Worst-Case Execution Time Guarantees for Runtime-Reconfigurable Architectures

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation
von

Marvin Damschen
aus Moers

Tag der mündlichen Prüfung:	19. Dezember 2018
Referent:	Prof. Dr.-Ing. Jörg Henkel
	Karlsruher Institut für Technologie (KIT)
Korreferent:	Prof. Frank Mueller, Ph.D.
	North Carolina State University (NCSU)

Marvin Damschen
Burgstr. 110
76356 Weingarten (Baden)

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internetquellen 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.

Marvin Damschen

Acknowledgments

I would like to express my deep gratitude to my advisor Prof. Dr.-Ing. Jörg Henkel for believing in me from the beginning and providing an environment that was challenging and full of opportunities. I am thankful for his support and guidance, and the invaluable experience that he shared. By asking the right questions and supporting my ideas, he had a strong impact not only on the quality of my work, but especially on my development as a researcher.

I want to thank Prof. Frank Mueller from the North Carolina State University for agreeing to co-advise my thesis. I deeply appreciate our collaboration, which began when he welcomed me as a guest to his research lab, and his continuing support ever since.

Dr.-Ing. Lars Bauer also had a big impact on my Ph.D. research and I want to express my sincere gratitude for all the time he invests into research projects that provide a great environment for doctoral researchers. The contributions he made during his Ph.D. are the basis for the evaluation platform that was used in this work. I also want to thank Dr.-Ing. Artjom Grudnisky for being a great help in technical and general aspects during the first months of my Ph.D. and helping me to kick-start my Ph.D. research. I further want to thank Dr.-Ing. Farzad Samie for answering all my questions about the Ph.D. defense and Martin Rapp for providing comments to my thesis.

I was fortunate to be part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89), which is funded by the German Research Foundation (DFG), and was a great source of experiences, collaborations and inspirations for me. In this context I want to express my gratitude to Andreas Fried, Dr.-Ing. Manuel Mohr, Alexander Pöppl, Sven Rheindt, Florian Schmaus and everyone else taking part in the integration of research ideas into a common prototype. The outstanding collaboration eventually allowed us to demonstrate a prototype of the full invasive computing technology stack during the review phase of Invasive Computing.

My thanks also go to Dr. Enrico Rossi, at the time a Ph.D. student of Prof. Dr. Giorgio Buttazzo from the Scuola Superiore Sant’Anna in Pisa, for visiting us and being a great collaborator in the area of runtime-reconfigurable real-time systems.

During my Ph.D. research I was able to supervise multiple student works and I want to thank all the students for the work that they put in and their contributions to prototypes.

Finally, I want to thank my family. I want to express my gratitude to my parents, who supported my interests and education however they could. My wife Katharina was not only understanding, but a continuing support during this journey. I want to express my deepest appreciation for her love and support.

Thank you!

Karlsruhe, January 2019 *Marvin Damschen*

I should like to say two things, one intellectual and one moral:

The intellectual thing I should want to say to them is this: When you are studying any matter or considering any philosophy, ask yourself only what are the facts and what is the truth that the facts bear out. Never let yourself be diverted either by what you wish to believe or by what you think would have beneficent social effects if it were believed, but look only and solely at what are the facts. That is the intellectual thing that I should wish to say.

The moral thing I should wish to say to them is very simple. I should say: Love is wise, hatred is foolish. In this world, which is getting more and more closely interconnected, we have to learn to tolerate each other. We have to learn to put up with the fact that some people say things that we don't like. We can only live together in that way, and if we are to live together and not die together we must learn a kind of charity and a kind of tolerance which is absolutely vital to the continuation of human life on this planet.

— Bertrand Russel, *Face to Face* (BBC, 1959)

Kurzfassung

Echtzeitsysteme sind in unserem Alltag allgegenwärtig, beispielsweise in sicherheitskritischen Umgebungen wie der Automobil- und Luftfahrtelektronik oder der Robotik. Die Korrektheit eines Echtzeitsystems hängt nicht nur von der Korrektheit der durchgeführten Berechnungen, sondern auch von der nicht-funktionalen Anforderung der Einhaltung von Deadlines ab. Wird eine Deadline nicht eingehalten, kann dies zu ernsthaften Fehlfunktionen führen. Daher müssen maximale Ausführungszeiten (*worst-case execution times*, WCET) garantiert werden. Trotz signifikanter wissenschaftlicher Fortschritte, können lediglich Mikroarchitekturen im Hinblick auf WCET Garantien analysiert werden, die der Entwicklung von aktuellen hochperformanten Mikroarchitekturen um Jahre hinterher sind. Zur Erfüllung der wachsenden Anforderung an Performance in Echtzeitsystemen, sind *analysierbare* Funktionen zur Performancesteigerung erforderlich. Um dem Mangel an analysierbaren Funktionen zur Performancesteigerung zu entkommen, ist der Hauptbeitrag dieser Dissertation die Einführung von Laufzeitrekonfiguration von Hardwarebeschleunigern auf einem Field-Programmable Gate Array (FPGA) mit dem Ziel Performance unter WCET Garantien zu erreichen. Hierbei wird die Flexibilität des Systems aufrechterhalten und nicht etwa im Hinblick auf einen einzigen Anwendungsbereich eingeschränkt.

Zunächst trägt diese Dissertation in einer ausführlichen Analyse davon, wie (durchschnittliche) Performance auf *fused CPU-GPU Architekturen* erreicht wird, neuartige Ablaufplanungsansätze zur Arbeitsverteilung auf CPU und GPU bei. Fused CPU-GPU Architekturen sind aktuell eine der Hauptrichtungen innerhalb der Entwicklung von aktuellen hochperformanten Mikroarchitekturen, die eine CPU und eine GPU auf einem einzigen Chip vereint. Architekturen dieser Art für die Realisierung von Echzeitsystemen einsetzen zu können wäre überaus wünschenswert, da sie hohe Performance innerhalb eines beschränkten Flächen- und Leistungsbudgets bieten. Ein Ergebnis der präsentierten Analyse ist jedoch die Entdeckung eines Flaschenhalses in der Cache-Kohärenz von aktuellen fused CPU-GPU Architekturen, die den Last-Level-Cache zwischen CPU und GPU teilen. Dies führt dazu, dass (i) Performancevorhersagen erschwert werden und so (ii) ein geteilter Last-Level-Cache zwischen CPU und GPU der wachsenden Liste von Mikroarchitekturfunktionen hinzugefügt wird, die der durchschnittlichen Laufzeit nutzen, aber die Analyse von WCET Garantien auf hochperformanten Architekturen praktisch unmöglich machen. Somit wird der Bedarf an neuartigen Mikroarchitekturfunktionen für *vorhersagbare* Performance, die zugänglich für die Analyse von WCET Garantien sind, weitergehend motiviert.

Diesem Ziel folgend, wird ein Kontroller zur Steuerung von Laufzeitrekonfigurationen namens „Command-based Reconfiguration Queue“ (CoRQ) präsentiert, der für seine Operationen garantierte Latenzen bietet. Dies gilt insbesondere für den *Rekonfigurationsdelay*, der Zeit die benötigt wird um einen Hardwarebeschleuniger auf einer rekonfigurierbaren Fläche (FPGA) zu konfigurieren. CoRQ ermöglicht das Design von zeitlich analysierbaren Architekturen, die WCET Garantien unterstützen. Basierend auf dem –nun möglichen– garantierten Rekonfigurationsdelay von Beschleunigern wird eine WCET Analyse eingeführt, die es Tasks ermöglicht applikationsspezifische *Spezialinstruktionen* (CIs) zur Laufzeit zu rekonfigurieren. CIs werden von einer Prozessorpipeline ausgeführt und stoßen die Ausführung von einem oder mehreren Beschleunigern an. Verschiedene Maßnahmen zur Behandlung von Rekonfigurationsdelay werden im Hinblick auf ihren Einfluss auf WCET Garantien und Überabschätzungen verglichen. Die *Timinganomalie der Laufzeitrekonfiguration* wird identifiziert und sicher beschränkt: einen Fall in dem das schnellere Ausführen von Iterationen eines Berechnungskernels als in WCET während der Rekonfiguration von CIs die Gesamtlaufzeit eines Tasks verlängern kann. Sobald Tasks für WCET Garantien analysierbar sind die Laufzeitrekonfiguration von CIs durchführen, stellt sich die Frage *welche* CIs auf einer beschränkten

rekonfigurierbaren Fläche zur Optimierung der WCET konfiguriert werden sollen. Diese Frage wird für Systeme behandelt, in denen mehrere CIs mit jeweils unterschiedlichen Implementierungen (die einen Trade-off zwischen Latenz und Flächenbedarf erlauben) ausgewählt werden können. Dies ist üblicherweise der Fall, beispielsweise wenn von High-Level Synthese Gebrauch gemacht wird. Dieses sogenannte *Instruktionsselektionsproblem zur Optimierung der WCET* wird basierend auf der *Implicit Path Enumeration Technique* (IPET) modelliert. IPET ist die Pfadanalysemethode auf die sich Timing Analyseprogramme stützen, die dem Stand der Technik entsprechen. Nach unserem Wissen ist dies der erste Ansatz von WCET Optimierung, der den Gebrauch von globalen Programmflussinformationen (und Informationen über Rekonfigurationsdelays) ermöglicht. Ein optimaler Algorithmus (der Branch-and-Bound ähnelt) und ein schneller heuristischer Algorithmus (der auf Greedy basiert und in den meisten Fällen die optimale Lösung erzielt) werden vorgestellt. Schließlich wird ein Ansatz präsentiert, der es erstmals ermöglicht die Optimierung von statischen WCET Garantien *und* die Optimierung der durchschnittlichen Ausführung zur Laufzeit (unter Einhaltung von WCET Garantien) mittels Laufzeitrekonfiguration von Hardwarebeschleunigern zu vereinen. Der Ansatz besteht aus einer Analyse von Schranken für *Laufzeitslack* (der Menge an Ausführungszeit, die ein Programmteil schneller als in WCET ausgeführt wird), die es auf sichere Weise ermöglichen Beschleuniger für die Optimierung durchschnittlicher Performance zu rekonfigurieren. Bestehende WCET Garantien bleiben hierbei erhalten. Weiterhin wird ein Mechanismus präsentiert, der es auf Basis von einfachen Performancezählern ermöglicht den Laufzeitslack zu überwachen. Die benötigten Performancezähler sind üblicherweise in vielen aktuellen Mikroprozessoren verfügbar.
Zusammenfassend zeigt diese Dissertation, dass Laufzeitrekonfiguration eine Schlüsselfunktionalität für das Erreichen von vorhersagbarer Performance ist.

