controller, which reads the sensors and transmits the desired speed of each motor to the corresponding ESC.

These components are attached to a rigid frame that has four arms extending outwards in a cross pattern, with the four motors each mounted at the end of one arm. Fig. 3.4 shows the quadcopter in flight.

The thrust of each propeller is oriented downwards, thus counteracting gravity and thereby keeping the quadcopter aloft. Variations in the speed of the propellers at opposing ends of the frame create torque and allow it to control its attitude. By slightly tilting the frame and hence the thrust vector, the quadcopter is able to move sideways.

If the quadcopter deviates too far from a horizontal orientation, with the thrust of the propellers no longer aimed downwards and thus no longer fighting gravity, then the quadcopter will fall and crash.

Quadcopters are unstable and require active control [59]. In order to maintain stable flight, the quadcopter must react to changes in attitude by adjusting the propeller speeds such that they generate a torque that counteracts any deviations.

Stable flight is only possible if the controller counteracts attitude changes within a certain time span, the exact duration of which is determined by the dynamics of the quadcopter. As such, it is a real-time control system with a strict upper limit on the allowable response time of the controller.

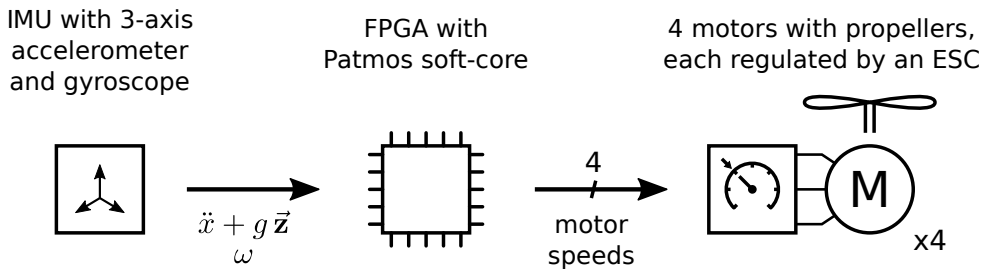


Figure 3.3: Schematic overview of the hardware setup. The IMU measures acceleration forces (which comprise the true acceleration $\ddot{x}$ as well as the force exerted by gravity $g\,\vec{z}$) and angular rotation rates ω. The Patmos processor, which is synthesized on an FPGA, uses these measurements to estimate its state and determines adequate speeds for each of the four propellers. These desired speeds are then communicated to motor controllers, which take care of regulating motor power accordingly.

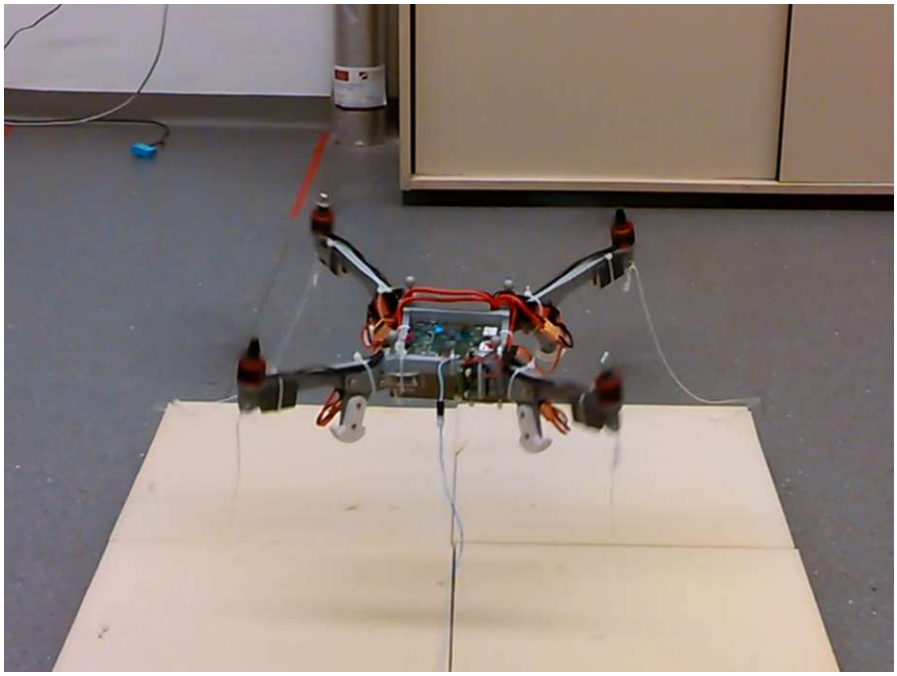


Figure 3.4: Photograph of the quadcopter in flight. It consists of a frame with four extending arms. A motor with a propeller is attached to each arm. The IMU and the FPGA on which the controller runs are located on a circuit board in the center of the quadcopter.

3.3.2 Real-Time Controller Software

The Patmos processor executes both a state estimation and a control algorithm. The former estimates the quadcopter's attitude based on the IMU measurements, and the latter computes a control response based on that attitude, which is then applied to the motor ESCs.

The IMU measures accelerations and rotational speeds at a constant rate, and whenever a new measurement is available, an interrupt is generated in the Patmos processor, and the new measurement data is transferred to the core. That interrupt prompts the execution of the state estimator and controller code, using the newly generated IMU measurements to update the current state estimate and then using that state estimate to compute an adequate control action. Upon completion, the control outputs generated by the controller are fed to the motor controllers, which subsequently adapt the propeller speeds accordingly.

Fig. 3.5 shows a timing diagram of a few controller cycles. The IMU updates trigger the execution of the state estimator and controller, which in turn computes and applies the control action. Since the execution time of the single-path code is constant, the response time of the controller, i.e., the delay from when the IMU acquires the measurements

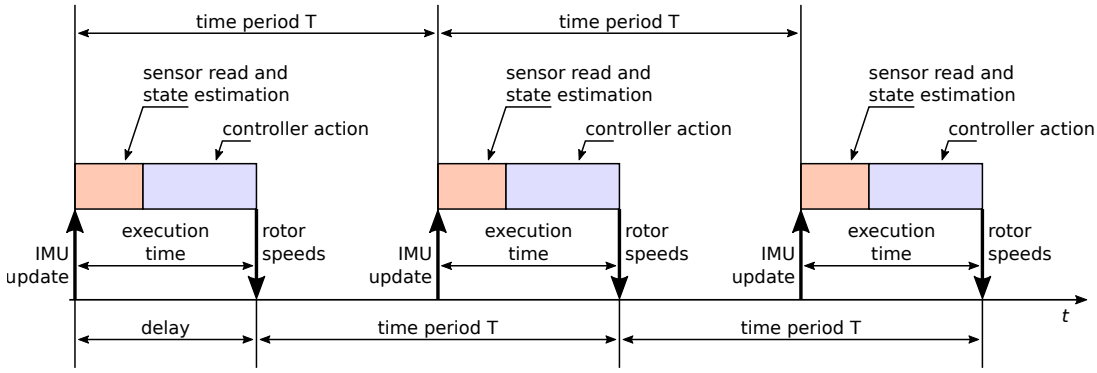


Figure 3.5: Timing diagram of the state estimation and control algorithms executing on the Patmos core. The execution time of these algorithm is a constant, therefore the delay between an update from the IMU and the adjustment of the rotor speeds by the controller (i.e. the response time of the controller) is also constant and the period between subsequent control actions equals the period between IMU updates.

until when the computed propeller speed are applied, is constant. Therefore, the period between subsequent adaptions of the propeller speeds is constant and equals the sampling period of the IMU.

In order to approximate the state of the quadcopter, the IMU orientation estimation algorithm proposed by Madgwick et al. [80] is used. The control algorithm is based on the nonlinear H-infinity controller with input coupling presented by Raffo et al. [103]. The choice of a nonlinear control algorithm rather than a linear PID controller for the present evaluation is motivated by the fact that nonlinear control algorithms are commonly used on quadcopters [137]. It serves to evaluate the single-path approach for a complex control algorithm.

The state estimation and the control algorithm have been implemented and compiled to single-path code using the compiler toolchain of the T-CREST project [96].

3.3.3 Timing Requirements

In order to determine the timing requirements of a real-world real-time control system, the limitations of the hardware must be taken into account. In particular, the rate at which the desired propeller speeds can be adjusted is limited by the motor controllers.

The propeller speeds of quadcopters are commonly regulated by ESCs, which are controllers for the motors which spin the propellers. Most commercially available ESCs are controlled via Pulse-Width Modulation (PWM). The input signal to the ESC is a pulse train with pulses of varying width but usually with a constant frequency. The pulse duration, i.e., the time between a rising edge of the signal and the next falling edge, encodes the desired speed. This is the reference input of the ESC, which will regulate the power delivered to the motor coils in order to maintain that speed.

The industry standard for regulating ESC is a PWM signal with a pulse width between 1 and 2 ms, where a 1 ms pulse corresponds to the minimum speed (i.e., standstill), and a 2 ms pulse is equivalent to the maximum speed [88]. The frequency of the PWM signal has no influence on the speed applied by the ESC. However, the range of acceptable pulse widths limits the available frequency range. In order for the ESC to correctly identify the pulses, they must not overlap, so in order to allow pulse widths of 2 ms, the period between subsequent pulses must be larger than that. Hence, the frequency of the PWM signal must be less than 500 Hz.

Some ESC controllers allow to calibrate the range of acceptable pulse widths, and given that a maximum pulse width of less than 2 ms has been programmed, it is possible to regulate the propeller speed with a PWM signal of more than 500 Hz. However, these appear to be non-standardized extensions to the default control signal. Therefore, a maximum update rate of the ESC of 500 Hz is selected and used as the task frequency for the control system.

The state estimator and controller are executed as one task, and since there is no other task to be executed, the only requirement for schedulability is that the period between two subsequent IMU interrupts must be greater than the execution time of the code. Therefore, it is required that the code of the state estimation and control algorithms must be executed within 2 ms in order to be schedulable on the execution hardware.

By measuring the execution time of the state estimation and control algorithm, the total execution time is 202.8 μs, which is well below the 2 ms limit. Since both algorithms are implemented in single-path code, this execution time is constant w.r.t. input data.

3.4 Evaluation

The practicability of applying the single-path paradigm to real-time control systems is evaluated by compiling the state estimation and control algorithms described in section 3.3 to single-path code and using it to stabilize a quadcopter in flight.

Since only the attitude (i.e., the angular orientation) is controlled, but not the position, the quadcopter tends to drift away. Therefore, the quadcopter is secured with tethers, which can be seen in Fig. 3.4, to limit the space within which it can move. Several flight tests showed that the single-path controller is indeed able to successfully control the orientation of the quadcopter and keep it in a stable flight, hovering above the ground.

The predictable timing behavior of the Patmos processor guarantees that single-path code executes in a constant time. Hence, there was no need to carry out an STA for the controller code since a single measurement is enough to determine its constant execution time and thus its WCET. However, the predictability of single-path code is traded for potentially reduced performance. The remainder of this section analyses the performance penalty incurred by the single-path transformation.

Table 3.1: Execution Time Measurements of the State Estimation Algorithm

Processor Architecture	CPU Cycles		Time (μs)	
	mean	max	mean	max
Patmos (80 MHz)	4612.5	4636	57.656	57.950
Patmos single-path	5087	5087	63.587	63.587
ARM (1400 MHz)	12605.9	87136	9.004	62.240

The single-path code for the state estimation and control algorithms has execution times of 63.3 μs and 139.2 μs, respectively. By contrast, the mean execution times for regular variants of these algorithms on the Patmos processor are 57.7 μs and 87.3 μs, respectively. Note that the execution time varies for the regular version, as opposed to the single-path version with its constant execution time. Yet, the maximum execution times that were observed are less than 1 μs above the mean value, suggesting that these algorithms naturally have a low timing variability. Indeed, the source code contains only very few short data-dependent code branches.

The overhead introduced by the single-path transformation is larger for the control than for the state estimation algorithm. While the constant execution time of the single-path variant of the state estimation algorithm is about 10 % larger than the mean execution time of the regular variant, the single-path variant of the control algorithm takes almost 60 % longer to execute than the regular variant takes on average.

Patmos is a timing-predictable processor, and the control algorithms for the quadcopter show little timing variability even if they are not compiled to single-path code. In order to further motivate the appeal of constant execution times, both algorithms are also executed on an ARM Cortex-A53 processor and their execution times during 10 000 executions are recorded. Tables 3.1 and 3.2 list the mean and maximum execution times of the state estimation and the control algorithm, respectively, for all code variants and processor architectures discussed so far. While on Patmos the algorithms were running on the unmanaged bare-bones core, on the ARM they were executed as processes managed by an Operating System (OS). In both cases performance counters were used to gather the execution times and on the ARM all measurement that were interrupted by the OS (e.g., by context switches or signals) were discarded to ensure a fair comparison. The measurements on both platforms include execution times with the cache being empty (or at least, not containing any of the instructions or data of the algorithms) as well as with a warmed-up cache. Note that the timing-predictable method and data caches of the Patmos core are much more predictable than the caches of the ARM. Additionally, the single-path code always forces an initial cache fill to ensure a constant execution time.

Executing the state estimation and the control algorithms on the superscalar ARM processor is significantly faster on average than on Patmos. However, the timing variability

Table 3.2: Execution Time Measurements of the Control Algorithm

Processor Architecture	CPU Cycles		Time (μs)	
	mean	max	mean	max
Patmos (80 MHz)	6986	7026	87.326	87.825
Patmos single-path	11136	11136	139.200	139.200
ARM (1400 MHz)	16876.7	201177	12.055	143.698

is also much larger for both algorithms. The box plot shown in Fig. 3.6 visualizes the differences in timing behavior between these architectures. While the average performance of the ARM core is much better, there are several outliers with a much larger execution time, with some executions taking up to 10 times longer than the average. These outliers are most likely caused by cache misses which force the core to fetch instructions and/or data from varying cache levels or even from main memory. Despite the lower clock speed of the Patmos soft-core, some of the execution times recorded on the ARM core even exceed the constant execution time of the single-path variants on Patmos.

The comparison shows that the execution time of single-path code is larger than that of regular code on the same architecture. Additionally, a timing-predictable computer architecture is expected to have a worse average-case performance than an architecture optimized for maximum throughput, since speed-up mechanisms affecting the predictability of its timing behavior must be avoided. However, when it comes to predictability and worst-case performance, then the timing-predictable Patmos soft-core seems able to compete with these high-performance architectures. Single-path code takes timing predictability to the next level, completely eliminating any timing variability on a predictable execution platform such as the T-CREST architecture.

3.5 Findings

Single-path code reduces timing-variability and renders timing analysis trivial by removing all data-dependent control-flow changes. Instead, predicated execution is used for conditional code. Despite the decreased performance, single-path code might be an attractive choice for hard real-time systems where predictability, timing guarantees, and low response-time variability are essential for overall system performance and safety. As such, single-path code is a potential alternative to traditional WCET analysis since it avoids many of the troubles that typically complicate STA.

On hardware that can execute single-path code with constant execution time, there is no need for static analysis at all. Instead, the constant execution time (which thus also corresponds to the WCET) can be determined with a single measurement. Hence,

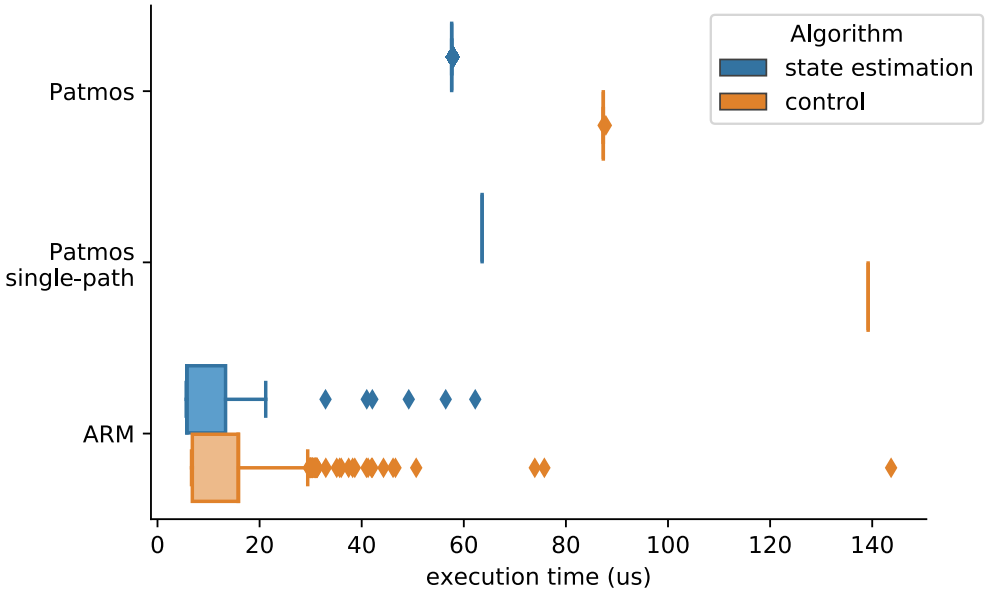


Figure 3.6: Box-plot showing the distribution of the execution time measurements of the implementation of the state estimation and the control algorithms on the Patmos processor (with and without using single-path code) running at 80 MHz and on a 1.4 GHz ARM processor. For the time-predictable Patmos core the variability is very low (or 0 for single-path code), thus the box is reduced to a line. However, the measurements on the superscalar ARM processor show significant variability, with some outliers in the order of 10 times larger than the average.

the need for a static analysis is completely avoided, which means that no timing model or tool support is required.

While single-path code with constant execution times does not require a static analysis, tool support is required for the single-path transformation itself. The evaluation presented in this chapter used the compilation toolchain of the T-CREST project [96] to generate single-path code for the Patmos core. Yet, transforming regular code to single-path code depends on the ISA but not on the specific processor used. Therefore, adding support for a new ISA to the existing single-path transformation tools potentially enables the use of single-path code on all processors complying with that ISA. By contrast, the timing models required for traditional STA are processor-specific. Hence, static timing analysis software requires changes for every new processing platform that shall be supported. As such, tool support for single-path code is likely much simpler to extend to a broad range of architectures than static WCET analysis.

Yet, one of the main practical limitations for the use of single-path code in hard real-time systems is the limited support for predicated execution on existing computer architectures, which is essential for the execution of single-path code. Therefore, in order

to allow more widespread use of single-path code, it would be desirable if support for fully predicated execution could be extended to more processing architectures.

CHAPTER 4

Making COTS Processors Timing-Predictable

The content of this chapter is based on and extends a conference paper titled *A processor extension for time-predictable code execution* [92].