Abstract

Real-time systems are ubiquitous in our everyday life, e.g., in safety-critical domains such as automotive, avionics or robotics. The correctness of a real-time system does not only depend on the correctness of its calculations, but also on the non-functional requirement of adhering to deadlines. Failing to meet a deadline may lead to severe malfunctions, therefore worst-case execution times (WCET) need to be guaranteed. Despite significant scientific advances, however, timing analysis of WCET guarantees lags years behind current high-performance microarchitectures with out-of-order scheduling pipelines, several hardware threads and multiple (shared) cache layers. To satisfy the increasing performance demands of real-time systems, analyzable performance features are required. In order to escape the scarcity of timing-analyzable performance features, the main contribution of this thesis is the introduction of runtime reconfiguration of hardware accelerators onto a field-programmable gate array (FPGA) as a novel means to achieve performance that is amenable to WCET guarantees. Instead of designing an architecture for a specific application domain, this approach preserves the flexibility of the system.

First, this thesis contributes novel co-scheduling approaches to distribute work among CPU and GPU in an extensive analysis of how (average-case) performance is achieved on *fused CPU-GPU architectures*, a main trend in current high-performance microarchitectures that combines a CPU and a GPU on a single chip. Being able to employ such architectures in real-time systems would be highly desirable, because they provide high performance within a limited area and power budget. As a result of this analysis, however, a cache coherency bottleneck is uncovered in recent fused CPU-GPU architectures that share the last level cache between CPU and GPU. This insight (i) complicates performance predictions and (ii) adds a shared last level cache between CPU and GPU to the growing list of microarchitectural features that benefit average-case performance, but render the analysis of WCET guarantees on high-performance architectures virtually infeasible. Thus, further motivating the need for novel microarchitectural features that provide *predictable* performance and are amenable to timing analysis.

Towards this end, a runtime reconfiguration controller called "Command-based Reconfiguration Queue" (CoRQ) is presented that provides guaranteed latencies for its operations, especially for the *reconfiguration delay*, i.e., the time it takes to reconfigure a hardware accelerator onto a reconfigurable fabric (e.g., FPGA). CoRQ enables the design of timing-analyzable runtime-reconfigurable architectures that support WCET guarantees. Based on the –now feasible– guaranteed reconfiguration delay of accelerators, a WCET analysis is introduced that enables tasks to reconfigure application-specific *custom instructions* (CIs) at runtime. CIs are executed by a processor pipeline and invoke execution of one or more accelerators. Different measures to deal with reconfiguration delays are compared for their impact on accelerated WCET guarantees and overestimation. The *timing anomaly of runtime reconfiguration* is identified and safely bounded: a case where executing iterations of a computational kernel faster than in WCET during reconfiguration of CIs can prolong the total execution time of a task. Once tasks that perform runtime reconfiguration of CIs can be analyzed for WCET guarantees, the question of *which* CIs to configure on a constrained reconfigurable area to optimize the WCET is raised. The question is addressed for systems where multiple CIs with different implementations each (allowing to trade-off latency and area requirements) can be selected. This is generally the case, e.g., when employing high-level synthesis. This so-called *WCET-optimizing instruction set selection problem* is modeled based on the *Implicit Path Enumeration Technique* (IPET), which is the path analysis technique state-of-the-art timing analyzers rely on. To our knowledge, this is the first approach that enables WCET optimization with support for making use of global program flow information (and information about reconfiguration delay). An optimal algorithm (similar to Branch and Bound) and a fast greedy heuristic

algorithm (that achieves the optimal solution in most cases) are presented. Finally, an approach is presented that, for the first time, combines optimized static WCET guarantees *and* runtime optimization of the average-case execution (maintaining WCET guarantees) using runtime reconfiguration of hardware accelerators by leveraging *runtime slack* (the amount of time that program parts are executed faster than in WCET). It comprises an analysis of runtime slack bounds that enable safe reconfiguration for average-case performance under WCET guarantees and presents a mechanism to monitor runtime slack using a simple performance counter that is commonly available in many microprocessors.
Ultimately, this thesis shows that runtime reconfiguration of accelerators is a key feature to achieve predictable performance.

Author's Contributions

The following list enumerates journal, conference and workshop papers published by the author of this thesis while pursuing his doctorate at the Chair for Embedded Systems of the Karlsruhe Institute of Technology.

[1] Lars Bauer, Artjom Grudnitsky, Marvin Damschen, Srinivas Rao Kerekare, and Jörg Henkel. "Floating point acceleration for stream processing applications in dynamically reconfigurable processors". In: *IEEE Symp. on Embed. Syst. For Real-time Multimedia (ESTIMedia), Amsterdam, The Netherlands, October 8-9, 2015*. 2015, pp. 1–2. DOI: 10.1109/ESTIMedia.2015.7351762.

[2] Marvin Damschen, Lars Bauer, and Jörg Henkel. "Extending the WCET Problem to Optimize for Runtime-Reconfigurable Processors". In: *ACM Trans. on Archit. and Code Optim. (TACO)* 13.4 (2016), 45:1–45:24. DOI: 10.1145/3014059.

[3] Marvin Damschen, Lars Bauer, and Jörg Henkel. "CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems". In: *IEEE Embedded Systems Letters (ESL)* 9.3 (2017), pp. 77–80. DOI: 10.1109/LES.2017.2714844.

[4] Marvin Damschen, Lars Bauer, and Jörg Henkel. "Timing Analysis of Tasks on Runtime Reconfigurable Processors". In: *IEEE Trans. on Very Large Scale Integration Syst. (TVLSI)* 25.1 (2017), pp. 294–307. DOI: 10.1109/TVLSI.2016.2572304.

[5] Marvin Damschen, Frank Mueller, and Jörg Henkel. "Co-Scheduling on Fused CPU-GPU Architectures with Shared Last Level Caches". In: *IEEE Trans. on Comput.-Aided Design of Integrated Circuits and Syst. (TCAD)* (2018). ESWEEK Special Issue, to appear. DOI: 10.1109/TCAD.2018.2857042.

[6] Tanja Harbaum, Christoph Schade, Marvin Damschen, Carsten Tradowsky, Lars Bauer, Jörg Henkel, and Jürgen Becker. "Auto-SI: An adaptive reconfigurable processor with run-time loop detection and acceleration". In: *IEEE Intl. System-on-Chip Conf., (SOCC), Munich, Germany, September 5-8, 2017*. 2017, pp. 153–158. DOI: 10.1109/SOCC.2017.8226027.

[7] Alexander Pöppl, Marvin Damschen, Florian Schmaus, Andreas Fried, Manuel Mohr, Matthias Blankertz, Lars Bauer, Jörg Henkel, Wolfgang Schröder-Preikschat, and Michael Bader. "Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model Using Reconfigurable Hardware in Invasive Computing". In: *Workshop on UnConventional High Performance Computing (UCHPC), Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*. 2017, pp. 676–687. DOI: 10.1007/978-3-319-75178-8_54.

[8] Enrico Rossi, Marvin Damschen, Lars Bauer, Giorgio Buttazzo, and Jörg Henkel. "Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing with FPGAs". In: *ACM Trans. on Reconfig. Technol. and Syst. (TRETS)* 11.2 (2018). to appear. DOI: 10.1145/3182183.

[9] Stefan Wildermann, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jürgen Teich, Andreas Weichslgartner, and Andreas Zwinkau. "Invasive computing for timing-predictable stream processing on MPSoCs". In: *it - Information Technology* 58.6 (2016), pp. 267–280. DOI: 10.1515/itit-2016-0021.

The main focus of this thesis is on references [2–5], the contribution of Chapter 7 is currently under submission.

Selected Supervised Student Theses

The following list enumerates selected student theses that were supervised by the author of this thesis and that contributed to prototyping and implementation of the evaluation platforms used in the following chapters.

[i] Typke, Marc. “A SystemC/TLM-based Simulator for a Reconfigurable Heterogeneous Multi-core System”, *Master Thesis*, 2016.

[ii] Middelschulte, Leif. “Extending a WCET Estimation Tool for Runtime Reconfigurable Processors”, *Master Thesis*, 2016.

[iii] Eckhart, Artur. “A Command-Driven Reconfiguration Controller for Hard Real-Time Systems”, *Bachelor Thesis*, 2016.

[iv] Rapp, Martin. “A Mixed Criticality Architecture with Reconfigurable Accelerators”, *Master Thesis*, 2016.

[v] Blankertz, Matthias. “Extending the *i*-Core architecture for pipelined floating-point accelerators”, *Diploma Thesis*, 2017.