Single-path code is a code generation paradigm that renders execution time analysis trivial by eliminating any data-dependent control-flow branches. Instead of redirecting the control flow, all conditional code branches make use of predicated execution to conditionally enable or disable instructions based on the truth value of predicates. Thereby, the execution traces of a program are all merged into a single path, and the same sequence of instructions is executed for every invocation of the program, although some instructions might have no effects as they are disabled depending on the current value of the associated predicate.

In regular code, the number of possible execution paths grows exponentially in the number of control-flow alternatives, which complicates timing analysis. By reducing the program to a single execution path, single-path code drastically simplifies timing analysis.

In order to be able to execute single-path code, the execution platform must support predicated execution. In particular, as discussed in Chap. 3, the *efficient* execution of single-path code requires fully predicated execution, where every instruction can be predicated. However, fully predicated execution is an uncommon feature in nowadays processor architectures.

The control system presented in Chap. 3 executes on the timing-predictable Patmos processor developed as part of the T-CREST project. This architecture has been designed with support for single-path code in mind. Yet, most processing architectures are not specially designed to be able to execute single-path code, and the effort and cost of developing a custom architecture suited for a certain purpose and with support for single-path code are significant. More widespread adoption of single-path code is hindered by

the limited support in existing architectures and the huge amount of work required to develop and maintain a custom architecture.

This chapter evaluates the feasibility of a single-path extension, a lightweight processor extension intended to add support for single-path code to any existing processor architecture while requiring only minimal changes to the processor design.

4.1 Single-Path Filter

The goal of the present work is to execute single-path code on existing processor cores that do not have native support for fully predicated execution. For that purpose, both the instruction set and the hardware design of a processor are extended.

Special instructions for manipulating predicates are added to the ISA of a target architecture. These new instructions are encoded with unused opcodes. Existing cores do not understand these special instructions. Therefore, an instruction filter that interprets the special instructions is placed on the instruction fetch path of the core. Regular instructions are then filtered based on the value of predicates. These predicates are hosted by the filter itself and modified by the new instructions.

The automated single-path transformation algorithm developed by Prokesch et al. [96] is used to convert regular machine code to single-path code. While Prokesch et al. implemented single-path generation inside their port of the LLVM compiler, this transformation is applied as a post-processing step to a fully compiled and linked executable. That way the single-path conversion is not tied to a specific compilation toolchain. The transformation rearranges the basic blocks of the Control-Flow Graph (CFG) of a program and replaces conditional control-flow instructions with special instructions that modify predicates.

Allowing existing processor cores to execute single-path code, therefore, involves two steps:

1. Single-path code is generated from regular machine code and consists of restructured object code that includes special instructions for computing predicates. These special instructions extend the ISA of a processor architecture and are encoded with unused opcodes.

2. An instruction filter is added to the processor core. At runtime, this filter interprets the special predicate-defining instructions of the single-path code and filters regular instructions depending on the predicate states. As a result, the processor receives a stream of filtered native instructions (either instructions from the object code or NOPs) at runtime.

Fig. 4.1 shows a conceptual diagram of a processing platform using the single-path filter. All instructions that are fetched by the core pass through this filter. The filter

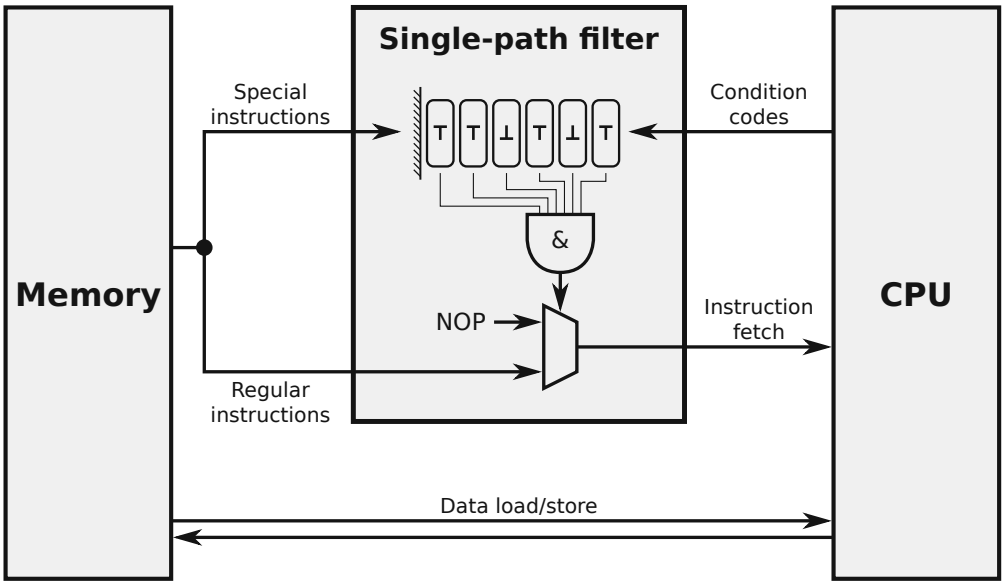


Figure 4.1: Concept diagram of the single-path filter: Instructions are fetched from memory and pass through the filter, from where they are either passed on to the core or replaced by an instruction with no effects. Special instructions are used to control the predicates. The filter has access to the condition codes of the core, thus allowing to set predicates conditionally.

directly interprets the special predicate-defining instructions that were injected into the code by the single-path transformation algorithm and implements predication for all other instructions. Regular instructions are either passed on to the processor or replaced with NOP instructions (depending on the architecture, there might be several instructions that have no effect, but for simplicity, all of them are referred to as NOPs). Instructions are only forwarded to the core if all predicates on the predicate stack are true. Otherwise, they are replaced by NOPs.

Conditionally modifying predicates requires access to the condition codes of the processor. Therefore, the filter has an interface that allows to route the condition codes out of the core and into the filter. The filter is then capable of evaluating these condition codes and modifying predicates accordingly.

4.1.1 Required modifications to support single-path code

The single-path conversion is applied to the executable file of a program after all compilation and linking steps have been completed, which has the advantage that it is not dependent on a specific compilation toolchain. That requires, however, that any additional state information necessary for the execution of the single-path code (such as, for instance, the predicate values) need to be saved in the filter, as saving it in memory or

registers might lead to collisions with the memory or register allocation of the preceding compilation or linking steps.

The first requirement to execute single-path code generated in this manner is that the execution platform must support fully predicated execution. The single-path filter must interpret special instructions that compute predicates, manage the predicates and filter out instructions that are disabled by these predicates. Predicates capture the truth values of conditions, and a new predicate is required for every condition that is encountered. Predicates expire when the execution of subsequent instructions no longer depends on the associated condition. Programming constructs that use conditions, such as conditional statements or loops, can be nested, with new conditions applying on top of others. Consequently, the predicates should be managed in a predicate stack. A new predicate is pushed to the stack when encountering a condition, and the predicate is removed from the stack when it expires. That predicate stack must be stored in dedicated hardware in the filter such that the predicate values are readily available to it.

Another requirement is that loops require an iteration counter. In regular code, the number of iterations of a loop depends on the loop condition only. However, in single-path code the loop bound dictates the number of iterations and a counter is required to count these iterations. This counter cannot be stored in memory or a register either, in order to avoid restricting the hardware resources available to the compiler. Therefore, the iteration counters for loops in single-path code also need to be stored in hardware. Loops might be nested. Hence, a loop counter stack is required. A new loop counter is pushed to that stack upon entering a loop and initialized with the total iteration count. The counter is then decremented on every iteration. When the loop counter reaches 0, the loop exits, and the loop counter is removed from the stack.

The single-path filter must also have a dedicated return address stack for single-path functions. In regular code, a function call writes the address of the call instruction to a specific register known as the *return address register*. When the function returns, it transfers control back to that address. Function calls can be conditional, for instance, when they appear inside conditional statements. In single-path code, every function call is executed unconditionally, but depending on the values of predicates, all instructions of that function might be inactive, and thus the function call might have no effects. This is equivalent to a function that would not have been executed in regular code. Since an inactive function does not modify any memory locations or registers, including the return address register, the return address would be lost if it were not saved elsewhere. Therefore, the return address of single-path function calls must be stored in the filter as well. Function calls are usually nested. Hence, a return address stack is required.

Finally, in order to support recursive function calls the single-path filter also requires recursion counters. Since in single-path code function calls are always unconditional, a recursive function would call itself over and over again indefinitely if the recursion depth is not limited otherwise. Therefore single-path code requires a recursion bound for every recursive function. That bound is compared against a recursion counter every time a recursive function is called. The recursion counter is incremented every time the function

is entered and decremented every time the function is left. A call of the recursive function is aborted if the recursion counter reaches the recursion bound.

4.2 Filter Implementation

In order to add the ability to execute single-path code to an existing processor, an instruction filter with a predicate stack is added to it, which computes and saves predicates triggered by special predicate-defining instructions and filters regular instructions based on the values of these predicates, by either passing them on to the core or replacing them by NOPs. The filter also manages a loop counter stack that holds the iteration counters of loops in single-path code, a return address stack that stores the return addresses of single-path function calls and recursion counters. To control the behavior of the single-path filter, the instruction set must be extended with special single-path instructions, which modify the state of these hardware stacks. Unused opcodes in the instruction set are used to encode these special instructions, which replace conditional control-flow instructions when generating single-path code and are parsed and applied directly by the filter when fetched by the processor core.

Single-path code requires the ability to conditionally modify predicates since the predicates are used to capture the truth value of conditions. Therefore, the instruction filter needs access to the results of comparisons in the core. On most architectures, condition codes are used to capture the results of compares and to evaluate conditions. Hence, by giving the instruction filter access to these condition codes, it can evaluate conditions analogously to the processor core and modify predicates accordingly.

The present implementation requires that all predicates on the predicate stack are true in order to enable instructions and thereby forward them to the core. Although the hardware implementation does not differentiate between different types of predicates, they are distinguished logically based on the purpose they serve in single-path code.

1. *Conditional predicates:* A conditional predicate is pushed to the stack for each conditional statement (e.g., *if-then-else* statements). The conditional predicate is initialized based on the result of a condition and remains on the stack for as long as the condition applies.

2. *Loop predicates:* Each loop has a loop predicate which is the first predicate pushed to the predicate stack when entering a loop and the last predicate removed when exiting the loop. The loop predicate is true as long as the loop condition is true. Once cleared, it remains false for all remaining loop iterations.

3. *Iteration predicates:* In addition to the loop predicate, every loop also has an iteration predicate. The iteration predicate is set to true at the beginning of each loop iteration and is cleared if one iteration of the loop is aborted without exiting the loop, such as would happen when encountering a *continue* statement.

4. *Function predicates:* Single-path code requires that all instructions of a function are always executed. Hence, an early return from a function is realized by clearing a dedicated function predicate. Each function has a function predicate which is the first predicate pushed to the predicate stack upon entering the function, and conversely the last predicate popped from the stack upon leaving that function.

Fig. 4.2 shows the C code for a simple conditional statement, along with pseudo-assembler representations of the regular version as well as of the single-path version of the machine code for that conditional. The generic operations *OP_A*, *OP_B*, *OP_C*, and *OP_D* represent instructions from the processor's native instruction set. *OP_A* is executed unconditionally prior to the conditional block. *OP_B* is executed if the condition *COND* is true. Otherwise, *OP_C* is executed instead. Finally, *OP_D* comes after the conditional block and is again executed unconditionally. In regular machine code, the conditional execution of either *OP_B* or *OP_C* is realized with control-flow instructions. A conditional branch instruction moves control to the *else* label if *COND* is false, thus executing *OP_C*. Otherwise, *OP_B* is executed, and then an unconditional jump takes control to the end of the conditional block. The single-path version, by contrast, does not use any control-flow instructions. Instead, a new predicate is pushed to the stack, and that predicate (with index 0 since it is at the top of the stack) is cleared if *COND* is false. Hence, the predicate at the top of the stack initially corresponds to the truth value of *COND*, and therefore the operation *OP_B* is only enabled if *COND* is true. Then, the value of the predicate is inverted, thereby enabling *OP_C* if *COND* is false. The right column shows the state of the predicate stack depending on the truth value of *COND* for each of the generic operations.

Fig. 4.3 shows a similar representation for a simple loop. This time, however, the single-path version also contains a control-flow instruction. This is a special instruction that is used in conjunction with a loop counter, which will be replaced either by a jump to the start of the loop as long as the loop counter is not 0 or by a NOP to exit the loop when the loop counter reaches 0. The loop counter is pushed to the loop counter stack and initialized with the loop bound specified in the annotation before the start of the loop. It is then decremented on each iteration. Loops use a loop predicate to capture the state of the loop condition and an iteration predicate that replaces backward jumps to the start of the loop (e.g., via a *continue*-statement in C code). While the loop predicate at index 1 in the predicate stack is cleared if the loop condition *COND_A* is false and then remains false for all remaining iterations, the iteration predicate at index 0 is conditionally cleared if *COND_B* is true for one loop iteration only and is reset to true for the next iteration. Both of these predicates are pushed to the predicate stack before entering the loop and removed from the stack after the loop has been left.

The single-path filter substitutes the instructions fetched from memory by NOPs when any of the predicates on the stack is false. In order to achieve constant execution time, that substitute instruction must have the same execution time as the original instruction. Which and how many instructions are used for this purpose will therefore

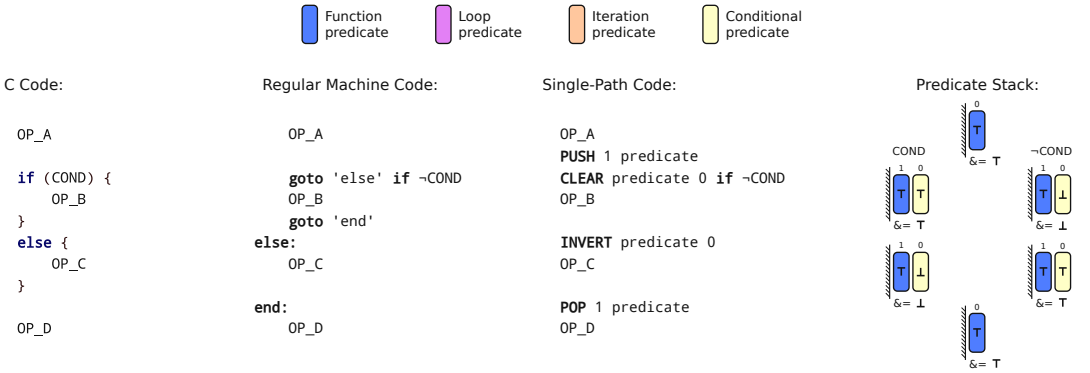


Figure 4.2: Example of a conditional statement in single-path code: While regular machine code uses control-flow instructions to conditionally execute code, in single-path code predicates are used instead.

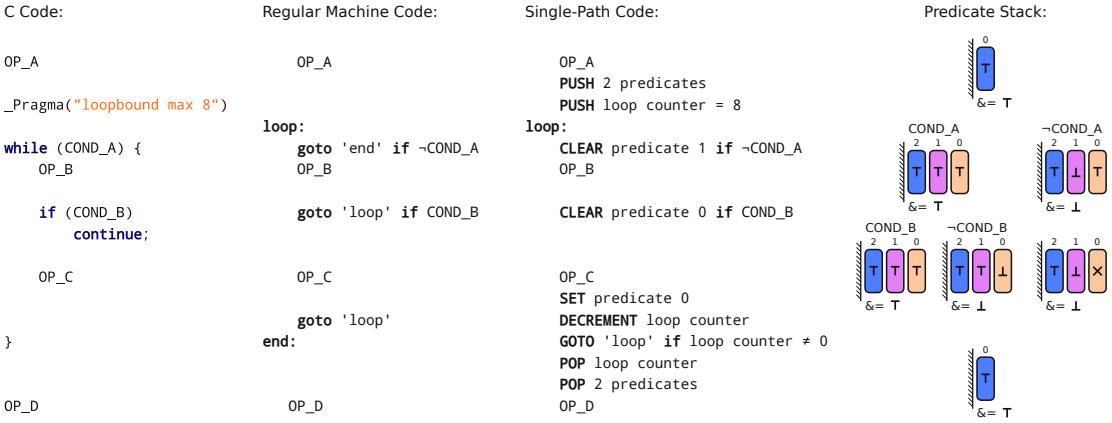


Figure 4.3: Example of a loop in single-path code: The loop bound annotation is used to initialize the loop counter in single-path code and the loop is executed for a constant number of iterations. The loop predicate capturing the loop condition and the iteration predicate, which is cleared by a *continue* statement and reset at the start of each iteration, control whether the instructions are actually active.

depend on the specific processor. On architectures that use a hard-wired zero register (i.e., a register that always reads as 0 and cannot be written), the destination register of an instruction can simply be replaced by that zero register, in which case the instruction has no effect.

The single-path filter can conceptually be integrated into any processor core, ranging from simple in-order cores to superscalar processors. The only important requirement is that the filter needs access to the condition codes in order to store the result of compare instructions into predicates. Also, while the presence of optimizations such as data caches and dynamic predictors does not impede the operation of the single-path filter, it would most likely affect the timing-predictability that has been gained by adding

the filter. A notable exception are branch predictors, which are commonly used to speculatively execute instructions following a data-dependent conditional branch before the outcome of the condition and thus the branch target are known. Since single-path code does not use conditional branches, the presence of a dynamic branch predictor does not impede its timing predictability. The single-path filter is evaluated on processors that are implemented in a hardware description language which can be synthesized in an FPGA.

4.3 Implementation details

The single-path generation and the instruction filter have been implemented for two Reduced Instruction Set Computer (RISC) processors:

1. LEON3, a SPARC v8 processor core developed by Cobham Gaisler for safety-critical applications [5].
2. ARM Cortex-M0, a processor core developed by ARM that uses the 16-bit ARM Thumb instruction set [63].

Both processors have a multi-stage in-order pipeline with predictable timings. Thus, they execute a given single-path program in constant time regardless of input values. They are synthesized as soft-cores on an FPGA together with the proposed single-path filter extension.

The single-path filter is controlled by special single-path instructions that are added to the respective instruction sets of these two architectures. Unused opcodes in both the SPARC v8 and the ARMv6-M Thumb ISAs are used to encode these special instructions in order to support the LEON3 and the ARM Cortex-M0 processors, respectively.