[vi] Maier, Eduard. “Heterogene Mehrkernprozessorunterstützung für *i*-Core”, *Diploma Thesis*, 2017.

[vii] Sader, Thomas. “Leveraging BCET Analysis to Improve WCET Estimates on Runtime Reconfigurable Processors”, *Diploma Thesis*, 2017.

[viii] Vutov, Petar. “A Linux Driver for the Reconfigurable Accelerator Queue Architecture”, *Bachelor Thesis*, 2018.

[ix] Münchbach, Florian. “Dynamic I/O configuration in a partially reconfigurable accelerator framework”, *Master Thesis*, 2018.

Contents

1 Introduction

Real-time embedded systems are ubiquitous in our everyday life, e.g., in safety-critical domains such as automotive, avionics or robotics. The correctness of a real-time system does not only depend on the correctness of its calculations, but also on the non-functional requirement of adhering to deadlines where, under circumstances safety-critical, output signals are produced. Failing to meet a deadline may lead to severe malfunctions, therefore they need to be guaranteed in a process called *timing validation* [107]. As part of the timing validation, a schedulability analysis is performed to guarantee that a given task set can be scheduled at runtime under any circumstances. To perform a schedulability analysis, the *worst-case execution time* (WCET) of every task from the task set needs to be known [107].

Determining an accurate upper WCET bound of a task is a complex problem, because performance-enhancing features of modern processors like pipelining, caches and branch prediction introduce a microarchitectural state. This microarchitectural state results in a dependency of the latency of instructions on the execution history. Assuming worst-case behavior of a microarchitectural component, e.g., a cache miss, in situations where the microarchitectural state cannot be determined statically does not necessarily result in a safe WCET bound for the whole task. The state of one component may influence other microarchitectural components, e.g., whether a cache access is a hit or miss can influence whether a branch condition is calculated in time or potentially mispredicted. Such an effect is called *timing anomaly* [86], and it enforces exhaustive exploration of every possible microarchitectural state when determining the worst-case bound for executing a sequence of instructions.

Modern high-performance processors like supplied by Intel[1] feature microarchitectures with out-of-order scheduling pipelines, several hardware threads and multiple (shared) cache layers, as detailed in Chapter 3. These average-case performance enhancing features cause an explosion of possible microarchitectural states that render timing analysis practically infeasible [6]. However, the demand for processing power in real-time systems is strongly increasing, e.g., automated driving requires vast amounts of sensor data to be processed under timing constraints. Therefore, high-performance microarchitectures amenable for WCET analysis are requested [6, 37, 99].

To escape the scarcity of timing-analyzable performance features, the main focus of this thesis is to introduce runtime reconfiguration of hardware accelerators onto a field-programmable gate array (FPGA) as a means to achieve performance that is amenable to the analysis of WCET guarantees. Hardware accelerators speed up the tasks' most compute-intensive parts, so called *computational kernels* (also known as hotspots) that are comprised of one or more nested loops. When implementing these accelerators as application-specific integrated circuits, the system would lack flexibility with respect to revised standards or new algorithms. Instead, using an architecture that is reconfigurable by employing an FPGA maintains a high flexibility and even allows for reconfiguring the accelerators at runtime, thereby increasing the performance as well as the computing efficiency (compared to a static set of accelerators) at the cost of a more complex timing analysis.

While runtime reconfiguration was previously investigated with respect to real-time scheduling [16, 36, 54, 93], novel models and analyses are required to make the benefits of runtime-reconfigurable architectures accessible for WCET guarantees by tasks, even in uniprocessor systems. In Chapter 5 it will be shown that –in addition to a considerable speedup– the overestimation of a task's static WCET guarantee can be reduced by providing WCET guarantees for kernels in which compute-intensive calculations are performed by reconfigurable hardware acceler-

[1] Intel is currently the biggest supplier of high-performance microprocessors worldwide with a market share of over 70% in Q1 of 2017 according to the International Data Corporation (IDC) (see `https://www.idc.com/getdoc.jsp?containerId=lcUS42519017`)

ators. Accelerators typically provide functionality that corresponds to several hundred instructions when executed on the CPU pipeline, possibly including conditional branches and other control flow. Analyzing instructions for worst-case latency introduces pessimism due to, e.g., pipeline hazards or instruction cache misses that need to be accounted for when the CPU behavior can not exactly be determined statically. The latency of the hardware accelerators that are executed on the reconfigurable fabric is under direct control of the application designer and often precisely known (e.g., this is the case when leveraging high-level synthesis tools [60]).

1.1 Thesis Contributions

The main contribution of this thesis is to establish worst-case execution time guarantees for runtime-reconfigurable systems as a means to achieve predictable performance. Specifically, *the novel contributions of this thesis are:*

- Novel co-scheduling approaches are presented in a case study on *fused CPU-GPU architectures*, a main trend in current high-performance microarchitectures that combines a CPU and a GPU on a single chip. A cache coherency bottleneck is uncovered that has implications for *predictable* performance on such architectures.
- A runtime reconfiguration controller called "Command-based Reconfiguration Queue" (CoRQ) is presented that provides guaranteed latencies for its operations and supports timing analysis of runtime reconfiguration for WCET guarantees.
- WCET analysis is introduced for tasks on a runtime-reconfigurable processor. Different measures to deal with reconfiguration delays are compared as well as the *timing anomaly* of runtime reconfiguration is identified and safely bounded.
- The WCET-optimizing instruction set selection problem is modeled with support for global program flow information and reconfiguration delay by extending state-of-the-art models used in timing analyzers for WCET guarantees. An optimal algorithm and a fast heuristic algorithm (that achieves the optimal solution in most cases) are presented.
- An approach is presented that for the first time combines optimized static WCET guarantees *and* runtime optimization of the average-case execution (maintaining WCET guarantees) using runtime reconfiguration of hardware accelerators. It comprises an analysis of runtime slack bounds that enable safe reconfiguration for average-case performance under WCET guarantees.

In the following chapter, the background on real-time systems and runtime reconfiguration is introduced that is beneficial to understanding the contributions of this thesis and puts them into context of real-time system research. Chapter 3 details how performance is achieved on current high-performance processors that follow one of the main current architectural trends of integrating a CPU and a GPU on a single die. It provides evidence that high-performance architectures, which target average-case performance, can virtually not be analyzed for execution time guarantees and motivates the need for timing-analyzable performance features. Afterwards, Chapter 4 presents how reconfiguration of accelerators can be performed in real-time systems within statically-guaranteed delays. Chapters 5 and 6 focus on the WCET analysis and optimization of tasks that utilize runtime reconfiguration under WCET guarantees, respectively. Chapter 7 presents an online optimization approach that monitors the *runtime slack* of a task (the amount of time it executed parts of code faster than in worst case) to reconfigure accelerators that benefit average-case execution instead of worst-case execution, while maintaining WCET guarantees. Finally, Chapter 8 concludes the contributions of this thesis.

2 Background

The background on real-time systems, worst-case execution time analysis, runtime reconfiguration and the utilized evaluation platform is introduced in the following sections.

2.1 Real-Time Systems

In contrast to general-purpose computing systems, *real-time systems* must meet non-functional requirements. More specifically, real-time systems are computing systems that must react within time constraints to events in their environment [20]. Consequently, their correctness does not only depend on the logical results of their computations, but also on the time at which results are produced. Time constraints are given as *deadlines*, i.e., a maximum time per task that is to be executed on the real-time system, within which the task needs to complete its execution.
In real-time systems, a task that fails to meet its deadline is not not only late, but wrong, because failing to meet a deadline can lead to severe malfunctions: An increasing amount of safety-critical application domains that play a crucial role in our society relies on real-time systems, e.g., chemical and nuclear plants, transportation systems (railway, avionics, automotive), telecommunications, medical systems, industrial automation, robotics, and more [20]. Generally, real-time systems are embedded as part of a larger system that is to be controlled, which ranges from small portable devices (e.g., cellular phones, cardiac pacemaker) to larger systems (e.g., aircrafts, industrial robots)[1].
Software bugs are a common cause for accidents in safety-critical applications that can have catastrophic consequences. A well-known example that demonstrates the importance of rigorous verification of real-time systems is a Patriot missile defense system that was operated in Saudi Arabia during the Gulf War. The defense system contained a software bug in its interrupt handling routine[2], which resulted in accumulation of delay in the system. The delay influenced the system's classification process of flying objects. On February 25, 1991, the defense system was in operation for about 100 hours and had accumulated a total delay of 343ms, which caused it to incorrectly classify an incoming missile as a false alarm (its trajectory was mispredicted by 687m). In a catastrophic result, the missile struck an American Army barracks and lead to the loss of 28 lives and numerous injuries. Extreme events like these have shown that software testing is not sufficient to verify the correctness of a real-time system. Instead, deadlines need to be guaranteed in a process called *timing validation* [107].
As part of the timing validation, a *schedulability analysis* is performed to guarantee that a given task set can be scheduled at runtime under any circumstances. Depending on the consequences that may result from a missed deadline, a real-time task is assigned to one of three different categories [20]:

- *Hard*: A real-time task is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under control.
- *Firm*: A real-time task is said to be firm if producing the results after its deadline is useless for the system, but does not cause any damage.
- *Soft*: A real-time task is said to be soft if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

[1] The terms 'real-time system' and 'real-time embedded system' are therefore used interchangeably in the remainder of this thesis.
[2] "Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia" GAO/IMTEC-92-26: Published: Feb 4, 1992. Publicly Released: Feb 27, 1992.

This thesis focuses on hard real-time tasks, i.e., 'real-time task' always refers to a hard real-time task in the remainder of this text. In order to perform a schedulability analysis of a hard real-time task set, each task of the task set needs to be analyzed for characteristics in terms of execution time, required resources, and precedence relations with other tasks. For guaranteeing that a given real-time task set can be scheduled at runtime under any circumstances, the worst-case execution time (WCET) of each task of the set needs to be determined (see [20] for details on hard real-time scheduling). The following section details how the WCET of a task is obtained.

2.2 Worst-Case Execution Time Analysis

In general, obtaining upper bounds on the execution times of tasks is not possible, because it would require the halting problem to be decidable [83]. Therefore, real-time tasks are programmed restrictively: they are required to always terminate and recursion depths as well as iteration counts of loops need to be statically known. Virtually any task executed on a modern hardware platform exhibits execution time variation that is influenced by the task's input. If the worst-case input (the input leading to the worst-case execution) of a task were known, a worst-case execution time (WCET) guarantee could be easily obtained [106]. Generally, however, this is not the case and the worst-case input is hard to derive. Therefore, an upper bound on the WCET (analogously, a lower bound on the best-case execution time (BCET)) is estimated during timing analysis instead of the actual WCET (and BCET) as shown in Fig. 2.1. The estimated bounds need to be *safe*, i.e., the WCET (BCET) bound must never underestimate (overestimate) the actual WCET (BCET), and *precise*, i.e., the overestimation (underestimation) of the WCET (BCET) should be as small as possible to enable a successful schedulability analysis later on.

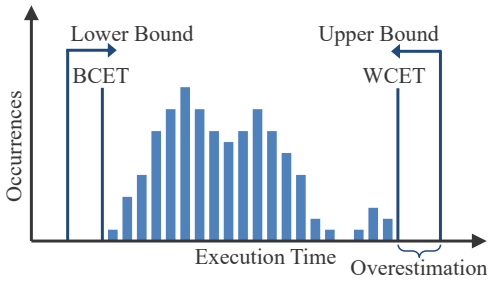


Figure 2.1: Histogram of all execution times of a task. The WCET of a task is upper-bounded using static timing analysis

In this thesis, static *timing analysis* is employed, which produces guaranteed WCET bounds, instead of measurement-based approaches, which produce bounds on observed execution times only (it is never guaranteed that all execution times haven been observed using measurements alone). The bounds obtained by static timing analysis allow safe schedulability analysis of hard real-time systems. A WCET bound is only valid for a specific hardware platform and is the result of the *worst-case path* through the task under analysis, i.e., the sequence of instructions that leads to the estimated WCET of the task. Consequently, timing analysis is performed on the finished task binary (instead, e.g., on the source code). It generally performs three major sub-analyzes on the task binary consisting of several passes each [107]:

(i) *Control-flow reconstruction and static analyses for control and data flow*. Reconstruct the control-flow graph (CFG) from the task binary, identify loops and bound iterations thereof (if possible), determine *infeasible paths* through the CFG (paths that exist in the CFG, but can never be executed in practice). Infeasible paths are eventually excluded during global bound analysis (in (iii)) to obtain a more precise WCET bound.

(ii) *Microarchitectural analysis*. Computes the execution time bounds of basic blocks. Assuming the latency of each instruction were constant, microarchitectural analysis would be simple. However, average-case performance enhancing features like deep pipelining or out-of-order scheduling of instructions, caches, branch prediction, etc. introduce a microarchitectural state [90]. This microarchitectural state results in a dependency of the latency of instructions on the execution history that can span numerous basic blocks. Therefore, it is not safe to analyze each basic block separately, but the microarchitectural state that can result from the execution of preceding basic blocks needs to be considered to obtain the latency bounds of a basic block. This

is done using *abstract interpretation* [25], a theory of program analysis that determines runtime properties of the task under analysis without actually executing it. Abstract interpretation allows to separate analysis of the microarchitectural state and the analysis of the worst-case path (which determines the WCET, as explained in (iii)) [98].

The state of one microarchitectural feature may influence other features, and the worst case of one feature does not necessarily lead to the worst-case execution of the whole task. Therefore, it is not safe to analyze microarchitectural features separately: E.g., whether a cache access is a hit or miss can influence whether a branch condition is calculated in time or potentially mispredicted. There are architectures, where a cache miss can lead to a correctly predicted branch condition that results in a shorter execution time than a cache hit (that would have led to a mispredicted branch) [69]. Such a situation, where a local worst case (e.g., a cache miss) leads to a shorter total execution time, is called a *timing anomaly* [86]. Consequently, it is not safe to assume a local worst case when static analysis of a task cannot precisely determine the state of each microarchitectural feature (e.g., cache contents). Instead, all possibilities need to be considered in further analysis. This leads to an explosion of microarchitectural states and has a strong influence on the applicability of methods for timing analysis to specific microarchitectures. Effectively, the microarchitectures that can be analyzed are several generations behind microarchitectures available today [90].

(iii) *Global bound analysis*. Combines information obtained in the previous analyzes (annotated CFG from (i) and WCET bounds of basic blocks from (ii)) to compute the WCET bound of the whole task. State-of-the-art timing analyzers compute the WCET by determining the worst-case path through the CFG using the ILP-based Implicit Path Enumeration Technique (IPET) [64, 106].

The contributions of this thesis do not rely on specific approaches to perform analyzes (i) and (ii) (implications to (ii) are addressed in Section 5.4). IPET, however, is a central technique in WCET analysis and also utilized by approaches presented in the following chapters. It is introduced in the following.

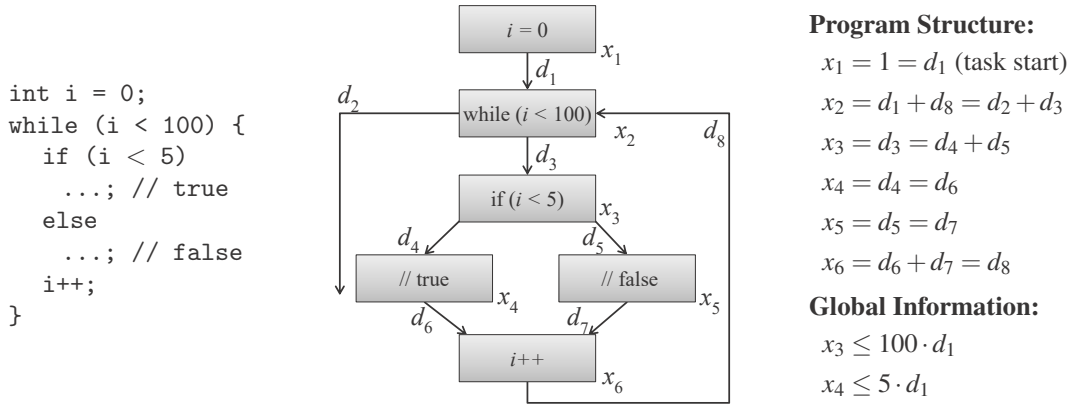
2.2.1 Global Bound Analysis using IPET

The approaches presented in Chapters 5 and 6 base on the Implicit Path Enumeration Technique (IPET) [64] for WCET bound calculation, as it is the program path analysis technique state-of-the-art timing analyzers rely on [4, 106]. IPET models program flow as arithmetic constraints in an ILP[3]-formulated problem. The objective function determines the CPU cycles executed on a path in the task's CFG. To find the WCET path, it needs to be maximized. Variables in the objective function represent the execution count of a single basic block (x_i) in the CFG and are weighted with the execution cycles of that basic block (c_i), which are determined in the microarchitectural analysis (see previous section). For a program with N basic blocks, the objective function is given as:

$$\max_{x \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i x_i \tag{2.1}$$

Similar to flow networks, the variables are constrained by modeling the control flow and capturing relative execution counts of basic blocks as ILP constraints. The more infeasible paths can be excluded by constraints, the more precise the WCET bound will be. IPET was first introduced in [64], which contains a detailed overview of how constraints are generated. A brief overview is given in the following. Besides the variables x_i representing the execution counts of basic blocks, variables d_i for every edge in the CFG are used. Figure 2.2 (b) shows the CFG of the simple source code excerpt shown in Fig. 2.2 (a). The loop header (represented by x_2) can be entered from outside using the edge represented by d_1 or from a previous iteration using d_8. The same basic block can be exited

[3] Integer Linear Programming [43]



(a) Task source code (b) Task CFG (c) IPET constraints

Figure 2.2: Example of constraint generation using the Implicit Path Enumeration Technique (IPET)

when the loop condition is false and the kernel is exited using d_2 or it can proceed to another iteration when the loop condition is true using d_3. Therefore, $x_2 = d_1 + d_8 = d_2 + d_3$ (see Fig. 2.2 (c)).
A key feature of IPET is that *global path information* about input-dependent control flow can be annotated using additional constraints, e.g., an upper bound of 100 loop iterations can be given by the constraint $x_3 \leq 100 \cdot d_1$. When static control and data flow analysis (see previous section) recognizes that the `true` case can be executed a maximum of 5 times, it is annotated using the constraint $x_4 \leq 5 \cdot d_1$. State-of-the-art timing analyzers utilize annotation languages [59] that enable users to conveniently annotate expert knowledge about task execution. Such annotations are automatically translated into IPET constraints and often lead to a more precise WCET bound.
Several extensions, e.g., for complex control flows and hardware timing effects depending on a long history of executed instructions have been published [7, 38, 106]. One of these extensions, multi-context analysis, is addressed in Chapter 5. In Chapter 6, IPET is extended from a WCET analysis problem to a WCET optimization problem for runtime-reconfigurable processors. The following section introduces the background on reconfigurable computing as a basis for the following chapters.