Figure 4.4 shows the selected encoding for single-path instructions for both instruction sets. For the SPARC v8 ISA, the unused opcode 1 of instruction format 2 is used for the custom instructions, with bits 31 and 30 cleared to indicate the instruction format and bits 24 through 22 holding the opcode with a value of 1. A 5-bit ID field identifies up to 32 individual single-path instructions and 22 bits of the instruction word are left for immediate values associated with the instruction. On the 16-bit ARM Thumb ISA the single-path instructions are encoded with two combined 16-bit instruction words. An unused opcode encoded by the 5 uppermost bits of the first instruction halfword are re-purposed for the single-path instructions. Again, 5 bits encode the specific type of single-path instruction and 22 bits remain for immediate values.

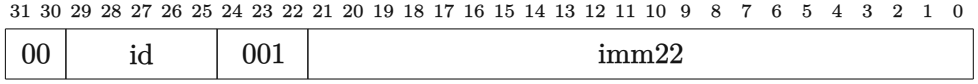
The single-path instructions that are processed by the filter are listed in Table 4.1. These instructions modify the internal state of the single-path filter by operating on the predicates that affect the execution of regular instructions, as well as the loop counter and return address stacks. These special instructions are only recognized by the single-path

filter but not by the processor core itself. Therefore, they are substituted by a NOP in most cases, except for instructions that alter the control-flow, such as the instruction *LOOP BRNZ*, which jumps back to the head of a loop unless the loop counter has reached 0. These instructions are replaced by unconditional jump instructions such that the main core performs the necessary control-flow changes required by the single-path code.

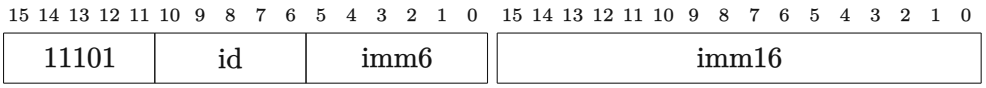
Besides interpreting and executing these special instructions, the single-path filter also needs to enable and disable regular instructions as required by the current state of the predicates. Disabled instructions are replaced by NOPs. However, the present implementation takes care to retain the same timing behavior, no matter whether an instruction is disabled or not, such that the execution time of the single-path code is truly independent of input data. Therefore, the single-path filter substitutes disabled instruction words with alternatives that have no effect but an execution time that equals the execution time of the disabled instruction.

The SPARC ISA uses a hard-wired zero register, i.e., a general-purpose register that holds the constant value 0 and cannot be modified. Such a zero register is common on many RISC architectures and typically has the address 0. It can conveniently be used to discard the result of an operation by using that zero register as the destination register. Hence, the single-path filter for the LEON3 processor can disable most instructions by simply overwriting their destination register address with 0. Memory accesses are an exception since a disabled memory instruction might use an invalid address but must not cause a bus error. A dedicated memory region is reserved for disabled load and store instructions, with disabled store instructions dumping the write data into that location and disabled load instructions reading from that location into the zero register, thus discarding whatever value has previously been stored there. The selected memory location is chosen such that it can be addressed with the immediate fields of the instruction word.

The 16-bit ARM Thumb instruction set, on the other hand, has no hard-wired zero register. Therefore, other means must be found to disable instructions while preserving



(a) SPARC-v8 Single-Path Instructions Encoding



(b) ARMv6-M Thumb Single-Path Instructions Encoding (note that these are two 16-bit instruction words combined into a 32-bit word)

Figure 4.4: Encoding formats for the special single-path instructions in the SPARC-v8 and the ARMv6-M Thumb instruction sets. The field *id* is used to identify the individual single-path instructions. For both architectures a total of 22 bits can be used to encode immediate values (see Table 4.1 for a list of single-path instructions and their respective use of the immediate field).

their timing on the ARM Cortex-M0. Fortunately, all arithmetic instructions have an execution time of exactly one cycle, the same execution time as the generic NOP instruction for that architecture. Memory accesses, in turn, can be substituted by stack-relative loads and stores (i.e., memory instructions that use an address relative to the stack pointer). Choosing a sufficiently large offset allows to redirect disabled memory accesses to unused stack regions below the address of the current stack pointer. A stack

Table 4.1: Special Predicate-Defining Instructions

Instruction	**Description**	**Immediate Field**
PRED PUSH	Push new predicates to predicate stack (initialized to true)	Number of predicates to push
PRED POP	Pop predicates from predicate stack	Number of predicates to pop
PRED SET	Set a predicate (change its value to true)	Index of the predicate to set
PRED INVERT	Invert a predicate (toggle its value)	Index of the predicate to invert
PRED CCLR	Conditionally clear a predicate (set its value to false)	3 bits: condition, rest: predicate index
LOOP PUSH	Push new loop counter to loop counter stack	Initial value of new loop counter
LOOP POP	Pop top loop counter from loop counter stack	Unused
LOOP BRNZ	Branch to start of loop if top loop counter is not zero, post-decrement top loop counter	Branch address (architecture-specific encoding)
SP CALL	Call a single-path function (push program counter to return address stack)	Call address (architecture-specific encoding)
SP RET	Return from single-path function (pop address from return address stack and jumps to that address)	Unused
RECUR ENTER	Enter a recursive function (increment the function's recurrence counter if it is below the recursion limit, otherwise return immediately)	Unique index for this recursive function
RECUR EXIT	Exit a recursive function (decrement the function's recurrence counter)	Unique index for this recursive function

analysis can be carried out to ensure that none of these accesses will ever interfere with any actual stack data.

4.4 Evaluation

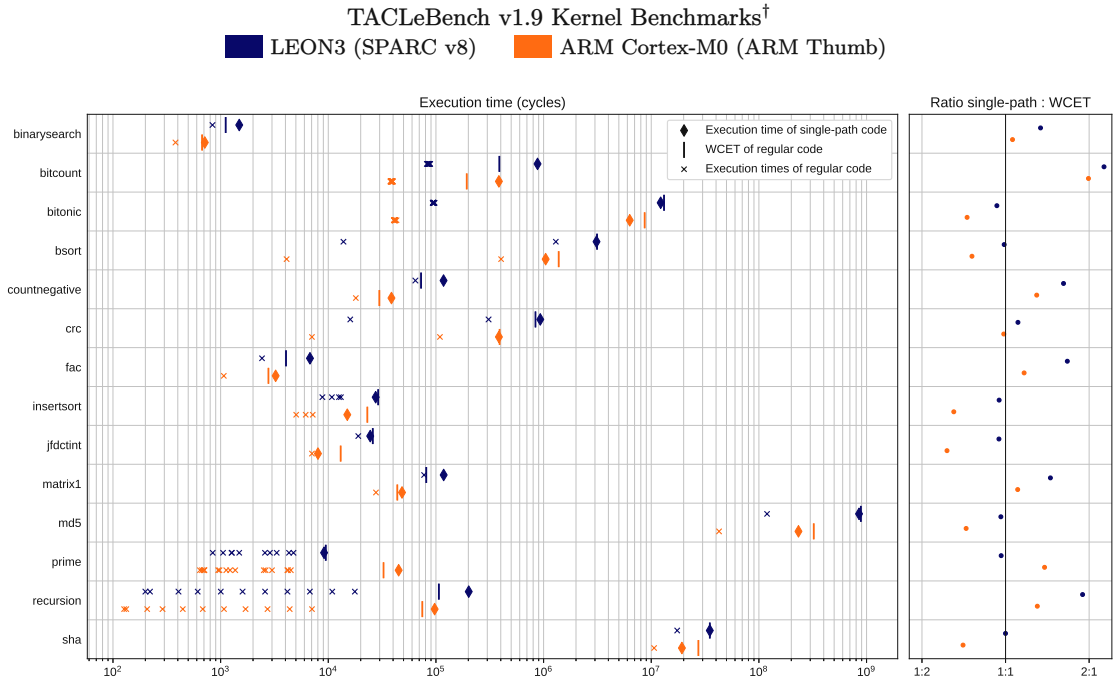
The performance of the proposed approach is evaluated by comparing the constant execution time of single-path code with the WCET of the equivalent regular code on both processors for a set of benchmark programs. For regular machine code, the WCET is the relevant metric in real-time systems since that is the amount of processing time that system designers need to reserve for the execution of a task. For single-path code, however, the execution time is constant on time-predictable hardware. Hence, the WCET of single-path code equals that constant execution time which can be determined by measuring it once.

The evaluation uses version 1.9 of the TACLe benchmark collection [39], a collection of benchmark programs targeted at WCET research. This benchmark collection contains several programs from the Mälardalen [50] and the MiBench [51] collections, among others, with annotations required for timing analysis, such as loop bounds, that are added to the code. Some of the benchmark programs had incorrect loop bound annotations or architecture-related errors (e.g., assumptions about the size of C data types that do not apply to all ISA). For these programs, either existing fixes that have not yet been included in an official release or custom patches, which have been submitted to the maintainers of the TACLe benchmarks and should be included in upcoming releases, have been used. One program had architectural issues on both ISAs. These problems could not be fixed and thus the program had to be removed from the evaluation.

For reference, WCET bounds have been determined for each program using the *aiT WCET Analyzer* [40], a widely used static timing analysis tool developed by AbsInt GmbH, which is able to find tight WCET bounds [49]. The annotations contained in the TACLe benchmark programs are used by the tool in addition to its capability of automatically extracting some flow-facts through value analysis. The WCET analysis thus uses the exact same loop bounds and recursion limits as the single-path transformation.

Fig. 4.5 and 4.6 show plots that compare the constant execution time of the single-path version with the WCET bound of the regular version on the LEON3 and the ARM Cortex-M0 processors of each program of the kernel and the sequential sets of the TACLe benchmarks, respectively. Those programs that use floating-point arithmetic have been excluded (neither of the two processors has a floating-point unit), as well as two programs which turned out to be unsuited for conversion to single-path code (see Section 4.5 for details) and one program (*ammunition* from the sequential set) which failed due to architectural issues that could not be fixed.

The left plot in each figure shows the absolute execution times of both the single-path and the regular version of each program on a logarithmic scale, as well as the WCET bound of the regular program. The execution time of the single-path version is constant.

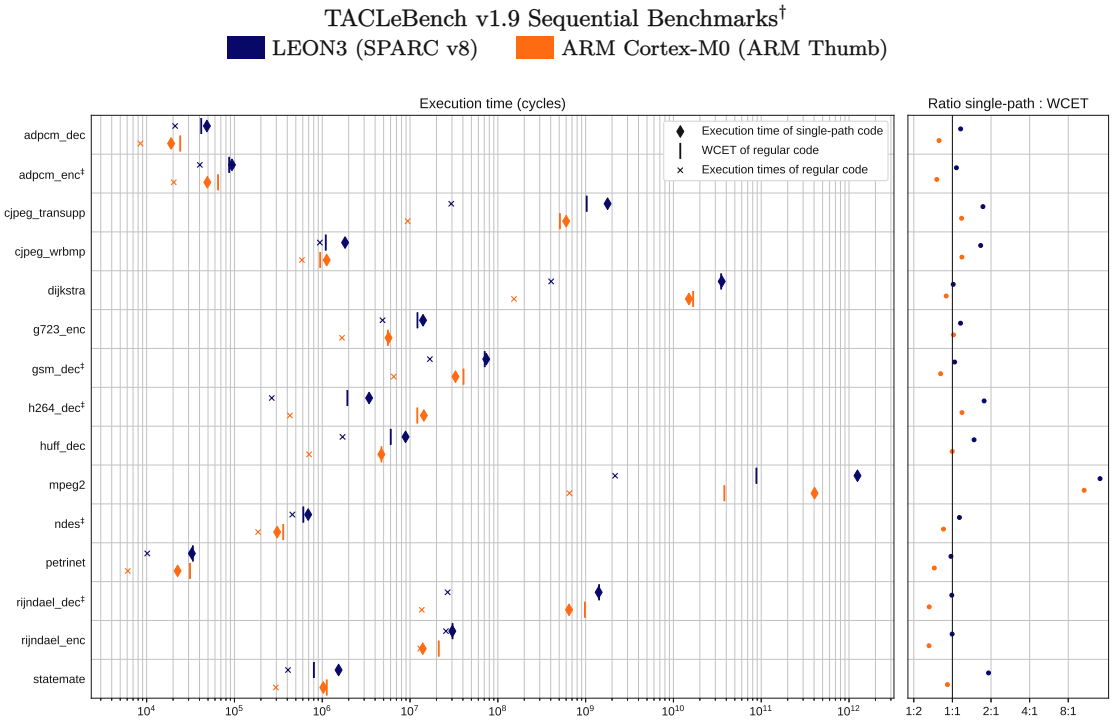


† Excluding programs using floating-point arithmetic.

Figure 4.5: Comparison of the execution time of the single-path version with the WCET bound of the regular version of the TACLeBench benchmark programs. The large plots on the left show the constant execution time of the single-path version of each benchmark program as well as the WCET bound and execution time of the regular code version of that program in CPU cycles. The narrow plots on the right show the ratio between the execution time of the single-path version and the WCET of the regular version for each program. The WCET bounds have been obtained with the *aiT WCET Analyzer* from AbsInt GmbH.

Hence, there is only one sample for each program, depicted by a diamond symbol. By contrast, the execution time of the regular version may vary, in which case the plot shows several samples as crosses, although frequently the test input data for many benchmark programs is constant rather than initialized randomly, which leads to constant execution times for the regular version as well. The WCET bound of the regular version of each program that has been determined with the *aiT WCET Analyzer* is depicted by a vertical bar. Execution times and bounds on the LEON3 are shown in dark blue and those on the ARM Cortex-M0 in orange.

Additionally, the narrow plot on the right of each figure highlights the ratio between the constant execution time of the single-path version and the WCET bound of the regular version of each program. If the ratio is 1:1, then the execution time of the single-path code equals the WCET bound of the regular code. A larger ratio indicates that the single-path code performs worse, with its execution time being larger than the



† Excluding programs using floating-point arithmetic, with infeasibly large WCET and *ammunition* which failed due to architectural issues. ‡ Programs that have been patched.

Figure 4.6: Comparison of the execution time of the single-path version with the WCET bound of the regular version of the TACLeBench benchmark programs. The large plots on the left show the constant execution time of the single-path version of each benchmark program as well as the WCET bound and execution time of the regular code version of that program in CPU cycles. The narrow plots on the right show the ratio between the execution time of the single-path version and the WCET of the regular version for each program. The WCET bounds have been obtained with the *aiT WCET Analyzer* from AbsInt GmbH.

bound of the regular code. However, for several programs, the constant execution time of the single-path version is less than the WCET bound of the regular code.

It turns out that the constant execution time of the single-path version of the benchmark programs is typically between twice as slow to twice as fast as the WCET of the regular version. Which version is faster apparently depends on the structure of the control-flow graph of the program itself, with the single-path variant of some benchmark programs, such as *bitonic*, always performing better than the regular version, while for some other programs, such as *bitcount*, the single-path version is always slower than the WCET bound of the regular version. Additionally, the processing platform seems to have an effect on the relative performance of single-path code w.r.t. the WCET bound

of regular code. One can see that the single-path versions performs better on the ARM Cortex-M0 than on the LEON3 for most benchmark programs.

The control-flow graph of a program obviously has a strong influence on the relative performance of its single-path version since all possible code branches need to be executed unconditionally. In particular, programs with conditional statements that have multiple branches with long execution times incur a significant performance penalty when transformed to single-path code. This effect is particularly pronounced for the program *mpeg2* from the sequential set of the TACLe benchmarks (see Fig. 4.6), for which the execution time of the single-path version is about an order of magnitude larger than the WCET of the regular version.

In general, the WCET of a program cannot be reduced by transforming it to single-path code (with the exception of some marginal improvements if the execution platform has a large branch-misprediction penalty). Yet, the results show that single-path code frequently performs better than the WCET bound of the regular code of a program. The main reason for this is the over-estimation of the actual WCET by the timing analysis software. Similarly, the fact that single-path code appears to perform better on the ARM Cortex-M0 than on the LEON3 is most likely due to the analysis software deriving tighter WCET bounds for the LEON3, possibly because that processor is better supported due to its frequent use in industrial real-time systems.

The main advantage of single-path code over static timing analysis is that timing analysis becomes trivial. For processors with predictable execution timing the execution time of single-path code is constant. Therefore, the WCET is determined by measuring it on the actual hardware. Hence, single-path code is not affected by the quality and accuracy of a timing model or the sophistication of the analysis software, and it is not limited to architectures supported by these tools. The results show that the perceived inefficiency of executing all instructions unconditionally does not contribute to a significant overhead when compared to the WCET bounds of regular code for most of the benchmark programs.

The code size of single-path code has also been compared with that of regular code. Fig. 4.7 shows the amount of instruction memory space required for the regular versions as well as the single-path versions of the TACLe benchmark programs. This reveals that the single-path transformation increases code size by about 20–30 % on both architectures. Roughly 5–7 % of the regular machine code consists of control-flow instructions, while for single-path code, it is only about 2 %. Instead, single-path code comprises 6–9 % of special instructions that manipulate the predicates and between another 14–17 % of additional overhead. Interestingly, the proportion of special predicate manipulating instructions is larger for the ARM Thumb architecture than for the other instruction set. The reason is probably that while most of the ARM Thumb instructions are 16-bit wide, these special instructions were all encoded as 32-bit instructions because almost all of the 16-bit instruction encoding space has already been assigned by the standard.