2.3 Reconfigurable Computing

Reconfigurable computing, i.e., performing computations using a reconfigurable fabric such as field-programmable gate arrays (FPGAs), was introduced in the early 1990s [97]. Today, it is an established computing paradigm in a growing number of application domains in research and industry, not only in embedded computing (e.g., signal processing [96], computer vision [53] or encryption [52]), but also in high-performance and scientific computing (e.g., financial pricing [34] or DNA-sequencing [13]), data centers (e.g., searching [84] or database queries [35]), networks (routing [68], intrusion detection [32]) and others. In these domains, applications generally comprise several compute-intensive loops, so-called *computational kernels*, that benefit greatly from implementation as application-specific hardware accelerators in terms of performance and energy efficiency. FPGAs enable the utilization of application-specific hardware accelerators without fabricating custom chips and they provide flexibility as well as the ability for upgrades, just like software.
Several alternatives exist when designing reconfigurable systems that combine a general-purpose CPU with a reconfigurable fabric [97]. Generally, a tighter integration reduces communication latency between CPU and reconfigurable fabric, but requires more effort in the architectural design of the system. Numerous products are available that add an FPGA as a separate chip to an existing system by attaching it to the system's peripheral bus. Especially for embedded systems however, it is crucial to minimize (i) the communication latency between CPU and reconfigurable fabric to enable acceleration of short-running kernels (e.g., in control loops) and (ii) the

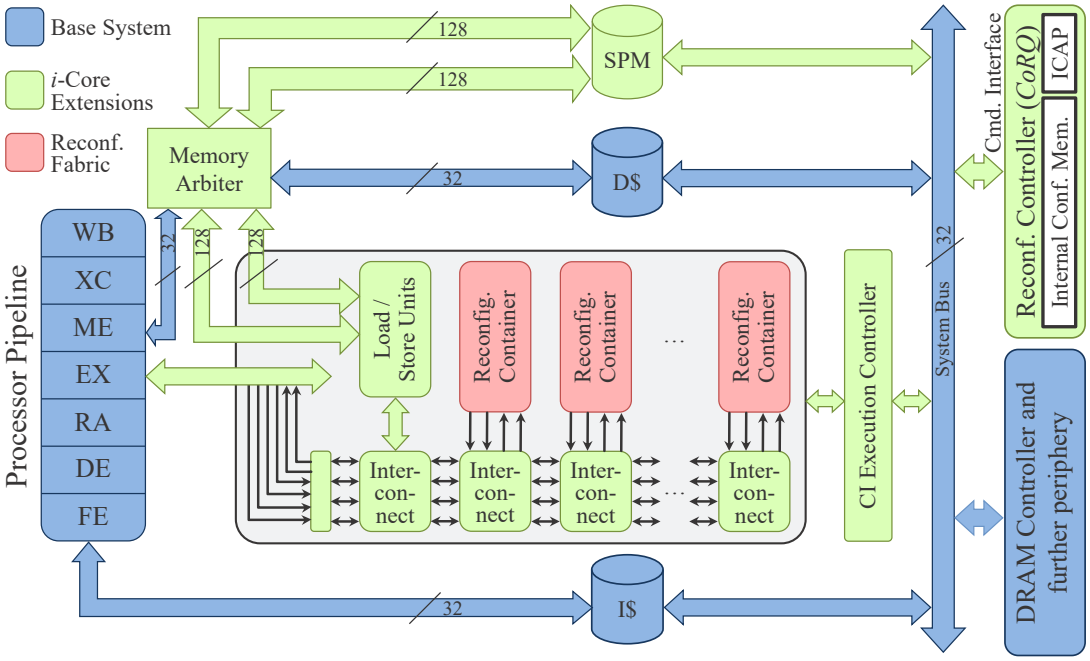


Figure 2.3: Overview of the evaluation platform – *i*-Core

system's power consumption as well as (iii) area footprint. Therefore, the advancing trend of processor integration has resulted in *reconfigurable SoCs* that combine FPGAs and CPUs on a single chip (e.g., Xilinx Zynq or Intel (formerly Altera) SoC FPGA), which have led to the wide adoption of reconfigurable systems in embedded systems, e.g., in implementations of advanced driver assistance systems in the automotive domain. In reconfigurable SoCs, CPU and FPGA are still separate processing devices that communicate over the (internal) system bus.
The evaluation platform that is employed to evaluate the contributions of Chapters 5 to 7 demonstrates that an even tighter integration of CPU and FPGA than in current reconfigurable SoCs is beneficial to target hard real-time execution. It is presented in the following section.

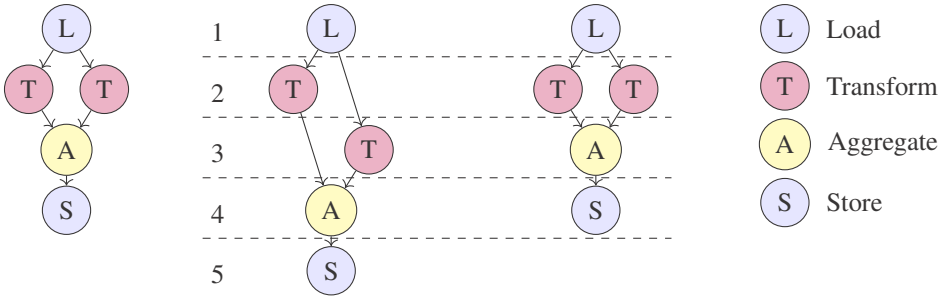
2.4 Evaluation Platform – *i*-Core

i-Core [10, 31] is a reconfigurable processor, i.e., it is based on a general-purpose (GPP) processor pipeline and enables the execution of runtime-reconfigurable *Custom Instructions* (CIs). Figure 2.3 gives an overview of the *i*-Core architecture. CIs extend the processor's core instruction set architecture (cISA) by application-specific instructions that are realized using (i) microcode and (ii) reconfigurable accelerators. They are detailed in the following.

2.4.1 Microcoded Custom Instructions

When the processor pipeline encounters a CI in its execute stage (EX), the pipeline stalls and initiates execution of the respective microprogram (i.e., a program written in microcode) that implements the functionality of the encountered CI on the *CI Execution Controller*. The communication between pipeline and CI Execution Controller is performed in a protocol that is similar to other multi-cycle instructions like integer division. The microprogram controls all resources of the reconfigurable fabric:

- Load/Store Units (LSUs), enable access to the main memory (through the processor's L1 data cache (D$)) and high-bandwidth scratchpad memory (SPM) for CIs



(a) DFG of a CI uses multiple accelerators (b) one 'Transform' accelerator configured (c) two 'Transform' accelerators configured

Figure 2.4: CIs define computations as DFGs that can be scheduled with different amounts of accelerators, resulting in different latencies

- Reconfigurable containers, (embedded) FPGAs that provide the reconfigurable area for runtime-reconfiguration of accelerators (one accelerator per container, each of similar complexity as, e.g., floating-point multiply-accumulate or a dozen integer operations)
- Interconnects, connect LSUs, reconfigurabe containers and the processor's register file to a common (four word-wide segmented) bus

When CIs access register operands and the non-cacheable SPM only, they can be modeled like just another multi-cycle instruction in the microarchitectural analysis during WCET estimation (see Section 2.2) and do not influence data cache analysis. Note that a single microprogram can utilize one or more accelerators. In other words, *the functionality defined by a CI is realized using one or more accelerators*. Application-specific hardware accelerators provide an important tradeoff: the more area is utilized, the higher the resulting performance. At the same time, multiple accelerators compete for the constrained reconfigurable area. This tradeoff is the result of instruction-level parallelism that can be exploited when more hardware resources are added to the application-specific accelerator. The main benefit of allowing CIs to utilize more than one accelerator is that this tradeoff can be chosen at runtime by providing several microprograms that implement the same CI, but utilize different amounts of accelerators that each implement a part of the CIs functionality. Consequently, CIs define computations as data-flow graphs (DFG) where nodes are accelerators and load/stores. Figure 2.4 (a) shows a simplified example of a CI that loads input data, performs transformations on the data, aggregates results and finally stores them. Depending on how many 'Transform' accelerators are configured in the reconfigurable containers at runtime, the DFG can be scheduled in 5 steps (see Fig. 2.4 (b)) or 4 steps (see Fig. 2.4 (c)). Each of these schedules corresponds to a microprogram for the CI Execution Controller, which implements the CI[4].

So far, it was discussed how CIs are executed, assuming all accelerators required by a certain implementation are currently configured on the reconfigurable fabric. However, CIs can be *unavailable*, i.e., there exists no schedule of the CI's DFG for the accelerators that are currently configured (reconfiguring all accelerators required by a CI can take several milliseconds). Two alternatives exists to handle the case that the *i*-Core attempts to execute an unavailable CI at runtime: *stalling* and *software emulation*. *Stalling* executes CIs on the reconfigurable fabric and trying to execute an unavailable CI is an error. Therefore, CIs need to be configured before the kernel is entered and the execution is stalled until the required CIs are available. *Software emulation* triggers functionally-equivalent software execution of an unavailable CI on the *i*-Core's pipeline using the base processor's cISA. It enables execution of the kernel while required CIs are still being reconfigured. Thus, progress can already be made without any CIs and as soon as reconfiguration of a CI finishes, the CI is utilized to speed up the following iteration of the kernel. While software emulation is always beneficial at runtime, it is more complex to analyze for WCET guarantees than stalling, which will be detailed in a more general context in Chapters 4 and 5.

[4] In the remainder of this thesis it will be referred to CI implementation and CI microprogram interchangeably.

i-Core exists as a constantly evolving hardware prototype. It is currently based on the Gaisler LEON3 SoC[5] and synthesizes to Xilinx Virtex-7 FPGAs. The LEON3 processor has a SPARC V8 in-order microarchitecture, separate data as well as instruction caches and supports several real-time operating systems. Appendix A presents demonstration setups that were extended and realized in the context of this thesis to show the practicality of the approach. A more detailed explanation of how the architecture is realized can be found in [10]. Additionally, a SystemC-based cycle-accurate simulator is available for early evaluation of runtime system algorithms. The specific details and parameters that were used to obtain the presented evaluation results are discussed in the respective chapters.

2.5 Associated Research Projects

The results of this thesis were achieved in the context of the research projects that are presented in the following.

2.5.1 Invasive Computing

The work presented in this thesis was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89), which just began its third and last funding phase (4 years per phase). The project is a collaboration between researchers from the Friedrich-Alexander University Erlangen-Nürnberg, Karlsruhe Institute of Technology and Technical University of Munich. In its current phase, it consists of 16 subprojects.

The governing thought of *Invasive Computing* is to grant applications, running on a massively-parallel computer that consists of 1000 and more compute cores, temporary exclusive access to resources like processor, communication channels and memory [50, 95]. This so-called *resource-aware programming* paradigm is of utmost importance to obtain high utilization as well as computational and energy efficiency numbers (including predictable execution). In Invasive Computing, a set of granted resources is called a *claim*. Applications allocate claims by *invading* resources, and then *infect* them with a program to run. Finally, the application *retreats* from its claim, freeing the resources. The state diagram of an invasive application is shown in Fig. 2.5.

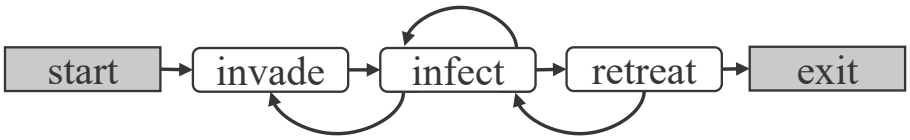


Figure 2.5: States of an invasive application (following the description of [74])

Realizing this resource-aware programming model effectively requires a holistic approach. Therefore, the subprojects of Invasive Computing cover the full compute stack of architecture, language/compiler, operating system/runtime system and applications. The hardware architecture targeted by invasive computing is a heterogeneous multiprocessor system-on-chip [50]. It consists of *tiles* of different types that are interconnected using a network-on-chip. Figure 2.6 shows an instance of the invasive architecture that uses three different types of tiles:

(i) compute tiles contain several RISC CPU cores that communicate over a shared bus,

(ii) memory tiles that provide DDR memory and

(iii) *i*-Core tiles contain RISC CPU cores and the reconfigurable processor ‘*i*-Core’ (which was presented in the previous section and is the evaluation platform of this thesis).

[5] `https://www.gaisler.com/`

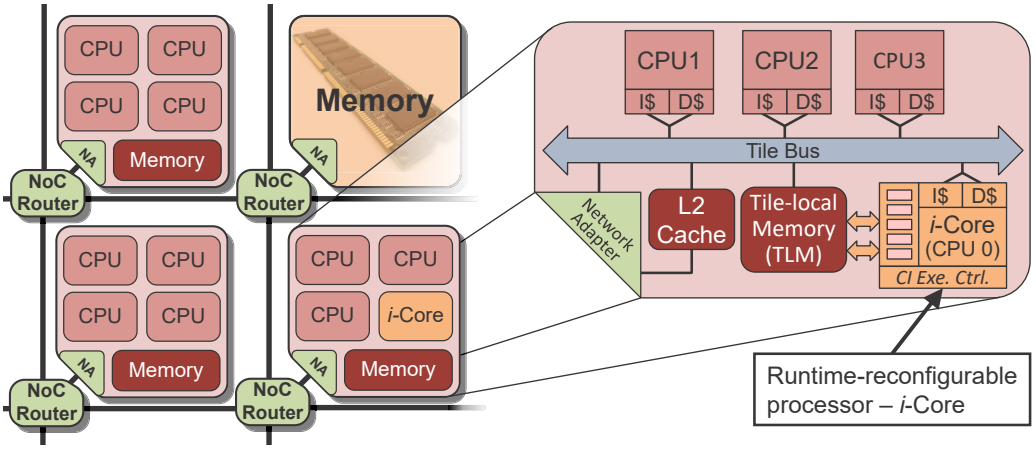


Figure 2.6: Overview of an instance of the tile-based invasive manycore architecture. Details of an *i*-Core tile are shown

i-Core is investigated as an architectural subproject of Invasive Computing. Within the project, it is a resource that can be invaded by applications for exclusive access, through which predictable execution is enabled. So far, however, execution time guarantees for runtime-reconfigurable processors are unavailable. This thesis introduces general methods for WCET analysis and optimization on runtime-reconfigurable processors that can be applied to the *i*-Core processor and the invasive architecture to achieve predictable high performance.

2.5.2 SPP 1500

Another associated research project is the DFG Priority Program SPP 1500 “Dependable Embedded Systems”, which focuses on the various reliability concerns in the nano-era [49]. The reliability concerns include manufacturing variability, aging, the impact of temperature and soft errors. These concerns are addressed from a wide range of perspectives including operating systems, compilers, micro-architectures and applications themselves. SPP 1500 comprises 12 projects in total from research groups of ten different universities throughout Germany. The project OTERA (Online Test Strategies for Reliable Reconfigurable Architectures) targets reliability concerns in runtime-reconfigurable architectures on the basis of *i*-Core (which is the main evaluation platform used in this thesis).

This section concludes the background for the main focus of this thesis discussed in Chapter 4 and following, i.e., WCET analysis and optimization using runtime reconfiguration. The following chapter takes a step back from architectures designed for real-time systems to verify the claim that performance on current high-performance architectures is increasingly hard to predict and motivate need for timing-analyzable performance features.

3 Achieving Performance on Fused CPU-GPU Architectures with Shared Last Level Caches

This chapter presents novel co-scheduling approaches to distribute work onto CPU and GPU in an extensive case study on how performance is achieved on heterogeneous high-performance processors that follow one of the main current architectural trends: integrating a (multi-core) CPU and a GPU on a single die[1]. Being able to employ such architectures in real-time embedded systems would be highly desirable, because they provide high performance within a limited area and power budget. E.g., NVIDIA partnered with numerous companies of the automotive domain (Audi, Mercedes-Benz, Toyota, Volvo among others) in the NVIDIA DRIVE project to create a computing platform that specifically aims at enabling autonomous driving[2]. Their current hardware platform "NVIDIA Drive PX Xavier" relies on integrating a CPU, a GPU and hardware accelerators on a single chip to achieve the required computing performance within a 30W power budget[3]. It remains an open question, however, how predictable execution times can be achieved on such platforms, which is crucial to be able to deploy them in actual products (e.g., self-driving cars) [85].

In this chapter it is shown that even when targeting average-case performance, performance predictions are beneficial to distribute work among heterogeneous compute devices for maximum performance. Three novel approaches to distribute work are introduced and compared, which leverage the unique features of architectures that share a last level cache between CPU and GPU. Furthermore, this chapter uncovers a cache coherency bottleneck in recent such architectures that has implications on predictable performance. It ultimately provides evidence to the claim made in Chapter 1 that high-performance architectures, which were designed for average-case performance, can virtually not be analyzed for execution time guarantees and motivates the design and analysis of a timing-analyzable architecture in the following chapters.

3.1 Fused CPU-GPU Architectures

With the release of AMD's Fusion and Intel's Ivy Bridge architecture in 2011, the trend of processor integration resulted in *fused CPU-GPU architectures* that integrate a CPU and general-purpose GPU on a single die. The main benefit of such an integration is that time-consuming memory transfers between main memory and dedicated GPU memory become unnecessary. Instead, CPU and GPU access the same physical memory such that *zero-copy* transfers can be employed. Zero-copy transfers ensure coherency and translate pointers to memory buffers for the common CPU and GPU address space, but do not actually transfer data. However, such an integration introduces a memory bottleneck, because CPU and GPU compete for memory bandwidth of the shared physical memory.

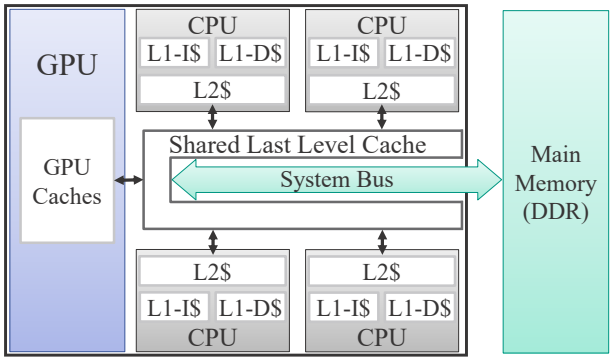


Figure 3.1: High-level overview of a fused CPU-GPU architecture with shared last level cache

[1] The work presented in this chapter was originally published in [30]
[2] https://www.nvidia.com/en-us/self-driving-cars/
[3] https://en.wikipedia.org/w/index.php?title=Drive_PX-series&oldid=864203140#Drive_PX_Xavier

In more recent architectures, e.g., Intel Broadwell and beyond, CPU and GPU were further integrated so that they access the shared last level cache (LLC) as shown in Fig. 3.1. This enables hardware-supported byte-level cache coherency between CPU and GPU. Effectively, CPU and GPU can execute computational kernels on the same data in parallel and solve problems collaboratively. In this case, the shared LLC has the potential to alleviate the memory bottleneck present in earlier fused CPU-GPU architectures (without a shared LLC), because it can serve accesses to a common working set instead of requiring frequent main memory accesses [110].