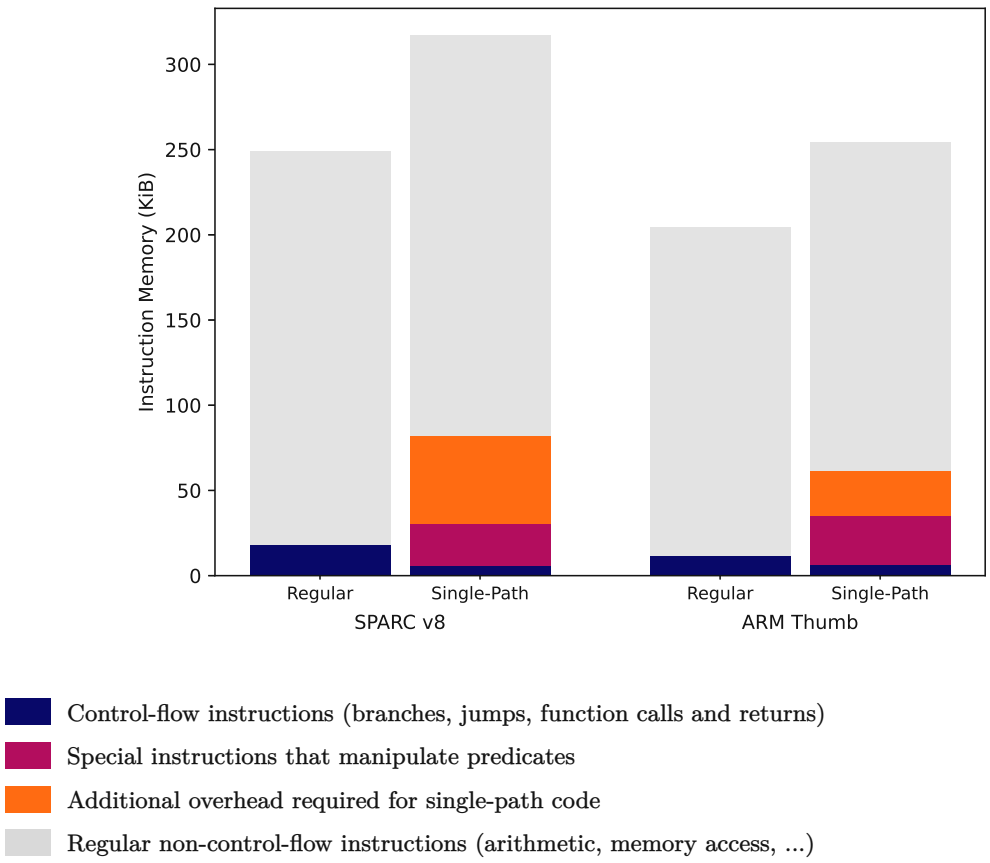


Figure 4.7: Code size of the regular machine code and the single-path versions of all TACLe benchmark programs used for the evaluation combined.

4.5 Limitations of Single-Path Code

The results of the evaluation show that the performance of single-path code with respect to the WCET bound of the equivalent regular code depends upon the program itself as well as on the amount of pessimism of the WCET bound. Some TACLe benchmarks appear to be better suited for single-path code than others. This section discusses some of the programming constructs and algorithms that cause these variations and derives recommendations for writing programs that perform well when converted to single-path code.

As already mentioned in Section 3.2, the WCET of a loop does not increase in single-path code since loops are simply executed for the maximum number of iterations. However, the WCET of a conditional statement with more than one branch increases when transformed to single-path. While in regular code, only one branch of a conditional statement is executed, single-path code executes all branches (but only one of them is

enabled by the predicates). The execution time of a conditional statement in single-path code is the sum of the execution times of all branches plus any overhead for evaluating the condition. In regular machine code, however, the WCET of a conditional statement is the maximum of the execution times of the individual branches. As a consequence, conditional statements with multiple lengthy branches should be avoided in single-path code.

The single-path version of the program *bitonic* from the kernel set of the TACLe benchmarks has a shorter execution time on both processors than the WCET of the regular version. Studying the source code of that program showed that it contains conditional statements with only one branch (i.e., simple *if*-statements without an *else*-branch), as well as a few loops. Therefore, there are no alternative branches whose execution time would sum up and thus the single-path transformation does not increase the WCET of that program.

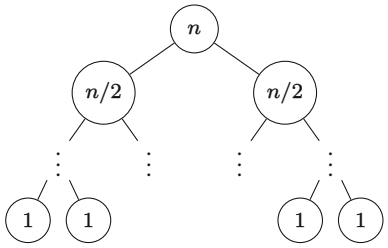
For the program *bitcount*, on the other hand, the execution time of the single-path version is larger than WCET of the regular version on both cores. The source code reveals that the main function of this program has a conditional statement with eight branches (implemented as a *switch*-statement with eight *case*-statements), each of which is calling a function. The whole conditional statement is placed within two nested loops and hence executed several times in a row. Here the single-path version clearly looses in efficiency since all branches need to be executed sequentially, while WCET analysis can take advantage of the fact that in the regular version only one of the branches is executed.

During this analysis, some programming constructs were encountered that were not well suited for conversion to single-path code, but that could easily be converted to a more efficient variant. Fig. 4.8 shows an example of a construct that becomes particularly expensive in single-path code. In the example program on the left, the function *func* is called with a different argument depending on a condition. In the respective single-path version, the function is called twice (only one call will have actual effects), while in the

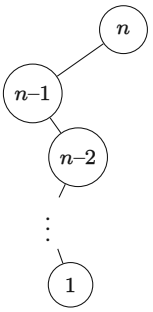
```
if COND then
  func(1)
else
  func(2)
end if
```

```
if COND then
  a = 1
else
  a = 2
end if
func(a)
```

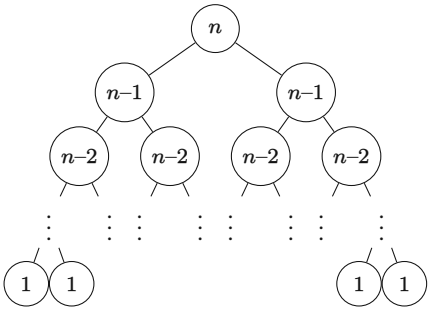
Figure 4.8: The program on the left requires that the function *func* is called twice in single-path code, once with 1 as argument and once with 2. The code on the right avoids the repeated calls by moving the function call out of the conditional statement and instead conditionally assigning the value of the argument to a temporary variable which is then passed to the function.



(a) Best-case for regular quicksort



(b) Worst-case for regular quicksort



(c) Single-path quicksort

Figure 4.9: Call trees of the recursive quicksort algorithm implemented in regular machine code in the best-case as well as the worst-case situation and for a single-path version of the algorithm. In the best-case scenario a regular quicksort implementation divides the list into two sublists of equal length during each recursive call and hence requires a call tree with a depth of $\log_2 n$, yielding an execution time of $O(n \cdot \log_2 n)$; in the worst-case scenario all elements except for one are in the same sublist on every recursive call and the call tree becomes a linear chain with a depth of $n - 1$, increasing the execution time to $O(n^2)$. Single-path code requires that the union of all possible call trees is executed, which requires $n^2 - 1$ nested calls and has an execution time of $O(n^3)$.

regular code version, the function would always only be called once. Fortunately, this inefficiency can be avoided by assigning the argument of the function to a temporary variable, as shown in the program on the right. That way only the assignment of the variable is conditional and hence executed twice in single-path code, while the function is called once after that assignment.

A very similar construct that was encountered frequently is that two branches of a conditional statement read the same value from an array and then assign it to different variables. The way these were usually implemented meant that the calculation of the memory address based on the array index as well as loading the value would happen within each of the two branches and so that code would be executed twice in single-path code. In general, several instances were encountered where code was duplicated within the branches of a conditional statement and could easily be moved out of the statement, thus improving the execution time of the single-path version.

Additionally, some algorithms have been identified that are not very well suited for single-path code. One of these is the well-known sorting algorithm *quicksort* [57], which sorts a sequence recursively by choosing a pivot element and then dividing the sequence into two sublists, one of which contains all the elements that are less than the pivot element and the other one containing all those elements that are greater. The two sublists are then passed to recursive calls of the sorting function, which return the sorted sublists. The final sorted sequence is obtained by concatenating the first sublist, the pivot element and the second sublist.

The regular machine code for a quicksort implementation has an execution time between $O(n \cdot \log_2 n)$ and $O(n^2)$ [58]. In the best case, the pivot element is always chosen such that the sequence is divided into two equally sized sublists with $n/2$ elements during every recursive call. In that case the algorithm requires $\log_2 n$ nested calls of the sorting function. In the worst case, all elements of the sequence are either less than the pivot element or all elements are greater, such that the entire original sequence minus the pivot element ends up in the same sublist on every recursive call, thus requiring $n-1$ nested calls. Fig. 4.9 shows the call graphs for these two extreme cases, as well as the call graph for a single-path version of quicksort. The call graph of the single-path version is the union of all possible call graphs since every recursive function call is executed unconditionally, despite the majority of these being disabled. Single-path code needs to take into account both worst-case scenarios: the scenario where all elements are less than the pivot element and the scenario where all elements are greater than the pivot element, which requires two recursive calls that can each support a sublist with up to $n-1$ elements. Hence, the single-path transformation increases the execution time of quicksort to $O(n^3)$.

The programs *anagram* and *huff_enc* from the sequential set of the TACLe benchmarks use a quicksort implementation borrowed from the GNU C library [14] that becomes particularly inefficient in single-path code. In that implementation, the sorting function conditionally calls itself from within a loop, which means that in single-path code, the recursive call is executed unconditionally on every loop iteration (and the loop is always

executed for the maximum number of iterations). The iteration bound of that loop is 8, thus each call of the sorting function results in another 8 recursive calls. The total number of nested calls therefore is $8^n - 1$ (with n being the maximum length of the sequence to sort and hence the maximum depth of the call tree). In the programs *anagram* and *huff_enc* the maximum length of the sequence to sort is 17 and 256, respectively. Hence, the single-path version of these programs requires a total of $2.25 \cdot 10^{15}$ and $1.55 \cdot 10^{231}$ nested calls, respectively. Both programs do not terminate in a reasonable amount of time and had to be excluded from the evaluation.

As an alternative to quicksort, *mergesort* is more appropriate for a single-path program. This algorithm always divides a sequence into two sublists of equal length, recursively sorts both of these individually and then merges the sorted sublists into a sorted sequence [46]. While the call graph of *quicksort* depends on the values of the pivot elements, the call graph of *mergesort* has no data dependencies, since the input sequence is always divided into two sublists of length $n/2$ irrespective of the values in the sequence. Therefore, the depth of the call graph is limited to $\log_2 n$, which allows to set tighter recursion limits for the single-path version.

While *quicksort* is an extreme example, there are likely several other algorithms that are less suitable for single-path code, with more appropriate alternatives available. The programs in the TACLe benchmark collection aim to be representative of programming schemes and algorithms used in various areas [39]. These usually optimize for average-case rather than worst-case performance, since average performance is what matters in non-real-time applications [99]. However, for real-time systems worst-case performance is critical, as it is for single-path code, which always executes all possible branches of a program, including the worst-case branch. Therefore, replacing algorithms and programming schemes with ones that are more suitable for single-path code would likely improve the execution time of the single-path versions of the benchmark programs.

The limitations discussed in this section are related to programming techniques and algorithms that take advantage of features found in most widely available processor architectures. Single-path code with its predicated execution requires a different way of thinking about program design and algorithms in order to avoid programming constructs that favor established processor designs and code generation techniques but fail to execute efficiently on predictable hardware.

4.6 Findings

Single-path code simplifies timing analysis by merging all possible execution paths of a program into a single path. This is achieved by using predicated execution instead of conditional control-flow changes. The overhead caused by predicated execution is kept to a minimum when fully predicated execution is available. However, hardly any processor architectures support it, thus confining single-path code to those few that do.

The single-path filter presented in this chapter brings support for fully predicated execution to existing processor designs. This opens up the possibility of efficiently executing single-path code on a wide range of architectures.

The single-path filter is well suited for single-core processors. However, many modern workloads are inherently parallel [69]. Therefore, the next chapter discusses a way to parallelize the concept of predicated execution.

CHAPTER 5

A Vector Coprocessor for Data-Parallel Real-Time Workloads

The content of this chapter is based on and extends a conference paper titled *Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation* [93].

So far, this thesis has dealt with timing-predictable computer architectures that are simple single-core processors executing a single stream of instructions, with each instruction producing one result at a time. Yet, parallelism is of major importance in nowadays computer architectures. Real-time systems are no exception, as many emerging time-critical applications require the performance offered by parallel architectures [69]. Therefore, it is essential to discuss how the concepts introduced so far can be parallelized.

Chap. 3 and 4 explained the concept of predicated execution and how it can be used to ease timing analysis by eliminating data-dependent control-flow branches. The present chapter introduces a kind of data-parallel computer architecture that allows to naturally extend this concept to parallel processing.

A straightforward way to create a parallel architecture based on a simple traditional processor core is to replicate it multiple times and to use a low-level network to connect the individual cores to the memory system, thus creating a multi-core architecture. While this approach has been chosen numerous times already in the context of timing-predictable computing [113, 124], the present work instead focuses on vector processors, a kind of processor architecture that is inherently data-parallel.

A vector processor implements the single-instruction multiple-data (SIMD) paradigm with instructions operating on a vector of elements instead of individual values. In contrast to array processors, the elements in a vector are not only processed simultaneously

using multiple processing elements but also sequentially over several clock cycles [7]. Consequently, the length of a vector is not limited by the number of processing elements and is usually configurable, allowing potentially very large vectors to be manipulated by a single instruction. Thus, the cost of fetching and decoding the instruction is amortized over a large amount of data, which aids energy-efficiency and also reduces the effectivity of complex hardware-level optimizations [85]. Therefore, the timing-predictability of vector processors can be achieved by avoiding certain optimizations without significantly impacting their performance.

On vector processors, it is usually possible to conditionally modify individual elements of a vector by using vector masks. These vector masks can be setup manually or can be the result of comparisons between individual vector elements. As such, vector masks allow a form of element-wise predicated execution, where an instruction can be enabled or disabled individually for each element of a vector. Thus, vector processors parallelize the concept of predicated execution.

This chapter presents the timing-predictable RISC-V vector processor Vicuna, which is tailored to the needs of time-critical systems [93]. Vicuna has been implemented in SystemVerilog and compliant with the recently ratified version 1.0 of the official RISC-V vector extension, RISC-V V [105]. Vicuna is a coprocessor and requires a main core to function. It has been integrated with the open-source RISC-V core Ibex [110]. This combined processing system is free of timing anomalies and, therefore, well suited for use in real-time systems. Despite its timing-predictability, Vicuna retains a peak performance of over 10 billion operations per second on a Xilinx 7 Series FPGA. Evaluations on data-parallel benchmarks show its efficiency is over 90 % for compute-bound workloads.

5.1 Parallel Processing Architectures

This section presents an overview of parallel processing platforms in general, as well as a detailed analysis of existing vector processors. The main aspects are summarized in Table 5.1.

Micro-architectural optimizations have been one of the primary means to increase the performance of processors since the break-down of Dennard scaling at the beginning of the century. However, as the performance gains achieved via these optimizations began to dry up, computer architects were forced to use varying degrees of parallelism to enhance performance further [62].

A straightforward way to implement parallelism is to use several processor cores, each executing an individual stream of instructions. This approach, referred to as the multiple-instruction, multiple data (MIMD) paradigm [41], has since allowed continuing increases in performance. Despite the prevalence of multi-core systems, it turns out that applications which effectively use this parallelism are often highly data-parallel and thus, the cores end up all executing the same instructions [24]. The fetching and decoding of identical instructions throughout multiple cores are not only a waste of

resources and energy but also increase the pressure on the shared memory interface. Multi-core processors usually connect the individual cores to the memory system via a NoC [13], and the efficient design and use of these NoCs is an active research area with a number of challenges [82]. In particular, *application mapping*, which assigns the available bandwidth on the NoC to the individual cores (or clusters of cores) according to application-specific needs, is a key factor in determining the performance of the overall system. Unfortunately, the problem of application mapping is, in general, NP-hard [94]. Hence efficiently utilizing the cores in such a system is challenging [107, 119, 4], particularly when safety requirements demand certain guarantees with respect to the bandwidth and latency of the NoC [67, 95]. As a result, the performance of the individual cores in a multi-core system is increasingly limited by the NoC as more cores are added. Schoeberl et al. found that the performance of the T-CREST platform scales only logarithmically with the number of cores [113]. Therefore, although being very flexible, multi-core systems are not well suited for data-parallel processing tasks.

Several parallel processing architectures which are a better match for data-parallel workloads have been proposed. Graphics Processing Units (GPUs) are increasingly used for massively parallel tasks [90], such as for instance machine-learning algorithms. GPUs take advantage of the single-instruction multiple-threads (SIMT) paradigm [76], as a single instruction stream controls data processed across several hardware threads, which enables highly efficient processing of huge quantities of data. GPUs also found their way into safety-critical systems in domains such as autonomous driving [36, 69, 47], where they are indispensable for processing vast amounts of sensor data. However, the use

Table 5.1: Performance and timing predictability of parallel computer architectures

Processor Architecture	Multi-Core CPU	General-purpose GPU	Domain-Specific Accelerators	Existing Vector Processors	Timing-Predictable Platforms	**Vicuna**
General-purpose	✓	✓		✓	✓	✓
Efficient parallelism		✓	✓	✓		✓
Timing-predictable			✓		✓	✓
Max. OPs per sec ($\cdot 10^9$) FPGA / ASIC	2.2 * / 1 200 **	3.2 † / 35 000 ††	5 000 ‡ / 45 000 ‡‡	15 § / 128 §§	2.4 ¶ / 49 ¶¶	10 / —

* 16-core Cobham LEON3
** 344-core Ambric Am2045B
† FlexGrip soft GPU [6]
†† NVIDIA RTX 3090
‡ Srinivasan et al. [120]
‡‡ Google TPU [66]
§ 32-lane VEGAS [22]
§§ 16-lane PULP Ara [19]
¶ 15-core T-CREST Patmos [113]
¶¶ 8-core ARM Cortex-R82

of GPUs in real-time systems still poses several challenges [34]. Most GPUs are not preemptive, which instead requires software-preemption techniques [47], and modeling their timing behavior is complicated by undisclosed arbitration techniques used to resolve contention for shared resources such as the memory bus [35]. In addition, the timing analysis of heterogeneous systems comprising a CPU and a GPU with separate memories introduces its own challenges [108, 34].

An alternative to GPUs enjoying a much tighter integration with the main core is single instruction multiple data (SIMD) arrays which have been added to several popular ISAs over the last years. These are additional functional units added to a processor core that are capable of processing several elements stored in a fixed-sized array at once. Their disadvantage is that the computational resources need to be replicated for each element, and new instructions are required to take advantage of the increased array size [19].

Recently, the advent of machine learning has fueled the development of new special-purpose accelerator architectures. These architectures only support specific operations required for a narrow set of applications and trade flexibility and often precision for unprecedented computational performance, such as the $45 \cdot 10^{12}$ operations per second achieved by the Tensor Processing Unit (TPU) [66]. An interesting side-effect of these domain-specific accelerators is that due to their simple architecture, their timing behavior is usually much simpler to analyze than that of other processing architectures [85]. However, the impressive peak performance figures are only reached for workloads that fit the intended purpose of the accelerator, while other tasks either suffer drastically reduced performance or cannot be executed at all. By contrast, a general-purpose vector processor can execute any task that can be run on a conventional processor.