The idea of heterogeneous compute devices performing computations on a common memory is also captured in the Open Compute Language (OpenCL) standard 2.0. Most prominently, OpenCL introduces *Shared Virtual Memory* (SVM), i.e., a shared virtual address space between heterogeneous compute devices in an OpenCL program. SVM is also supported by fused CPU-GPU architectures without a shared LLC, e.g., AMD's Accelerated Processing Units or System on Chips that feature ARM's Mali Bifrost GPU. However, because excessive coherency traffic is required across heterogeneous devices [73], SVM was proven inefficient on such architectures. In contrast, on fused CPU-GPU architectures with a shared LLC, OpenCL 2.0 promises efficient support for byte-level coherent (so-called *fine-grained*) SVM as well as cross-device atomics [56].

This work presents the first investigation of *collaborative execution* of computational kernels on a fused CPU-GPU architecture with a shared LLC using fine-grained SVM, i.e., CPU and GPU share cache-coherent memory so that the work of a computational kernel can be processed in parallel by both compute devices. We detail how OpenCL programs are ported to OpenCL 2.0's fine-grained SVM. This process is applied to the entire Rodinia Benchmark Suite [22] and overheads of fine-grained SVM are evaluated. Collaborative execution of computational kernels on fine-grained SVM requires novel co-scheduling approaches that determine how much work should be performed on CPU and GPU, respectively, for maximum performance. In previous studies on collaborative execution that used zero-copy transfers on fused CPU-GPU architectures with a shared LLC and OpenCL 1.2, a single static data-centric distribution of work was established for all kernels per program [113, 114]. Fine-grained SVM enables to decide the distribution of work dynamically, based on observed progress made by CPU and GPU while executing a kernel. Thus, a decision should be made per kernel instead of per program.

This work contributes three dynamic co-scheduling approaches that utilize different capabilities of OpenCL 2.0: one kernel-external method based on online profiling and two kernel-internal methods that utilize *cross-device atomics* (variables that can be modified atomically across multiple compute devices). Cross-device atomics are currently supported by OpenCL 2.0 only, apart from that our approaches could also be realized, e.g., in NVIDIA CUDA. One of the kernel-internal methods utilizes *device-side enqueuing*, another feature introduced with OpenCL 2.0 that enables enqueuing kernels to an OpenCL device from within an executing kernel. Device-side enqueuing is a similar technique to dynamic parallelism in NVIDIA CUDA. However, it is shown that device-side enqueuing introduces too much overhead to be suitable for implementing co-scheduling approaches. The other two co-scheduling approaches (one kernel-external and one kernel-internal) are further evaluated using the Rodinia Benchmark Suite, which we ported to OpenCL 2.0. Our kernel-external method performs competitively to the optimal choice of executing kernels within a program either on CPU or GPU (clairvoyant *xor-Oracle*, some kernels on CPU others on GPU within the same program). The method achieves 97% of the xor-Oracle's performance on average. We show, however, that for most benchmarks of the Rodinia Benchmark Suite it is not beneficial to split the work of a kernel between CPU and GPU compared to running a kernel either on CPU or GPU when fine-grained SVM is used. This observation is further analyzed and it is shown that it cannot be explained by cache conflicts, i.e., false or true sharing, but is the result of inefficient cache coherence. As of today, Intel platforms are the only architectures that support OpenCL 2.0's fine-grained SVM using a shared LLC. Therefore, we focus on this architecture in the remainder of this chapter.

The novel contributions of this chapter are as follows:

- We evaluate the overhead of OpenCL 2.0's fine-grained Shared Virtual Memory, and analyze the suitability of cross-device atomics as well as device-side enqueuing for co-scheduling kernels on fused CPU-GPU architectures with a shared LLC in three different co-scheduling approaches.
- We develop a co-scheduling approach that is competitive to the optimal choice of executing kernels within a program either on CPU or GPU (on average 97% of the clairvoyant xor-Oracle's performance and 1.43× speedup over only using the GPU), and via analysis show that inefficient cache coherence is the major performance bottleneck for collaborative execution of the same kernel on current fused CPU-GPU architectures with shared LLC.
- We port the Rodinia Benchmark Suite to OpenCL 2.0 with fine-grained SVM and make Rodinia-SVM as well as a variety of co-scheduling approaches available as open source[4].

3.2 Related Work

3.2.1 Co-Scheduling on Fused Architectures

In state-of-the-art related work on co-scheduling for fused CPU-GPU architectures, CPU and GPU do not share the last level cache [8, 58, 65, 75, 113, 114]. Thus, techniques like fine-grained SVM are not supported and communication between CPU and GPU has to rely on explicit data transfers. [58] presents an online profiling-based approach that is similar to our host-side profiling approach, but only treats the GPU as an OpenCL 1.2 device while CPU computations are performed in the host code. Therefore, barriers are required after every kernel run, whereas our approach treats CPU and GPU as OpenCL 2.0 devices and utilizes OpenCL events for lightweight synchronization (see Section 3.5.2). Data transfer overheads between different devices are mentioned as a key issue, but not further analyzed. [8] uses the online profiling method of [58] and presents a power-aware co-scheduling method that aims to minimize the energy-delay product of heterogeneous applications running on a fused CPU-GPU architecture. The authors report an average of 12.3% percent improvement over the best performance-oriented schedules. [114] presents an offline, machine learning-based approach to co-scheduling that determines a single ratio that partitions the input data into separate parts processed by CPU and GPU, respectively. This saves additional transfers to maintain coherency between kernel executions, but does not allow for per-kernel decisions. [75] presents an OpenCL runtime system that automatically schedules kernels to multiple devices that were originally written for a single device. The runtime system takes care of buffer allocation and transfers to maintain coherency between all devices without programmer effort. [65] and [113] specifically target irregular workloads, in which some work items take considerably longer than others such that profiling information from a subset of work items is often not representative for the performance of the whole kernel. Both approaches identify application-specific features to model the computational kernels' performance for scheduling decisions.
Compared to our work, state-of-the-art co-scheduling approaches did not share cache-coherent memory between CPU and GPU, but were instead limited by explicit data transfers that were required to establish consistency.

3.2.2 Exploiting Shared Virtual Memory

In [110] the potential of fused CPU-GPU architectures with a shared LLC is explored simulatively. The authors present an approach where compiler-generated "pre-execution code" is run on the CPU, before executing a computational kernel on the GPU. The aim of this approach is to fill the shared LLC such that the amount of main memory accesses that need to be performed by the GPU is minimized. Using this approach, the authors report a performance improvement of up to 113%, and 21.4% on average. [103] presents an extension of the gem5-gpu

[4] Source code available at: `https://git.scc.kit.edu/CES/Rodinia-SVM`

simulator for fused CPU-GPU architectures [82] that supports the features of OpenCL 2.0. Compared to these works, our approach utilizes a commercial off-the-shelf architecture (Intel) instead of simulation.
In [73] a comprehensive performance evaluation of OpenCL 1.2, OpenCL 2.0 and Heterogeneous System Architecture (HSA) 1.0 is presented. In contrast to our work, the evaluated AMD Kaveri architecture does not feature a shared LLC between CPU and GPU. As a result, the authors observe that excessive coherency traffic is generated across devices that can affect performance significantly.
In summary, state-of-the-art related work on co-scheduling on fused CPU-GPU architectures either failed to leverage cache-coherent memory between CPU and GPU or only explored cache coherency between CPU and GPU in simulation.

3.3 Motivational Example

Before the introduction of OpenCL 2.0's fine-grained SVM, data needed to be explicitly transferred to compute devices. Furthermore, consistency guarantees for memory buffers that were accessed in parallel by different compute devices did not exist. Therefore, state-of-the-art co-scheduling approaches divided the input data into two separate parts that were processed by CPU and GPU, respectively [113, 114]. Effectively, a single ratio that determines the share of work to be performed on each compute device was applied to all kernels of an OpenCL program. With fine-grained SVM pointers can be shared and accessed consistently by multiple devices in parallel.

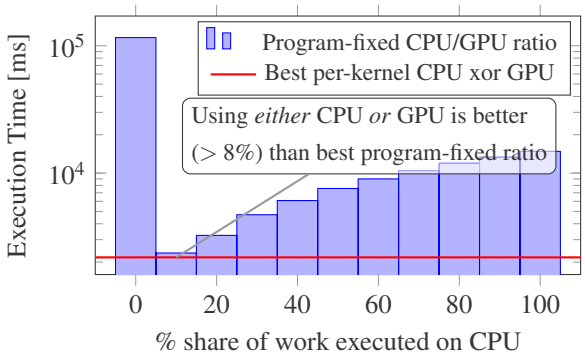


Figure 3.2: Particle Filter benefits from a per-kernel scheduling decision compared to a fixed ratio for the whole benchmark when executed on OpenCL 2.0's fine-grained SVM

Fig. 3.2 shows execution time results for the *Particle Filter* benchmark from the Rodinia Benchmark Suite (version 3.1 ported to fine-grained SVM) on an Intel Core i7-6700T (Skylake) fused CPU-GPU architecture with a shared LLC. The blue bars show the execution time for statically-fixed ratios of work performed on CPU and GPU, respectively, that are applied to all four kernels of the benchmark. The red line shows the execution time for deciding *per-kernel* whether to execute it *either* on CPU *or* on GPU. Only the single best overall decision (first two kernels on GPU, remaining two on CPU) is shown. In any case, the four kernels need to be executed in sequence. Two of four kernels contain loops that result in extremely poor performance when executed on the GPU only (thus, the execution time drops from $x = 0\%$ to $x = 10\%$), while the other two kernels benefit strongly from execution on the GPU compared to the CPU (thus, the execution time increases from $x = 10\%$ to $x = 100\%$). Therefore, deciding a single ratio of how to distribute work for all kernels results in a compromise that performs worse than executing each kernel exclusively on the most-suitable device. Due to the fact that fine-grained SVM is shared consistently among different compute devices without any explicit data transfers in between kernel executions.
This example shows that per-kernel decisions of how to distribute work have a performance benefit over a single data-centric ratio that is applied to all kernels of a program. In this work, we explore co-scheduling methods that leverage OpenCL 2.0 features to perform per-kernel decisions at runtime beyond the binary decision of either using the CPU or GPU but by utilizing both compute devices in parallel.