5.1.1 Vector Processors

Vector processing is similar to array processing, with one instruction operating on several elements. However, while an array processor requires a dedicated processing element for each data element to be processed, a vector processor has the added flexibility of processing a vector of elements not only simultaneously but also sequentially over several clock cycles. This allows variable vector lengths and the processing of large quantities of data by a single instruction. Fig. 5.1 depicts the differences in the processing patterns of array processors and vector processors. The array processor features a number of processing elements, and the length of the array that can be operated on by one instruction is limited by that number. In contrast, vector processors typically comprise specialized execution units which are capable of processing a certain number of elements concurrently, but more importantly, the elements are also processed over several cycles.

The ability to process a large vector of data without the need to replicate computational resources for each element makes vector processors particularly energy-efficient [24]. Vector processors handle the bottleneck imposed by a narrow memory interface shared for instructions and data, which is referred to as the von Neumann bottleneck, very effectively [12]. In a vector processor, the fetching and decoding of instructions are

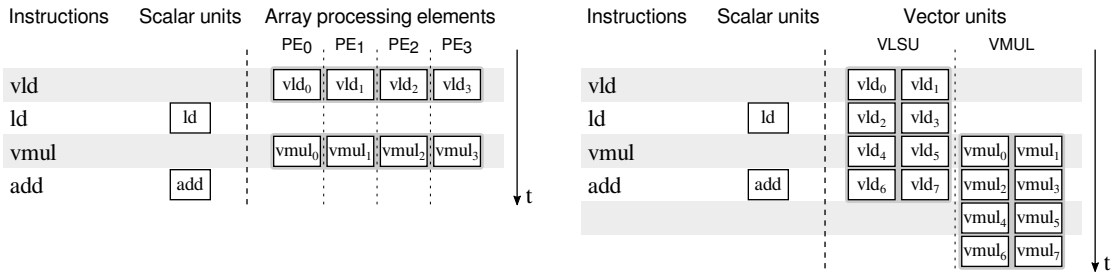
amortized over a potentially very large vector. Thus, vector processors have the potential to surpass even GPUs, for which instructions are amortized over a fixed block size only, in terms of efficiency [19].

Vector processing used to be popular in the 1960s and 70s, when most of the supercomputers were vector processors, such as the Illiac IV [61] or the Cray series [106]. These processors were modular designs comprising thousands of integrated circuits. However, towards the end of the century, they were superseded by integrated microprocessors, which achieved much higher clock frequencies [7].

Despite vanishing from the high-end computing market, vector processors continued to exist as general-purpose accelerators. In particular, several designs targeting FPGAs, which extend popular soft-core processors, such as the Altera Nios II/f [3], have been developed. For instance, the VESPA [133], VIPERS [134], and VEGAS [22] architectures all add vector processing capabilities to a MIPS-based or Nios main core. These designs have subsequently been refined, with VENICE [116], an area-efficient improved version of VEGAS, or MXP [117], which added additional support for fixed-point computation, being proposed.

In the last few years, vector architectures have been re-gaining attention as energy-efficient parallel processing platforms. The vector coprocessor Hwacha [75] is based on the open RISC-V ISA [126]. Despite inspiring many features of the new RISC-V vector extension, Hwacha uses a custom RISC-V extension that is not compatible with the now finalized official V extension. Hwacha is part of the RocketChip core generator [8], and several processor designs that use Hwacha as a coprocessor have been presented [131, 111].

To date, a major impediment to the more widespread adoption of vector processors has been the lack of standardization and tool support. All vector processors discussed so far use custom extensions to existing ISAs for vector instructions. Since vector processors have played no role in high-performance computing in the last decades, there is no



(a) Array processor. (b) Vector processor.

Figure 5.1: Comparison of the execution patterns of array and vector processors. Instructions prefixed with a *v* operate on a vector of elements, while the rest are regular scalar instructions. On an array processor, the number of available processing elements limits the amount of elements that can be operated on by a single instruction, while on a vector processor large data vectors can be processed over several clock cycles.

support for these extensions in major compilation toolchains. However, this is about to change as several major ISAs are extended with vector processing capabilities, such as the ARM ISA, which recently adopted the Scalable Vector Extension (SVE) [121] or the open-source RISC-V ISA [126] for which a vector extension is proceeding towards ratification [105]. Work is underway to add support for these extensions to the GNU Compiler (GCC) as well as the LLVM compiler suites, with initial support for the SVE and the RISC-V V extension available in the latest release of both of these toolchains.

One of the first vector architectures implementing the RISC-V V extension is Ara[19], which serves as a vector coprocessor for the application-class RISC-V core CVA6 [135]. However, Ara is a performance-oriented platform that uses optimizations that are known to cause timing anomalies, such as banked register files, greedy memory arbitration [55], and runtime decisions for selecting functional units [127].

Recently, several new processing architectures adopting the RISC-V vector extension have been proposed, some of which target resource-constrained embedded systems and are deliberately kept simple, which might facilitate the analysis of their timing behavior. However, these architectures frequently implement a small subset only of the full RISC-V V extension. The choice of supported instructions is usually geared towards a specific use case, with machine learning being by far the most prominent application. For instance, RISC-V^2 [91], Arrow [10], and the minimal vector processor presented by Johns and Kazmierski [65] are accelerators for machine learning implementing only a few select V extension instructions. Similarly, VPQC [132] is a RISC-V-based cryptography accelerator that supports some of the V extension's instructions which are required for certain post-quantum cryptographic algorithms.

Although these domain-specific architectures achieve significant performance for certain tasks, they lack the versatility of a general-purpose vector processor. In particular, RISC-V^2 and Arrow do not support any of the V extension's vector permutation instructions, which simplifies their design by eliminating any inter-connections between individual lanes. However, due to the lack of support for vector permutation instructions, any algorithms that require the re-ordering of vector elements, such as many signal processing algorithms (e.g., the DFT) or most cryptographic algorithms (e.g., AES encryption), cannot be executed on these platforms (or have to implement permutations via costly memory transactions). By contrast, Vicuna implements almost the entire RISC-V V extension, with the exception of floating-point instructions, which allows parallelizing any algorithm that can be executed on a regular processor using fixed-point arithmetic.

Vicuna is the first and, at the time of writing, only timing-predictable general-purpose vector processor. It has been shown that Vicuna is free of timing anomalies [93]. The reduced overhead and inherent efficiency of the vector processing paradigm eliminates the need for optimizations that would undermine its timing-predictability. Vicuna's performance matches that of other vector architectures while still providing full timing-predictability and freedom from timing anomalies.

5.2 RISC-V Vector Extension

This section gives an overview of the RISC-V ISA as well as its vector processing extension.

The RISC-V instruction set is an extensible ISA that is developed by the RISC-V foundation. The ISA specifies a base instruction set with limited functionality, which has to be supported by every conformant processor and a growing number of extensions that can optionally be implemented. As such, the RISC-V instruction set aims to be a flexible replacement for various existing instruction sets. Depending on their application-specific needs, architects can decide which of the extensions they choose to support in their implementation, from simple embedded systems requiring only the base set to complex high-performance architectures implementing several extensions.

The RISC-V V extension introduces vector processing to the RISC-V instruction set. While the base instruction set, as well as some extensions, have already been ratified, the V extension is currently undergoing a review process and is expected to be ratified soon.

The RISC-V V extension adds 32 vector registers to the ISA and defines vector instructions to manipulate these vector registers. While the bit-width of the vector registers is an implementation constant, the bit-width of the individual elements in a vector register can be configured at runtime. Elements are stored contiguously in a vector register, and thus the number of elements that a vector register can hold depends on the current element width. Up to 8 vector registers can be combined into a register group allowing a single vector instruction to operate on several vector registers. The vector instructions broadly fall into four categories:

1. Vector memory instructions are load and store instructions that move data between the vector registers and memory.
2. Vector arithmetic instructions are parallelized analogs of regular arithmetic and logic instructions that apply the same operation to the individual elements of vector registers. These are further divided into integer, fixed-point, and floating-point vector instructions.
3. Vector reduction instructions use an arithmetic or logic operation to reduce all elements in a vector register to a single value (such as producing the sum of all elements or returning the maximum value).
4. Vector permutation instructions rearrange the positions of elements within a vector register. These include slide instructions that move all elements up or down a vector register as well as general index-based permutation.

The V extension also allows masking individual elements for almost all vector operations. For this purpose, vector instructions have a dedicated mask bit in the instruction word, which controls whether the respective operation is masked or unmasked. Masked vector instructions are a form of predication allowing to conditionally enable or disable

the instruction for each element in the vector using an element mask. These masks are stored in regular vector registers and can be produced, for instance, by the vector comparison instructions. Additionally, the V extension defines a set of instructions to handle these vector masks.

5.3 Architecture

This section describes the architecture of the timing-predictable RISC-V vector coprocessor Vicuna.

Vicuna is a coprocessor that extends a RISC-V main core with support for the V extension instructions. The 2-stage 32-bit RISC-V core Ibex [110] is used as the main core. Ibex executes the instructions of the RISC-V base set and a few other non-vector instructions that are not part of the base set (e.g., multiplication and division), which are collectively referred to as scalar instructions, and forwards vector instructions to Vicuna via a coprocessor interface.

Vicuna features a vector instruction decoder, an instruction queue for decoded vector instructions, a dispatcher, and several specialized functional units which execute the vector instructions. Fig. 5.2 shows an overview of Vicuna's structure and its integration with the main core Ibex. Vicuna and the main core share a common data cache to ensure data consistency, while the main core fetches instructions from a separate instruction cache. A memory arbitration strategy that extends the concept developed for the timing-predictable processor SIC is used [52] in order to guarantee that all memory accesses are performed in program order. The data cache takes precedence over the instruction cache on the memory bus, and the main core's simple 2-stage pipeline guarantees that an instruction cache miss can never delay the memory access of a previous instruction. Similarly, Vicuna is given precedence in case it accesses the data cache at the same time as Ibex. Additionally, all memory transactions that would result from a cache miss of the main core are inhibited while there is a pending vector load or store instruction somewhere in the pipeline (i.e., either in the vector instruction queue or in the vector load and store unit).

Vicuna features the following specialized functional units, which each handle a subset of the vector instructions:

- A *Vector Load and Store Unit* (VLSU) interfaces the memory system and handles all vector memory instructions.
- A *Vector Arithmetic and Logical Unit* (VALU) executes the element-wise vector arithmetic instruction (except for multiplications).
- A dedicated *Vector Multiplier* (VMUL) handles vector multiplication.
- A *Vector Slide Unit* (VSLDU) implements the slide operations, which move elements up or down a vector register.

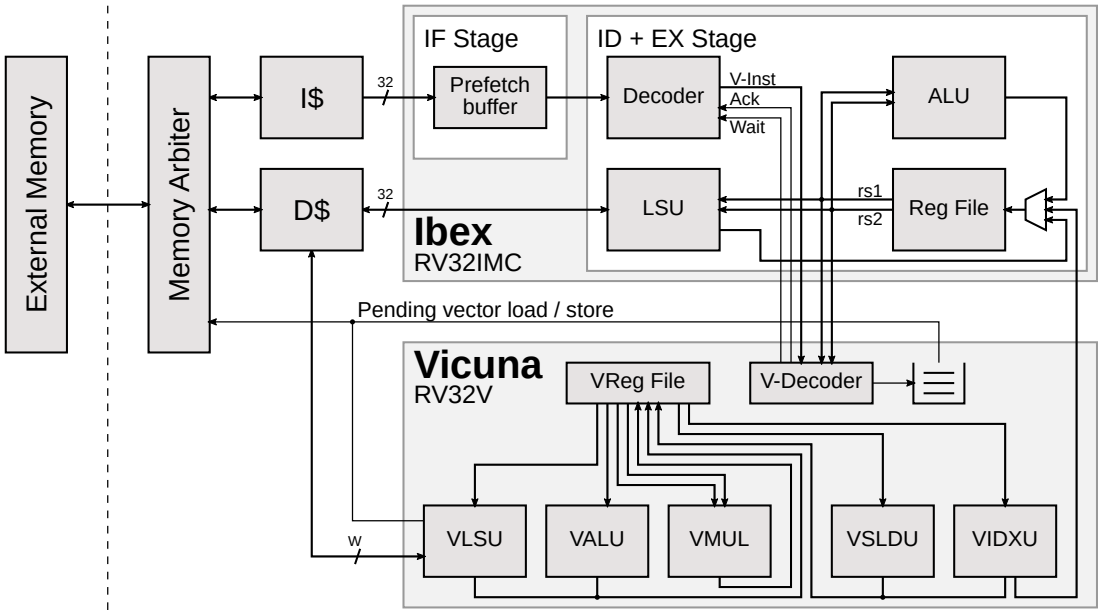


Figure 5.2: Overview of Vicuna's architecture and its integration with the main core Ibex. Both cores share a common data cache. To guarantee in-order memory access, the memory arbiter delays any access following a cache miss by the main core until pending vector load and store operations are complete. When accessing the data cache, the vector core always takes precedence.

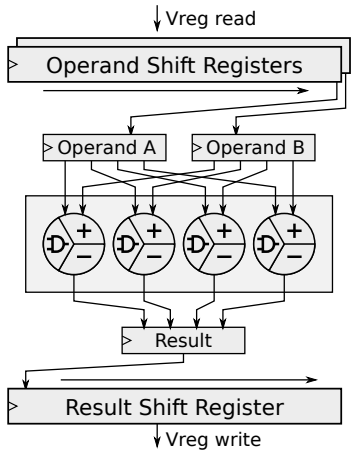
- A *Vector Indexing Unit* (VIXDU) is used for vector permutation (except for slides) and reduction instructions. In addition, only the VIXDU can write back scalar results to the main core's registers.

These units are capable of operating in parallel, thus allowing the concurrent execution of several vector instructions. The width of the datapath can be configured individually for each unit which allows the processing of several vector elements each cycle, except for the VIXDU, which can only process one element per cycle due to the irregular access patterns of some of the instructions it implements. In addition, this allows adapting the throughput of each unit based on how frequently it is used, increasing throughput for heavily-used instructions while saving resources by scaling down less used functional units. This approach contrasts with the more common lane-based vector processors, which replicate all computational resources across each lane and thus do not allow to configure the throughput of individual functionality.

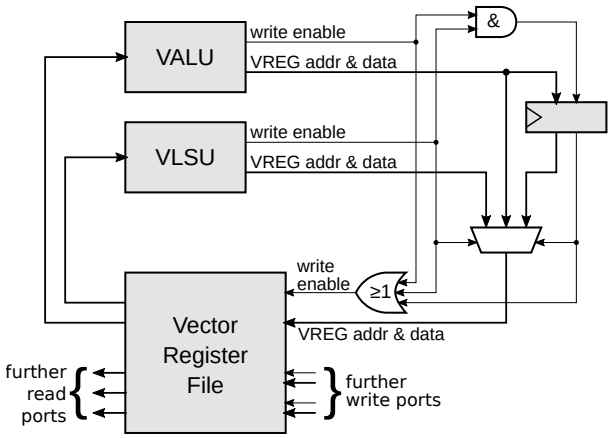
The VALU and VMUL both implement element-wise operations on a fixed-width datapath. Therefore, the number of elements that are processed simultaneously depends on the current width of the individual elements. Similar to other vector processors, Vicuna uses fracturable adders and multipliers [22] to perform mixed-width operations on a fixed-width vector register portion. A fracturable adder is a series of 8-bit adders

whose carry chains are cascaded to allow wider operations. Hence, four 8-bit, two 16-bit, or one 32-bit addition or subtraction are executed on four cascaded 8-bit adders with the carry chains between the adders open or closed depending on the element width. A similar approach is used for fracturable multipliers, which perform 8-bit, 16-bit, or 32-bit multiplications using four 16-bit multipliers.

Vicuna uses a centralized vector register file rather than distributing vector registers across lanes as done in lane-based vector processors. Distributed register files are well-suited for element-wise operations but require additional logic to exchange register data between lanes for widening or narrowing arithmetic instructions, for reduction, and for permutation instructions. In order to avoid the substantial consumption of logic resources required for selecting large sub-words from large vector registers, the functional units read entire vector registers into shift registers. The content of these shift registers is then shifted by the number of bits that are simultaneously processed by the unit each cycle, thus feeding contiguous portions of the vector register to the unit. Similarly, the results of computations are accumulated into another shift register and then written back to the vector register file at once. The concept is depicted in Fig. 5.3a, which shows the data flow within the VALU.



(a) Organization of the vector ALU. Operand registers are read sequentially into shift registers and consumed over several cycles by processing a fixed-width portion each cycle. Results are again accumulated into a shift register before write-back.



(b) The VALU and VLSU share a common write port, with the VLSU always taking precedence. In case of a collision, the value and address of the VALU write request are temporarily saved and written to the vector register file in the next cycle. Neither unit can write for two subsequent cycles. Hence the delayed write always succeeds.

Figure 5.3: Reading and writing whole registers from the vector register file avoids subword selection logic and allows multiplexing of read and write ports without affecting timing predictability.

Vicuna's units are capable of operating in parallel. Hence, the vector register file requires several read and write ports to supply the units and consume their results. However, since vector registers are read at once and then fed to the unit over several cycles and results are first accumulated before being written back to the destination register, the read and write ports can both use multiplexing. Each unit has a dedicated read port, which it uses to fetch the individual operand registers sequentially. The VMUL is the only unit that has two read ports in order to better support the fused multiply-add instructions, which use three operands. The write-ports are shared between units and use a special circuitry shown in Fig. 5.3b to handle collisions without introducing timing anomalies. Since results are aggregated before write-back, a functional unit cannot write in two subsequent cycles. Therefore, in case of a collision, one unit takes precedence and writes its result to the vector register file. The result of the second unit is written to a temporary buffer from where it is then written to the register file in the next cycle. The second write from the temporary buffer is guaranteed to succeed since neither the first nor the second unit can produce another result within just one cycle.

5.4 Timing Predictability

This section discusses the requirements for efficient timing analysis and shows that Vicuna meets these requirements.

Vicuna avoids many optimizations commonly found in modern processing architectures, which are known to increase timing variability and thus complicate the modeling and analysis of the timing behavior. This section analyzes Vicuna's timing-predictability and shows that it is free of timing anomalies, an essential property to enable efficient compositional timing analysis.

Vicuna has been carefully designed to avoid any inter-dependencies between its execution units which could cause variable execution times. In particular, the multiplexing technique used for reading and writing to and from the vector register file, as described in Sect. 5.3, avoids stalls even in case of collisions. Once an instruction has started executing on one of Vicuna's functional units, it completes within a fixed number of cycles which depends on the type of instruction, the throughput of the respective execution unit, the number of vector registers that are part of a register group, and in case of the VLSU on the bandwidth and access latency of the memory bus.

From recent literature we learn that even simple architectures which avoid complex optimizations have been shown to exhibit timing anomalies [54]. There are two kinds of timing anomalies: counterintuitive timing anomalies, which affect the timing-predictability of a processing system, and amplification timing anomalies, which threaten the timing-compositionality [64]. Both timing-predictability and timing-compositionality are essential properties to prevent the need for an exhaustive exploration of all possible system states during WCET analysis. Timing-compositionality, in particular, is required to allow decomposing the timing analysis and deriving a global worst-case from local worst-case behavior [53].

Counterintuitive timing anomalies occur whenever a better local case leads to a global worst-case behavior, such as, for instance, a cache hit causing a longer overall execution time than a miss. This kind of timing anomaly can occur when runtime decisions are involved in selecting a functional unit [127] or when instructions with multi-cycle latencies are delayed by the concurrent execution of later instructions [9]. Vicuna avoids this type of timing anomalies by guaranteeing deterministic execution times for all instructions.

Amplification timing anomalies occur when cascading effects of a local worst-case behavior cause an even larger increase in overall execution time. This type of anomaly can be very subtle to discover and affect even simple in-order pipelines with just a few stages. In particular, the re-ordering of memory accesses on the memory bus has recently been shown to be a source of timing anomalies [54]. Amplification timing anomalies can occur if the timing behavior of a processing platform is not monotonic w.r.t. the progress order of instructions [52], i.e., if an instruction can be delayed by a subsequent instruction.

Hahn and Reineke have introduced a formalism to prove the monotonicity of the timing behavior and thus freedom of timing anomalies of an execution pipeline [52] and applied that formalism to their timing-predictable core SIC. This formalism is extended to prove that Vicuna is equally free of amplification timing anomalies. Given a program with a fixed sequence of instructions $\mathcal{I} = \{i_0, i_1, i_2, \dots\}$, the pipeline state during the execution of that program maps each instruction to its current progress. The progress $\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$ of an instruction is defined by its current pipeline stage $s \in \mathcal{S}$ and the number $n \in \mathbb{N}_0$ of remaining cycles in that stage. Due to the deterministic nature of Vicuna's execution units, the combined pipeline of the main core Ibex and Vicuna is modeled with the following stages:

$$\mathcal{S} = \{\mathit{pre}, \mathit{IF}, \mathit{ID{+}EX}, \mathit{VQ}, \mathit{VEU}, \mathit{post}_S, \mathit{post}_V\}$$

The abstract stages *pre* and *post* model instructions that have not yet started execution or have already left the pipeline, respectively, analogous to the model used for SIC. However, scalar and vector instructions are differentiated in the *post* stage, which are represented by post_S and post_V, respectively. *IF* is the main core's fetch stage, and *ID+EX* is its combined decode and execute stage. The stage *VQ* models instructions that are currently in Vicuna's instruction queue, and the abstract stage *VEU* represents vector instructions currently executing on one of Vicuna's execution units. The vector instruction queue retains the ordering of instructions, and the execution time of vector instructions is fully deterministic on all vector execution units. Therefore, modeling each of the concrete stages is not required.

The ordering $\sqsubset_{\mathcal{S}}$ of these pipeline stages is as follows:

$$\mathit{pre} \sqsubset_{\mathcal{S}} \mathit{IF} \sqsubset_{\mathcal{S}} \mathit{ID{+}EX} \begin{array}{l} \sqsubset_{\mathcal{S}} \mathit{post}_S \\ \sqsubset_{\mathcal{S}} \mathit{VQ} \sqsubset_{\mathcal{S}} \mathit{VEU} \sqsubset_{\mathcal{S}} \mathit{post}_V \end{array}$$

Both scalar and vector instructions first enter the pipeline of Ibex, which fetches and decodes the instructions. The *ID+EX* stage executes scalar instructions right away and forwards vector instructions to Vicuna's decoder, which decodes and validates vector instructions. Scalar instructions exit the pipeline and move to the $post_S$ stage after that, while vector instructions enter the vector instruction queue VQ, are subsequently executed by one of Vicuna's execution units, and finally, move to the $post_V$ stage.

The execution time of vector instructions on Vicuna's functional units is fully deterministic, with the exception of load and store instructions which might stall in case of a cache miss. The memory arbiter holds back any memory accesses by the main core if any vector loads or stores are pending. Therefore, vector loads or stores are not delayed by other instructions that might cause memory accesses by the main core. The vector queue retains the ordering of vector instructions, and the first vector instruction is dispatched to its respective execution unit as soon as that unit becomes available, and potential data hazards have been resolved. Therefore, instructions in the vector queue can only be delayed by prior instructions but never by later instructions. Similarly, instructions in the *ID+EX* stage can be stalled by an ongoing memory access of the vector core, during memory loads and stores, by a vector instruction writing back to a scalar register, or when a vector instruction has been decoded, but the vector queue is full. In any case, an instruction in this stage can only be delayed by earlier instructions. Finally, the main core's instruction stage is stalled in case the subsequent *ID+EX* stage stalls or in case of an instruction cache miss, which can further be delayed by ongoing data loads or stores. Hence, the progress order of instructions is maintained throughout the execution platform, and instructions in any stage can only be stalled by prior instructions. Therefore, the execution time of instructions is monotonic w.r.t. the progress order, and thus amplification timing anomalies are avoided.

Vicuna is free of both counterintuitive and amplification timing anomalies. Therefore, it is timing-predictable and timing-compositional, enabling fast and efficient WCET analysis.

5.5 Performance Evaluation

This section presents an assessment of Vicuna's performance and scalability by measuring the execution time of three simple Basic Linear Algebra Subroutine (BLAS) subroutines with varying degrees of arithmetic intensity on different configurations of Vicuna synthesized on a Xilinx 7 Series FPGA and comparing the results with other vector processing architectures.

Three configurations of Vicuna are evaluated, with vector register lengths of 128, 512, and 2048 bits, respectively, with the parameters of each configuration as well as the peak multiplier performance and maximum clock rate listed in Table 5.2.

The performance effectively achieved by an application on a parallel processor architecture is frequently degraded by several bottlenecks, which can render the efficient

Table 5.2: Configurations of Vicuna for evaluation on a Xilinx 7 Series FPGA. Note that for larger configurations, the maximum clock frequency decreases slightly as these require more resources which complicates the routing process.

Config. Name	Configuration Parameters			8-bit MACs per cycle	Clock frequency (MHz)
	Vector Reg. Width (bit)	Multiplier Data-Path Width (bit)	Data-Cache Size (kB)		
Small	128	32	8	4	100
Medium	512	128	64	16	90
Large	2048	1024	128	128	80

utilization of parallel computing resources challenging. Vicuna has been evaluated on an FPGA platform with a 32-bit memory interface which eventually limits the performance gains achieved by increasing the vector register and datapaths of functional units. The roofline model visualizes the peak performance in operations per cycle in function of the arithmetic intensity of an application. The arithmetic intensity is the ratio of the number of operations of an application per byte of memory transfer. An application is either compute-bound if its performance is limited by available computation performance, or memory-bound if overall performance is limited by the memory bandwidth instead.

The roofline performance boundary for each configuration of Vicuna is shown in Fig. 5.4. The dashed lines are the theoretical performance boundaries for each configuration, with the horizontal part corresponding to the compute-bound region, where the peak performance equals the maximum arithmetic throughput, and the diagonal part showing the memory-bound region, where the memory bandwidth limits the achievable performance. The plot also shows the measured performance for three benchmarks: weighted vector addition, 3×3 image convolution, and matrix multiplication. The percentages next to the markers indicate the ratio of effective vs. theoretical performance.

The first benchmark, AXPY, is a weighted vector addition defined as $Y \leftarrow \alpha X + Y$, where X and Y are two vectors, and α is a scalar. It has been implemented for vectors of 8-bit elements, which requires n 8-bit multiply-accumulate (MAC) operations and $3n$ bytes of memory transfer for a vector with n elements. Hence, the arithmetic intensity for this benchmark is $1/3$, which places it in the memory-bound region for all three configurations of Vicuna.

The second benchmark is 3×3 image convolution, which loads an input image and applies a 3×3 convolution kernel. This requires 9 MACs per pixel and two memory transactions for loading and then storing each pixel, which yields an arithmetic intensity of 4.5 for 8-bit pixel values.

Finally, the third benchmark is the generalized matrix multiplication (GEMM) which is defined as $C \leftarrow AB + C$ where A, B, and C are matrices. The arithmetic intensity of

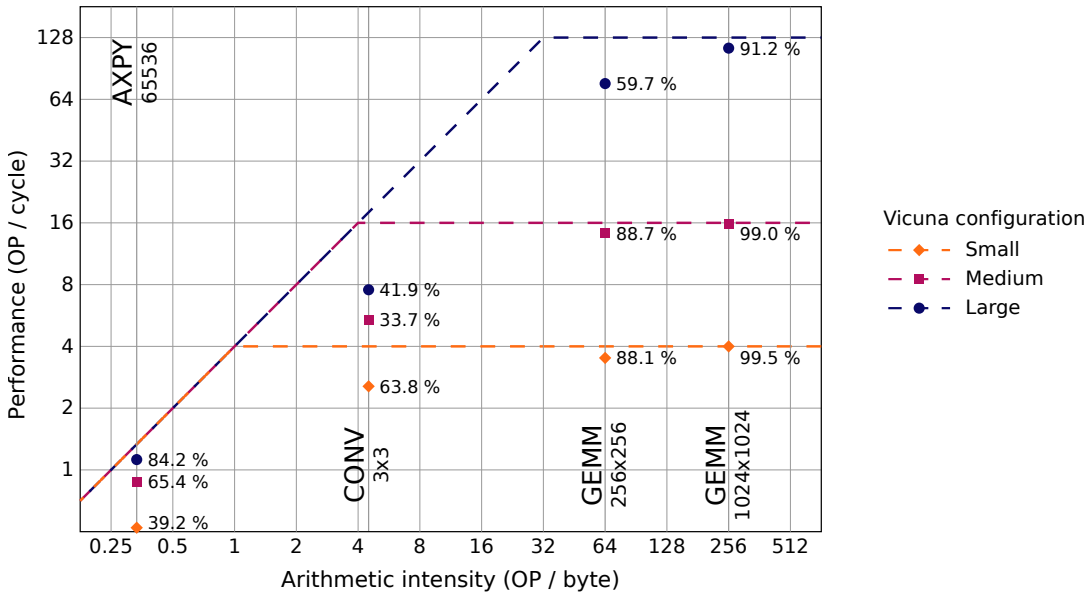


Figure 5.4: Roofline plot of the performance results for the benchmark algorithms for each of Vicuna's three configurations listed in Table 5.2. The dashed lines are the performance boundaries of each configuration, and the markers show the measured effective performance. The percentages indicate the ratio of effective vs. theoretical performance.

this benchmark depends on the matrix size, with n^3 MAC operations and $4n^2$ memory transactions required for $n \times n$ matrices. Using 8-bit values this gives an arithmetic intensity of $n/4$. Hence, for large matrices, this is a heavily compute-bound benchmark. For this evaluation, matrix sizes of 256×256 and 1024×1024 are used, which give an arithmetic intensity of 64 and 256, respectively.

The results show that the performance of Vicuna scales well across the three configurations, achieving over 90 % efficiency for compute-bound tasks, which is in line with other high-performance vector processors.

Vicuna's resource footprint is similar to other FPGA-based vector processors. Fig. 5.5 compares the resource utilization and performance of Vicuna with that of VESPA [133] and VEGAS [22]. The radar chart depicts the lookup table, flip-flop, DSP block, and RAM utilization of each of these architectures, along with the achieved clock period and the efficiency in terms of multiplier utilization for compute-bound workloads. Vicuna consumes a similar amount of logic resources as the other two processors. However, its clock period is larger, which stems from the latency of the ports of its vector register file. VESPA requires fewer register file ports since it is only capable of executing one instruction at a time, and VEGAS equally uses fewer ports into a scratchpad memory that replaces the conventional register file. Despite the reduced clock frequency, the overall performance of Vicuna is still higher due to its ability to better utilize computational

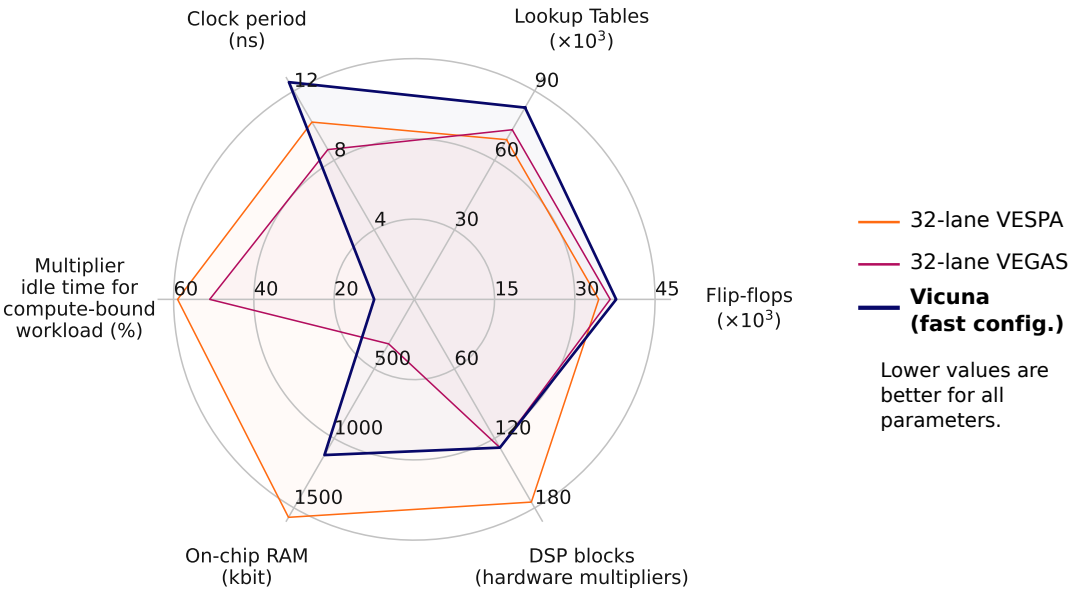


Figure 5.5: Resource utilization and performance of the FPGA-based vector processors Vicuna, VESPA, and VEGAS (each configured for a peak performance of 128 8-bit operations per cycle).

resources by executing several instructions concurrently. VEGAS, for instance, executes a 4096×4096 matrix multiplication within 4.377 billion cycles on a 32-lane configuration, which translates to a multiplier utilization of only 49 %. The large configuration of Vicuna, which has the same peak performance per cycle, reaches a multiplier utilization of over 90 % for similar compute-bound workloads.

Vicuna's efficiency is in line with more recent vector processing architectures targeting ASICs instead of FPGAs, such as Hwacha [75] and Ara [19], which equally reach an efficiency of over 90 % for compute-bound tasks. Ara has been reported to reach close to 98 % multiplier utilization for a 256×256 matrix multiplication on a configuration with 16 64-bit lanes. However, while these two processors are capable of efficiently exploiting data-level parallelism, they use speed-up mechanisms that can be a source of timing anomalies. Ara uses a banked register file that allows concurrent reads and writes for vector registers located in different banks but resolves banking conflicts dynamically using a weighted round-robin arbitration scheme that prioritizes arithmetic operations over memory operations. Therefore, slow memory operations can further be delayed by later arithmetic instructions, which violates the principle of monotonicity in the instruction timings [55]. Additionally, the progress of instructions can be affected by runtime decisions. Thus, Ara is most likely affected by both counterintuitive and amplification timing anomalies [127]. Hwacha also uses a banked register file, but sequences accesses

of vector register elements in a way that avoids banking conflicts. However, due to its out-of-order write-back mechanism, Hwacha also exhibits timing anomalies. Moreover, none of the mentioned vector architectures maintain program order for memory accesses. In particular, memory accesses of earlier vector instructions can be delayed by concurrent memory operations of later scalar instructions, which is likely to cause amplification timing anomalies on all of these platforms [55].

Vicuna is different from other vector processors due to its timing-predictability and freedom of timing anomalies, which enable the efficient timing analysis required for real-time applications. However, Vicuna's performance is close to that of other vector processors, which are affected by timing anomalies and thus unsuitable for such systems. Vicuna's performance for data-parallel workloads scales significantly better than that of timing-predictable multi-core architectures, for which the memory interconnect becomes a limiting factor [82, 119], especially in a real-time system requiring guarantees w.r.t. the latency and bandwidth of each core [67, 95]. As a result, the performance of the timing-predictable multi-core platforms, such as the T-CREST or the parMERASA multi-core architectures, only scales logarithmically in the number of cores [113, 43].

Finally, the adoption of Vicuna is eased by using a standardized ISA instead of a custom extension. With its combined efficiency, scalability, timing-predictability, and standard compliance, Vicuna is ideally suited for data-parallel real-time applications.

5.6 Application Benchmarks

So far, the performance of Vicuna has been evaluated using a small set of linear algebra kernels. This section presents a more thorough analysis based on three relevant real-world algorithms from the domains of signal processing, image processing, and cryptography, as well as a comparison with a timing-predictable multi-core platform.

The primary purpose of these benchmarks is to evaluate the performance gains achieved by increasing Vicuna's computational resources and comparing these with the performance gains achieved by increasing the number of cores in a timing-predictable multi-core architecture. The timing-predictable multi-core platform T-CREST is used [113] for this comparison. This architecture connects the individual cores to the memory with a tree-shaped time-division multiplexing (TDM) memory arbiter, which guarantees repeatable and predictable access times for each core.

Besides demonstrating Vicuna's performance and scalability for several different workloads, the choice of applications from different areas also serves to highlight its versatility, which is a crucial advantage of general-purpose vector processing architectures over domain-specific accelerators.

The following three algorithms have been implemented and evaluated on Vicuna and T-CREST:

1. The discrete Fourier transform (DFT), implemented using the fast Fourier Transform (FFT) method proposed by Cooley and Tukey [23].
2. Image registration, based on the algorithm proposed by Lucas and Kanade [79].
3. The Advanced Encryption Standard (AES) [25].

The DFT is an important signal processing algorithm that is used to perform the Fourier transform for subsequent analysis of the frequency spectrum in many signal processing applications. Due to its importance, the DFT is frequently implemented in hardware [20] or accelerated using dedicated hardware functionality [83]. A parallel version of the FFT proposed by Cooley and Tukey [23] is implemented with RISC-V vector instructions, as well as a multi-core variant for use on T-CREST.

Image registration is the process of finding a vector that translates an image such that the difference to a second image becomes minimal. It is an important image processing technique used, for instance, to estimate the optical flow between two consecutive frames of an image sequence or to match image regions in stereo vision. Similar to the DFT, the importance of this algorithm has equally led to the development of dedicated hardware accelerators [27, 118], some of which are specifically targeting safety-critical applications [21]. The popular image registration algorithm proposed by Lucas and Kanade [79] has been implemented using RISC-V vector instructions for Vicuna and on the multi-core T-CREST platform.

The block cipher Rijndael [25] was selected as the AES in 2001 and is since widely used for data encryption. Security and privacy considerations with Internet of Things (IoT) devices and edge computing are fostering the use of AES encryption in embedded systems [44]. However, the computational performance required by AES often prohibits the use of software implementations on low-power devices. Therefore, several specialized AES hardware accelerators for use in embedded systems have been proposed [87, 86]. A vector processor can substitute the need for a dedicated accelerator and can be used for other tasks as well, thus potentially reducing the resource footprint of data processing architectures that require encryption and decryption capabilities in addition to their primary function. AES encryption has been implemented in RISC-V vector instructions as well as a multi-core variant on T-CREST.

Although Vicuna and the T-CREST platform are very different architectures, it has been attempted to match the resources available to each as closely as possible. Both Vicuna and the T-CREST platform use a 32-bit main memory interface, which eventually limits the achievable performance increase. For Vicuna, the duration of loads and stores increases as the vector length increases since only 32 bits of data can be fetched from the main memory each cycle. For T-CREST, the duration of loads and stores accessing main memory also increases since its memory arbiter has fewer time-slots to give to each core as the number of cores in the system increases. The data caches of the T-CREST cores have been found to use a write-through policy, which causes unnecessary writes to main memory for intermediate data. Therefore, the implementation loads all data into the

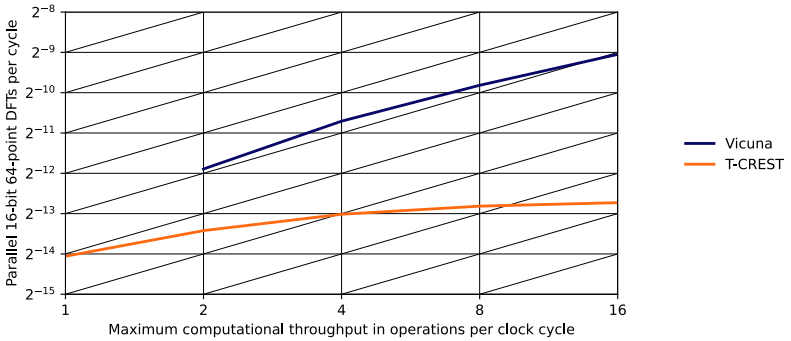
core-local scratchpad memory first, thus allowing each core to process that data without requiring intermediate accesses on the memory bus.

Each core of the T-CREST platform is capable of executing one operation per cycle. Hence, the peak performance of the multi-core system in operations per cycle is equal to the core count. The benchmark applications have been evaluated on configurations with 1, 2, 4, 8, and 16 cores. For Vicuna, the number of operations per cycle depends on the datapath width of its functional units and on the bit-width of the individual vector elements. Configurations with datapath widths of 32, 64, 128, and 256 bits are used. The implementations of the FFT and the image registration algorithms both use 16-bit fixed-point values, which yields a peak performance from 2 to 16 operations per cycle for these four configurations. The AES algorithm operates on individual bytes. Thus the peak performance of the Vicuna configurations varies between 4 and 32 operations per cycle.

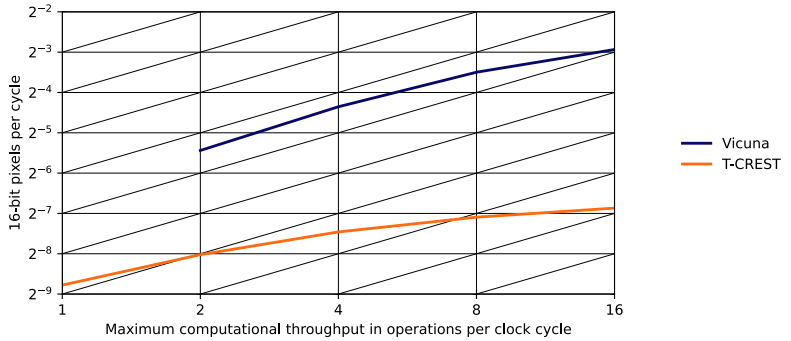
Fig. 5.6 presents the results of the evaluation. The three plots show the performance for each of the three algorithms w.r.t. the available computational resources achieved on Vicuna in blue and the T-CREST platform in orange. The theoretical peak performance on the x-axis corresponds to the maximum throughput of the arithmetic units for Vicuna and the number of cores for T-CREST. An application-specific performance metric is used on the y-axis to quantify the effective performance in function of theoretical peak performance for the respective algorithm. The diagonal grid lines correspond to linear growth (i.e., a doubling in effective performance for a doubling in computational resources). The results show that Vicuna's effective performance scales better than that of the T-CREST platform as the computational throughput is increased. While the performance gains eventually drop below linear growth for both architectures, Vicuna is capable of sustaining linear or close-to-linear gains for a larger computational throughput than T-CREST.

In a multi-core system, each core separately needs to fetch the instructions of the algorithm, a penalty which is not incurred for Vicuna where the instructions are fetched by the main core only. Additionally, the code executed by Vicuna consists of fewer instructions overall since the ability to have instructions operate on several values at once allows for a reduction in code size. Therefore, Vicuna performs better than T-CREST for the FFT and the image registration benchmarks.

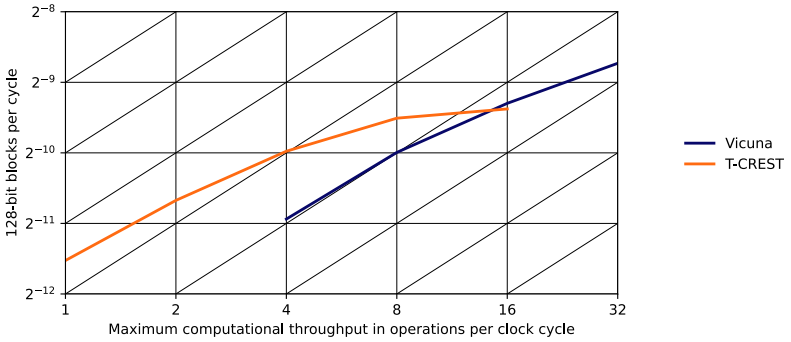
The T-CREST platform is at an advantage for the AES benchmark since the byte permutations required as part of the algorithm are free when combined with other steps (by storing the individual bytes at different memory offsets than they were loaded from). For Vicuna, the byte permutations have been implemented with a sequence of masked vector slide instructions. Further, the AES algorithm involves a lookup into a 256-entry lookup table, which has been implemented with an indexed load instruction on Vicuna. The indexed load cannot be parallelized since the elements to be fetched are not stored contiguously, thus requiring a separate load request for each element. Yet, thanks to its superior scalability, Vicuna eventually surpasses the performance of the T-CREST platform for a configuration with a peak performance of 16 operations per cycle.



(a) Effective vs. theoretical performance for a parallel variant of Cooley and Tukey's FFT algorithm [23]



(b) Effective vs. theoretical performance for Lucas and Kanade's image registration [79]



(c) Effective vs. theoretical performance of the 128-bit AES algorithm [25]

Figure 5.6: Performance growth comparison between Vicuna in blue and the T-CREST multi-core platform in orange based on three real-world applications. As the theoretical peak performance rises by increasing the throughput of units for Vicuna and adding more cores for T-CREST, the effective performance gain is much higher for Vicuna than for the T-CREST platform. The diagonal grid lines correspond to linear growth.

While raising the theoretical peak performance by increasing the throughput of the functional units on Vicuna and adding additional cores on the T-CREST platform leads to improved performance on both architectures, the performance gains are lower for T-CREST. The results confirm the observation by Schoeberl et al. that the performance of the T-CREST platform scales only logarithmically in the number of cores [113]. By contrast, the performance of Vicuna scales better, as has already been demonstrated in Sect. 5.5, which is in line with the performance advantages of non-timing-predictable vector processors over non-timing-predictable multi-core architectures [19].

Dedicated hardware accelerators have been developed for each of the three applications used in this analysis. While accelerators usually deliver significant performance gains for specific tasks, the hardware cannot be re-used for other purposes. Thus, if a processing system needs increased performance for several such algorithms, then a separate accelerator is required for each of them. Vicuna is a general-purpose vector processor and, as such, is capable of parallelizing any algorithm that can run on a conventional processor. Hence, all three applications can be executed on Vicuna, thus re-using the same computational resources for different tasks. While domain-specific accelerators typically are timing-predictable, they lack the versatility of a general-purpose processing platform.

Vicuna can be used as a general-purpose parallel processing platform for a variety of time-critical applications. It is more scalable and efficient for highly parallel workloads than multi-core architectures and more versatile than domain-specific accelerators. Vicuna fuses the benefits of vector processors and timing-predictable platforms into a processing architecture with the following unique characteristics:

- The scalable parallelism of vector processors, in contrast to the limited parallel performance of multi-core or array processors.
- The flexibility of a general-purpose computing platform, as opposed to domain-specific accelerators.
- The timing-predictability required in real-time applications.

5.7 Findings

Most modern processor architectures improve average performance by making use of micro-architectural optimizations, which complicate the analysis of their timing behavior. Processors used in real-time systems must avoid these features such that the execution times of tasks can be analyzed. Thus, real-time architectures generally suffer from drastically reduced computation power and are not able to compete with high-performance architectures. Yet, the requirements of many safety-critical applications force system architects to select less safe non-real-time processing platforms for performance reasons.

Vicuna is the first timing-predictable vector processing architecture. Vector processors are a promising architecture for real-time systems that need to handle the performance

requirements of emerging highly parallel applications. The timing behavior of vector processors is simpler to analyze than that of complex heterogeneous systems. Vicuna has been shown to be free of timing anomalies and thus enables compositional timing analysis.

Despite its predictability, the performance of Vicuna is close to that of other vector processors that are not timing-predictable. Thanks to the vector processing paradigm's inherent efficiency, optimizations that affect predictability can be avoided without a significant impact on performance. Vicuna implements the recently finalized vector extension of the RISC-V ISA and is ideally suited for the massively data-parallel workloads in many emerging time-critical applications in domains such as autonomous driving.

Additionally, vector processors support fully predicated execution via masked vector instructions. This is a form of predication that allows compilers to enable or disable the instruction for each individual element of a vector. As such, vector processors are capable of efficiently executing single-path variants of vectorized code. In contrast to the single-path filter discussed in Chap. 4, vector processors have native support for fully predicated execution and do not require such an extension.

Consequently, vector processors appear to be ideally suited for demanding data-parallel real-time applications for two reasons: The processing hardware has inherently better timing-predictability, and the native support for fully predicated execution means that the vectorized code is equally inherently more predictable.

CHAPTER 6

Conclusion

The design of hard real-time systems requires that the WCET of tasks is determined in order to establish guarantees w.r.t. the response time of the system and hence its correctness. The static analysis methods widely used for this purpose require an accurate timing model of the execution platform, which allows an analysis tool to determine the execution time of the various possible execution paths. However, optimization features in modern processor architectures reduce the predictability to the point where a WCET analysis may not be possible at all. Even for processors designed to be timing-predictable, actually performing this analysis on production code requires significant manual annotations, and the WCET bounds determined by the analysis tools are often over-estimated.

The combination of stringent timing requirements, a constrained choice of suitable execution platforms that is further reduced by limited tool support, analysis results that are severely impacted by the accuracy of the timing model and the quality of code annotations, as well as the poor predictability of performance-oriented processor architectures currently confines hard real-time systems to the few architectures especially designed for this purpose. This has led to a significant performance gap between processors suited for time-critical tasks and those attempting to maximize performance. Yet, the performance requirements of modern hard real-time applications can no longer be met with the existing timing-predictable architectures.

This thesis attempts to identify possible ways to bridge the performance gap by increasing the pool of potential candidate architectures that can be used for real-time systems. The main idea is to leverage predicated execution in order to eliminate data-dependent control-flow branches, which dramatically simplifies timing analysis. While Chap. 3 demonstrates the use of single-path code in real-time systems, Chap. 4 proposes a processor core extension that adds support for predicated execution and can conceptually be integrated into any existing processor.

The concept of predicated execution can be parallelized by vector processors, an inherently data-parallel architecture that is currently gaining popularity due to its energy-efficiency. Incidentally, processor architectures that have better energy-efficiency also tend to be more timing-predictable [85]. The shift towards parallel and energy-efficient architectures has in recent years been fueled by the popularity of machine-learning, a trend that might also benefit real-time systems since the micro-architectural optimizations used in most processors that hinder predictability are becoming less attractive due to their poor energy-efficiency. Therefore, it is likely that future computer architectures see less of these optimizations and thus an improved timing-predictability. Chap. 5 exemplifies this trend by presenting a timing-predictable vector processor that retains the performance, scalability, and efficiency of other vector processors that have not been designed with real-time systems in mind.

While favoring parallel and energy-efficient hardware, the data-parallel nature of most machine-learning applications also appears to favor a more data-oriented programming style rather than the pre-dominant imperative programming style. This is reflected in a major re-thinking of traditional compiler technologies and the introduction of new compiler infrastructure, such as the MLIR project [73], that embraces a new and much more flexible way of writing programs than traditional imperative programming. This new data-flow-oriented approach to compiler technology is intended to facilitate the mapping of complex parallel tasks to an ever-growing number of parallel architectures [125]. As a side effect, support for predicated execution is becoming an integral part of the compilation toolchain.

With toolchain and hardware support for predicated execution and data-parallel processing on the rise, real-time system architects can take advantage of these new developments to reduce the performance gap between timing-predictable and mainline processors. The single-path filter and the timing-predictable vector processor presented in this thesis show that improved predictability does not necessarily jeopardize performance and that real-time systems do not need to be confined to a few purpose-built processors.

Based on these findings, the research questions raised in Chap. 1 can be answered as follows:

1. It is indeed possible to improve the predictability of software by leveraging predicated execution to eliminate data-dependent control-flow branches without impacting its performance and without severely limiting the available processing platforms, as has been demonstrated by the single-path filter proposed in Chap. 4, which allows to extend existing processor designs with support for predicated execution and thus allows to execute predictable code on a potentially very large range of processors. Chap. 3 has shown that single-path code is well suited for real-time control systems.

2. Further, it is possible to parallelize the concept of predicated execution by leveraging data-parallel architectures, such as vector processors, which can be made timing-predictable without affecting their performance, as has been shown in Chap. 5 by proposing and evaluating the timing-predictable vector processor Vicuna.

While the present work attempts to open up new perspectives for future real-time system architectures, a number of problems remain to be solved by future work. Chap. 4 identified some programming constructs and algorithms that should be avoided in single-path code due to their reduced efficiency and proposed better alternatives. However, a more systematic investigation of programming techniques and their efficacy in single-path code should be conducted, with the goal of deriving recommendations for programmers and ideally also compiler techniques that allow to avoid these inefficiencies. Predicated execution can be used on vector processors to avoid the need for a traditional WCET analysis by taking advantage of the properties of single-path code. Nonetheless, it would be desirable to develop a systematic approach to assess the WCET of tasks on Vicuna in particular and on vector processors in general.

Future work in the domain of real-time system architectures will certainly be shaped by the broader trends and developments taking place today, and researchers should keep a close eye on emerging compiler technologies and hardware architectures, which hopefully will allow timing-predictable processors to catch up and thus be able to meet the performance requirements of future time-critical systems.

List of Figures

List of Tables

Acronyms

AES Advanced Encryption Standard. 68

ASIC application-specific integrated circuit. 13

BLAS Basic Linear Algebra Subroutine. 63

CFG Control-Flow Graph. 32

DFT discrete Fourier transform. 68

ESC electronic speed controller. 23–26

FPGA Field-Programmable Gate Array. 22, 38, 55, 65, 66

GPU Graphics Processing Unit. 53–55

IoT Internet of Things. 68

ISA Instruction Set Architecture. 2, 4, 18, 21, 22, 29, 32, 38, 39, 41, 54–57, 67, 72

MAC multiply-accumulate. 64, 65

NoC network-on-chip. 9, 53

RISC Reduced Instruction Set Computer. 38, 39

SIMD single-instruction multiple-data. 51

STA Static Timing Analysis. 2, 4, 10–13, 17–19, 26, 28, 29

TDM time-division multiplexing. 67

WCET worst-case execution time. 2–5, 7, 9–21, 26, 28, 29, 41–46, 61, 63, 73, 75, 77

Bibliography

[1] AbsInt Angewandte Informatik GmbH: Supported targets for WCET analysis. https://www.absint.com/ait/targets.htm. Accessed: 2022-03-28.

[2] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018. ISSN 2405-8440. doi:10.1016/j.heliyon.2018.e00938.

[3] *Nios II Processor Reference Handbook*. Altera Corporation, 2009.

[4] W. Amin, F. Hussain, S. Anjum, S. Khan, N. K. Baloch, Z. Nain, and S. W. Kim. Performance evaluation of application mapping approaches for network-on-chip designs. *IEEE Access*, 8:63607–63631, 2020. doi:10.1109/ACCESS.2020.2982675.

[5] J. Andersson, J. Gaisler, and R. Weigand. Next generation multipurpose microprocessor, 2010.

[6] K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 230–237, Dec. 2013. doi:10.1109/FPT.2013.6718358.

[7] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, CA, USA, 1998.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. R. Hauser, A. M. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moretó, A. J. Ou, D. A. Patterson, B. C. Richards, C. Schmidt, S. Twigg, H. D. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr. 2016.

[9] M. Asavoae, B. B. Hedia, and M. Jan. Formal Executable Models for Automatic Detection of Timing Anomalies. In F. Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, volume 63 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-073-6. doi:10.4230/OASIcs.WCET.2018.2.

[10] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, and M. A. R. Saghir. Arrow: A RISC-V vector accelerator for machine learning inference, July 2021.

[11] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Ingredients of operating system correctness: Lessons learned in the formal verification of pikeos. 2010.

[12] S. F. Beldianu and S. G. Ziavras. Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*, 64(3):805–817, 2015. doi:10.1109/TC.2013.2295820.

[13] L. Benini and G. De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 418–419, 2002. doi:10.1109/DATE.2002.998307.

[14] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software—Practice and Experience*, 23:1249–1265, 1993.

[15] C. Berg, J. Engblom, and R. Wilhelm. Requirements for and design of a processor with predictable timing. In L. Thiele and R. Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2004/5`.

[16] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288, 2002. doi:10.1109/REAL.2002.1181582.

[17] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static wcet analysis to automotive communication software. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 249–258, 2005. doi:10.1109/ECRTS.2005.7.

[18] P. Castillo, R. Lozano, and A. E. Dzul. *Modelling and Control of Mini-Flying Machines*. Springer, 2005.

[19] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. Ara: A 1 GHz+ scalable and energy-efficient RISC-V vector processor with multi-precision floating point support in 22 nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP:1–14, Dec. 2019. doi:10.1109/TVLSI.2019.2950087.

[20] T.-S. Chang, J.-I. Guo, and C.-W. Jen. Hardware-efficient dft designs with cyclic convolution and subexpression sharing. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):886–892, 2000. doi:10.1109/82.868456.

[21] J. Cho, Y. Jung, D.-S. Kim, S. Lee, and Y. Jung. Moving object detection based on optical flow estimation and a gaussian mixture model for advanced driver assistance systems. *Sensors*, 19(14), 2019. ISSN 1424-8220. doi:10.3390/s19143217. URL `https://www.mdpi.com/1424-8220/19/14/3217`.

[22] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305549. doi:10.1145/1950413.1950420.

[23] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965. doi:10.1090/S0025-5718-1965-0178586-1.

[24] D. Dabbelt, C. Schmidt, E. Love, H. Mao, S. Karandikar, and K. Asanovic. Vector processors for energy-efficient embedded systems. In *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, MES '16, page 10–16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342629. doi:10.1145/2934495.2934497.

[25] J. Daemen and V. Rijmen. Aes proposal: Rijndael. 1999.

[26] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the spear design example. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 169–176, July 2003.

[27] J. Diaz, E. Ros, F. Pelayo, E. Ortigosa, and S. Mota. Fpga-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16 (2):274–279, 2006. doi:10.1109/TCSVT.2005.861947.

[28] B. Dinechin, D. Amstel, M. Poulhies, and G. Lager. Time-critical computing on a single-chip massively parallel processor. pages 1–6, Mar. 2014. ISBN 9783981537024. doi:10.7873/DATE.2014.110.

[29] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Jan. 2010. doi:10.1109/JPROC.2009.2034764.

[30] P. S. Duggirala and M. Viswanathan. Analyzing real time linear control systems using software verification. In *2015 IEEE Real-Time Systems Symposium*, pages 216–226, 2015. doi:10.1109/RTSS.2015.28.

[31] E. Ebeid, M. Skriver, and J. Jin. A survey on open-source flight control platforms of unmanned aerial vehicle. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 396–402, Aug 2017. doi:10.1109/DSD.2017.30.

[32] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 264–265, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936271. doi:10.1145/1278480.1278545.

[33] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *2009 IEEE International Conference on Computer Design*, pages 54–59, 2009.

[34] G. A. Elliott and J. H. Anderson. Real-world constraints of GPUs in real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 2, pages 48–54, 2011. doi:10.1109/RTCSA.2011.46.

[35] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 2012. doi:10.1007/s11241-011-9140-y.

[36] G. A. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: a framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44, 2013. doi:10.1109/RTSS.2013.12.

[37] J. Engblom and B. Jonsson. Processor pipelines and their properties for static wcet analysis. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software*, pages 334–348, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45828-9.

[38] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from Industrial WCET Analysis Case Studies. In R. Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-24-8. doi:10.4230/OASIcs.WCET.2005.811. URL `http://drops.dagstuhl.de/opus/volltexte/2007/811`.

[39] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In M. Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[40] C. Ferdinand and R. Heckmann. ait: Worst-case execution time prediction by static program analysis. In R. Jacquart, editor, *Building the Information Society*, pages 377–383, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8157-6.

[41] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972. ISSN 0018-9340. doi:10.1109/TC.1972.5009071.

[42] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *2014 IEEE Real-Time Systems Symposium*, pages 53–62, 2014. doi:10.1109/RTSS.2014.28.

[43] M. Frieb, R. Jahr, H. Ozaktas, A. Hugl, H. Regler, and T. Ungerer. A parallelization approach for hard real-time systems and its application on two industrial programs. *Int. J. Parallel Program.*, 44(6):1296–1336, Dec. 2016. ISSN 0885-7458. doi:10.1007/s10766-016-0432-7.

[44] M. Frustaci, P. Pace, G. Aloi, and G. Fortino. Evaluating critical security issues of the iot world: Present and future challenges. *IEEE Internet of Things Journal*, 5 (4):2483–2495, 2018. doi:10.1109/JIOT.2017.2767291.

[45] Y. Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016. doi:10.1613/jair.4992.

[46] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. In *John yon Neumann Collected Works, Volume V: Design of Computers, Theory of Automata and Numerical Analysis*, pages 152–214. Pergamon Press, Oxford, England, 1963.

[47] V. Golyanik, M. Nasri, and D. Stricker. Towards scheduling hard real-time image processing tasks on a single GPU. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 4382–4386, 2017. doi:10.1109/ICIP.2017.8297110.

[48] F. Guet, L. Santinelli, and J. Morio. On the Reliability of the Probabilistic Worst-Case Execution Time Estimates. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.

[49] J. Gustafsson. The worst case execution time tool challenge 2006. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 233–240, 2006.

[50] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen wcet benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 136–146, 01 2010. doi:10.4230/OASIcs.WCET.2010.136.

[51] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, page 3–14, USA, 2001. IEEE Computer Society. ISBN 0780373154.

[52] S. Hahn and J. Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481, Dec. 2018. doi:10.1109/RTSS.2018.00060.

[53] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, Mar. 2015. doi:10.1145/2752801.2752805.

[54] S. Hahn, J. Reineke, and R. Wilhelm. *Toward Compact Abstractions for Processor Pipelines*, pages 205–220. Springer International Publishing, Nov. 2015. ISBN 978-3-319-23505-9. doi:10.1007/978-3-319-23506-6_14.

[55] S. Hahn, M. Jacobs, and J. Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 299–308, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347877. doi:10.1145/2997465.2997471.

[56] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[57] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, July 1961. ISSN 0001-0782. doi:10.1145/366622.366644.

[58] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 01 1962. ISSN 0010-4620. doi:10.1093/comjnl/5.1.10.

[59] G. Hoffman, H. Huang, S. L. Waslander, and C. J. Tomlin. Quadrotor helicopter flight dynamics and control: Theory and experiment. In *Conference of the American Institute of Aeronautics and Astronautics*, Aug 2007.

[60] G. Hoffmann, D. G. Rajnarayan, S. L. Waslander, D. Dostal, J. S. Jang, and C. J. Tomlin. The Stanford testbed of autonomous rotorcraft for multi agent control (STARMAC). In *The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576)*, volume 2, pages 12.E.4–121, Oct 2004. doi:10.1109/DASC.2004.1390847.

[61] R. M. Hord. *The Illiac IV: The First Supercomputer*. Springer-Verlag Berlin Heidelberg GmbH, 1982.

[62] M. Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb. 2014. doi:10.1109/ISSCC.2014.6757323.

[63] D. Jagger. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, 1996. ISBN 978-0-13-736299-8.

[64] M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee. Formal semantics of predictable pipelines: a comparative study. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 103–108, Jan. 2020. doi:10.1109/ASP-DAC47756.2020.9045351.

[65] M. Johns and T. J. Kazmierski. A minimal risc-v vector processor for embedded systems. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–4, 2020. doi:10.1109/FDL50818.2020.9232940.

[66] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. ISSN 0163-5964. doi:10.1145/3140659.3080246.

[67] N. Kadri and M. Koudil. A survey on fault-tolerant application mapping techniques for network-on-chip. *Journal of Systems Architecture*, 92:39 – 52, 2019. ISSN 1383-7621. doi:10.1016/j.sysarc.2018.10.001.

[68] G. Kane. *MIPS RISC Architecture*. Prentice-Hall, Inc., USA, 1988. ISBN 0135847494.

[69] J. Kim, R. R. Rajkumar, and S. Kato. Towards adaptive gpu resource management for embedded real-time systems. *SIGBED Rev.*, 10(1):14–17, Feb. 2013. doi:10.1145/2492385.2492387.

[70] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi:10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[71] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68697-2.

[72] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Inc., 2nd edition, 2011. ISBN 1441982361.

[73] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021. ISBN 9781728186139. doi:10.1109/CGO51591.2021.9370308.

[74] S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 189–199, July 2016. doi:10.1109/ECRTS.2016.21.

[75] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202, Sept. 2014. doi:10.1109/ESSCIRC.2014.6942056.

[76] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008. ISSN 0272-1732. doi:10.1109/MM.2008.31.

[77] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 87–93, 2012. doi:10.1109/ICCD.2012.6378622.

[78] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017. ISSN 0925-2312. doi:10.1016/j.neucom.2016.12.038.

[79] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[80] S. O. H. Madgwick, A. J. L. Harrison, and R. Vaidyanathan. Estimation of imu and marg orientation using a gradient descent algorithm. In *2011 IEEE International Conference on Rehabilitation Robotics*, pages 1–7, June 2011. doi:10.1109/ICORR.2011.5975346.

[81] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 138–150, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916980.

[82] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(1):3–21, Jan. 2009. ISSN 0278-0070. doi:10.1109/TCAD.2008.2010691.

[83] R. Meyer and K. Schwarz. Fft implementation on dsp-chips-theory and practice. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 1503–1506 vol.3, 1990. doi:10.1109/ICASSP.1990.115692.

[84] S. Milutinovic, J. Abella, and F. J. Cazorla. On the assessment of probabilistic WCET estimates reliability for arbitrary programs. *EURASIP Journal on Embedded Systems*, 2017. ISSN 1687-3963. doi:10.1186/s13639-017-0076-8.

[85] T. Mitra. Time-predictable computing by design: Looking back, looking forward. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367257. doi:10.1145/3316781.3323489.

[86] R. Mondal, H. Ngo, J. Shey, R. Rakvic, O. Walker, and D. Brown. Efficient architecture design for the aes-128 algorithm on embedded systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, page 89–97, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379564. doi:10.1145/3387902.3392624. URL https://doi.org/10.1145/3387902.3392624.

[87] S. Morioka and A. Satoh. An optimized s-box circuit architecture for low power aes design. In B. S. Kaliski, ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 172–186, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36400-9.

[88] R. Morones. Documentation for the esc, imu and arduino ide code for the computer systems. 2013.

[89] S. Ovaska and O. Vainio. Predictive compensation of time-varying computing delay on real-time control systems. *IEEE Transactions on Control Systems Technology*, 5(5):523–526, 1997. doi:10.1109/87.623038.

[90] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96:879–899, May 2008. doi:10.1109/JPROC.2008.917757.

[91] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos. RISC-V2: A scalable RISC-V vector processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, Sept. 2020. doi:10.1109/ISCAS45731.2020.9181071.

[92] M. Platzer and P. Puschner. A processor extension for time-predictable code execution. In *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 34–42, 2021. doi:10.1109/ISORC52013.2021.00016.

[93] M. Platzer and P. Puschner. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In B. B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages

1:1–1:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-192-4. doi:10.4230/LIPIcs.ECRTS.2021.1. URL https://drops.dagstuhl.de/opus/volltexte/2021/13932.

[94] R. Pop and S. Kumar. A survey of techniques for mapping and scheduling applications to network on chip systems. Jan. 2009.

[95] B. Pourmohseni, S. Wildermann, M. Glaß, and J. Teich. Hard real-time application mapping reconfiguration for NoC-based many-core systems. *Real-Time Systems*, 55:433–469, 2019. doi:10.1007/s11241-019-09326-y.

[96] D. Prokesch, S. Hepp, and P. Puschner. A generator for time-predictable code. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 27–34, April 2015. doi:10.1109/ISORC.2015.40.

[97] P. Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems: Design and Analysis of Distributed Embedded Systems*, DIPES '02, page 163–172, NLD, 2002. Kluwer, B.V. ISBN 1402071566. doi:10.1007/978-0-387-35599-3_17.

[98] P. Puschner. The single-path approach towards WCET-analysable software. In *IEEE International Conference on Industrial Technology, 2003*, volume 2, pages 699–704 Vol.2, Dec 2003. doi:10.1109/ICIT.2003.1290740.

[99] P. Puschner. Experiments with wcet-oriented programming and the single-path architecture. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 205–210, 2005.

[100] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, pages 85–91, Jan 2002. doi:10.1109/WORDS.2002.1000040.

[101] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. Compiling for time predictability. In F. Ortmeier and P. Daniel, editors, *Computer Safety, Reliability, and Security*, pages 382–391, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33675-1.

[102] S. Quinton, T. T. Bone, J. Hennig, M. Neukirchner, M. Negrean, and R. Ernst. Typical worst case response-time analysis and its use in automotive network design. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[103] G. V. Raffo, M. G. Ortega, and F. R. Rubio. Nonlinear H∞ controller for the quad-rotor helicopter with input coupling. In *18th IFAC World Congress*, volume 44, pages 13834 – 13839, 2011. doi:https://doi.org/10.3182/20110828-6-IT-1002.02453.

[104] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-03-3. doi:10.4230/OASIcs.WCET.2006.671. URL `http://drops.dagstuhl.de/opus/volltexte/2006/671`.

[105] *Working draft of the proposed RISC-V V vector extension.* RISC-V International, Jan. 2021. URL `https://github.com/riscv/riscv-v-spec`. Version 0.10.

[106] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, Jan. 1978. ISSN 0001-0782. doi:10.1145/359327.359336.

[107] P. K. Sahu and S. Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1):60 – 76, 2013. ISSN 1383-7621. doi:10.1016/j.sysarc.2012.10.004.

[108] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 220–229, 2015. doi:10.1109/CODESISSS.2015.7331385.

[109] L. Santinelli, F. Guet, and J. Morio. Revising measurement-based probabilistic timing analysis. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 199–208, April 2017.

[110] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sept. 2017. doi:10.1109/PATMOS.2017.8106976.

[111] C. Schmidt, J. Wright, Z. Wang, E. Chang, A. Ou, W. Bae, S. Huang, V. Milovanović, A. Flynn, B. Richards, K. Asanović, E. Alon, and B. Nikolić. An eight-core 1.44-ghz risc-v vector processor in 16-nm finfet. *IEEE Journal of Solid-State Circuits*, pages 1–1, 2021. doi:10.1109/JSSC.2021.3118046.

[112] M. Schoeberl, P. Puschner, and R. Kirner. A single-path chip-multiprocessor system. In S. Lee and P. Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 47–57, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10265-3.

[113] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha,

C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. ISSN 1383-7621. doi:10.1016/j.sysarc.2015.04.002.

[114] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 212–219, Nov 2006. doi:10.1109/ISoLA.2006.63.

[115] E. Seligman, T. Schubert, and M. V. A. K. Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann, Boston, 2015. ISBN 978-0-12-800727-3. doi:10.1016/C2013-0-18672-2.

[116] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–5, 2011. doi:10.1109/HOTCHIPS.2011.7477515.

[117] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the vectorblox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. doi:10.1109/CODES-ISSS.2013.6658993.

[118] K. Seyid, A. Richaud, R. Capoccia, and Y. Leblebici. Block matching based real-time optical flow hardware implementation. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2206–2209, 2016. doi:10.1109/ISCAS.2016.7539020.

[119] A. K. Singh, P. Dziurzanski, H. R. Mendis, and L. S. Indrusiak. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Comput. Surv.*, 50(2), Apr. 2017. ISSN 0360-0300. doi:10.1145/3057267.

[120] S. Srinivasan, P. Janedula, S. Dhoble, S. Avancha, D. Das, N. Mellempudi, B. Daga, M. Langhammer, G. Baeckler, and B. Kaul. High performance scalable FPGA accelerator for deep neural networks, 2019. URL `https://arxiv.org/abs/1908.11809`.

[121] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, Mar. 2017. ISSN 0272-1732. doi:10.1109/MM.2017.35.

[122] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157 – 177, 2004. doi:10.1023/B:TIME.0000045316.66276.6e.

[123] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

[124] T. Ungerer, C. Bradatsch, M. Frieb, F. Kluge, J. Mische, A. Stegmeier, R. Jahr, M. Gerdes, P. Zaykov, L. Matusova, Z. J. J. Li, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, N. Lay, D. George, I. Broster, E. Quiñones, M. Panic, J. Abella, C. Hernandez, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15 (3), May 2016. ISSN 1539-9087. doi:10.1145/2910589.

[125] N. Vasilache, O. Zinenko, A. J. C. Bik, M. Ravishankar, T. Raoux, A. Belyaev, M. Springer, T. Gysi, D. Caballero, S. Herhut, S. Laurenzo, and A. Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. 2022. doi:10.48550/ARXIV.2202.03293.

[126] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual: User-Level ISA*. CS Division, EECS Department, University of California, Berkeley, CA, USA, June 2019.

[127] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 295–303, 2005. doi:10.1109/QSIC.2005.49.

[128] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based timing analysis. *Communications in Computer and Information Science*, 17:430–444, 10 2008. doi:10.1007/978-3-540-88479-8_30.

[129] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008. ISSN 1539-9087. doi:10.1145/1347375.1347389.

[130] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

[131] J. C. Wright, C. Schmidt, B. Keller, D. P. Dabbelt, J. Kwak, V. Iyer, N. Mehta, P.-F. Chiu, S. Bailey, K. Asanović, and B. Nikolić. A dual-core risc-v vector processor with on-chip fine-grain power management in 28-nm fd-soi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2721–2725, 2020. doi:10.1109/TVLSI.2020.3030243.

[132] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8): 2672–2684, Apr. 2020. doi:10.1109/TCSI.2020.2983185.

[133] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: Portable, scalable, and flexible FPGA-based vector processors. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 61–70, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605584690. doi:10.1145/1450095.1450107.

[134] J. Yu, G. Lemieux, and C. Eagleston. Vector processing as a soft-core CPU accelerator. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, page 222–232, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939340. doi:10.1145/1344671.1344704.

[135] 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*, 27(11):2629–2640, Nov 2019. ISSN 1557-9999. doi:10.1109/TVLSI.2019.2926114.

[136] 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)*, pages 101–110, 2014. doi:10.1109/RTAS.2014.6925994.

[137] A. Zulu and S. John. A review of control algorithms for autonomous quadrotors. In *Open Journal of Applied Sciences*, number 4, pages 547–556, Sep 2014. doi:10.4236/ojapps.2014.414053.