3.4 Background on Heterogeneous Execution using OpenCL

In this section we provide an overview of OpenCL in general and discuss features introduced in OpenCL 2.0 that we utilize for co-scheduling.

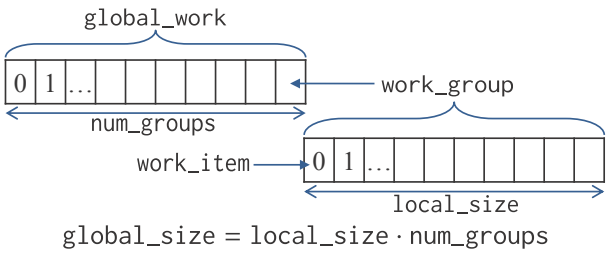


Figure 3.3: Hierarchy of Work Items in an OpenCL Kernel

3.4.1 OpenCL

The Open Compute Language (OpenCL) is an open standard for parallel programming of heterogeneous systems [57]. It consists of a host-side API and a C-like programming language for writing computational *kernels*. The host-side API provides access to the *platform*, i.e., a view of the system that the OpenCL program is executed on. The platform comprises one or more *devices* that are capable of executing OpenCL kernels. Within fused CPU-GPU architectures, CPU (including all cores) and GPU are separate devices[5] belonging to the same platform. For communication between host and devices, the host-side API provides functions to submit *commands* to *command queues*. Commands specify tasks that should be performed by a device, e.g., memory operations, synchronization or kernel execution. Each command queue is associated with exactly one device. *Events* can be used to formulate dependencies between commands (from the same or different command queues) as directed acyclic graphs. A command can emit an event upon successful execution. When submitting a command to a command queue, it can be specified that the command should only be executed after one or more events were emitted by finishing the execution of respective commands. Generally, when implementing an OpenCL kernel, the goal is to represent parallelism at the finest possible granularity. Figure 3.3 shows how OpenCL divides work hierarchically as well as OpenCL keywords used by the host-side API[6]. The smallest unit of execution is a *work item*. Each work item executes an instance of the kernel body, e.g., for a kernel that implements vector addition a work item would compute a single element. When submitting a kernel to a command queue, usually thousands of work items are instantiated and execute concurrently (as many as given by `global_size`). Work items are divided into *work groups*. Work groups are equally-sized (by `local_size`) and each group has a unique `group_id`. Work items have a `local_id` $(0, \ldots, \texttt{local\_size} - 1)$ that is unique within a work group only, as well as a globally unique `global_id` $(0, \ldots, \texttt{global\_size} - 1)$ that is used for address calculations. The `global_id` specifies on which part of the input data a specific work item executes the kernel body on. Only within a work group can work items perform barrier operations and share *local memory*. This way, the OpenCL compiler can perform device-specific optimizations, e.g., on CPUs a work group is serialized to a single thread.

3.4.2 OpenCL 2.0

The OpenCL specification 2.0 introduced several features that provide opportunities for improved collaboration between different devices as well as the host [44, 57]. The most prominent feature is Shared Virtual Memory (SVM) that introduces a shared virtual address space between host and devices in an OpenCL program. SVM eliminates explicit data transfers between host and device memory, and enables direct sharing of pointer-based data structures. OpenCL 2.0 introduces *coarse-grained* and *fine-grained* SVM. Coarse-grained SVM allows host and devices to share virtual memory pointers, but still requires buffers that are explicitly mapped and unmapped from host and devices. A coarse-grained SVM buffer can only be mapped to a single device or the host at a time, concurrent accesses by multiple devices are not supported. Fine-grained SVM is an optional feature of OpenCL 2.0 that defines memory consistency guarantees for SVM allocations that are concurrently accessed by the host and one or more devices. With fine-grained SVM, host and devices can share memory at byte-level granularity and

[5] Note that commonly used terms like 'compute unit' or 'processing element' are defined as specific parts of a device in OpenCL
[6] OpenCL supports up to three-dimensional index spaces. At this point, we explain the one-dimensional case for brevity

```
1  int* ptr = (int*)malloc(...);
2  for (int i=0; i<n; i++)
3    ptr[i] = i;
4  ptr_device = clCreateBuffer(...);
5  clEnqueueWriteBuffer(ptr_device, ptr,...);
6  clSetKernelArg(...,ptr_device);
7  clEnqueueNDRange(...);
8  clEnqueueReadBuffer(ptr_device, ptr,...);
9  clFinish(...);
10 printf("Result: %d\n", ptr[0]);
```

```
1  int* ptr = (int*)clSVMAlloc(...);
2  for (int i=0; i<n; i++)
3    ptr[i] = i;
4
5
6  clSetKernelArgSVMPointer(...,ptr);
7  clEnqueueNDRange(...);
8
9  clFinish(...);
10 printf("Result: %d\n", ptr[0]);
```

Figure 3.4: Simplified example of memory allocation in OpenCL 1.2 (left) and OpenCL 2.0 with fine-grained SVM (right)

read from it concurrently. Concurrent writes are supported to non-overlapping bytes. Consistency is guaranteed before and after each command execution. When more fine-grained consistency is required, *atomics* can be used. Atomics are another optional feature introduced by OpenCL 2.0. In combination with fine-grained SVM, atomics can be shared between different devices. This enables cross-device atomic operations and additionally provides a means of synchronization. This way, byte-level consistency can be guaranteed within a kernel.
Before OpenCL 2.0, the only way to execute commands on a device was to submit commands to a command queue using the host-side API. This means that the number of work items that should be executed when launching a kernel needed to be known before the kernel was executed. OpenCL 2.0 introduces *device-side enqueuing*, i.e., kernels get the ability to enqueue child kernels in a device-side command queue. Similarly to dynamic parallelism in NVIDIA CUDA, this enables implementation of kernels that perform calculations iteratively or use recursion. Like in host-side enqueuing, dependencies between child kernels can be specified using events, but generated events are only visible to the parent kernel. Child kernels run asynchronously to the parent kernel. However, the parent kernel is only registered as successfully executed (and may emit an event), when all its child kernels finished execution.

3.5 Utilizing Fine-Grained SVM on Fused CPU-GPU Architectures

3.5.1 Memory Allocation

Until OpenCL 2.0, communication between the host program and compute devices required explicit allocation of device-side buffers. As shown in the simplified example in Fig. 3.4 (l.4, left), memory that is allocated and initialized by the host program needs to be transferred to the device-side buffer first (l.5), before a kernel can be launched using `clEnqueueNDRange(...)`. After kernel execution finishes, the results are transferred back (l.8).
In Rodinia-SVM, we removed all device-side buffer allocations from the original Rodinia Benchmark Suite and utilize fine-grained SVM instead, as shown in Fig. 3.4 (right). This allows all devices and the host to access memory using shared pointers. As a result, all explicit transfers between host and devices are eliminated. Furthermore, while device-side buffers are owned by a single device at a time, fine-grained SVM can be accessed consistently by multiple devices and the host.

3.5.2 Kernel Launch and Synchronization

For launching kernels on a fused CPU-GPU architecture, one command queue is instantiated for each device (CPU and GPU). Then, the same kernel is enqueued with only a share of the total work items (`global_size`, see Fig. 3.3) plus offsets that are used for calculating global work item IDs. ID calculation depends on the specific co-scheduling method, and is therefore detailed in Section 3.6.

```
1 clEnqueueNDRangeKernelFused(commandsCPU, commandsGPU, kernel,...) {
2   // ... (calculate work item shares and IDs)
3   if(workItemsCPU>0) // work assigned to CPU?
4     clEnqueueNDRangeKernel(commandsCPU, kernel, ..., &eventGPUDone[curr-1],
          &eventCPUDone[curr]);
5   else
6     clSetUserEventStatus(eventCPUdone[curr], CL_COMPLETE);
7   if(workItemsGPU>0) // work assigned to GPU?
8     clEnqueueNDRangeKernel(commandsGPU, kernel, ..., &eventCPUDone[curr-1],
          &eventGPUDone[curr]);
9   else
10    clSetUserEventStatus(eventGPUdone[curr], CL_COMPLETE);}
```

Figure 3.5: Launching a kernel on a fused CPU-GPU architecture without host-side synchronization

Figure 3.6: Compared to the original OpenCL 1.2 implementation of the Rodinia Benchmarks Suite that executes on the GPU only and uses device-side buffers, the use of OpenCL 2.0 incl. fine-grained SVM introduces overheads but maintains consistency

Table 3.1: Rodinia Benchmark Suite – OpenCL Benchmarks

Name	Abbreviation	#Kernels
Back Propagation	bp	2
Breadth-First Search	bfs	2
B+Tree	b+	2
CFD Solver	cfd	4
GPUDWT	dwt	3
Gaussian Elimination	ge	2
Heart Wall	hw	1
HotSpot3D	hs3D	1
HotSpot	hs	1
Hybrid Sort	hys	7
K-Means	km	2
LavaMD	md	1
Leukocyte Tracking	lc	3
LU Decomposition	lud	4
Myocyte	mc	1
Nearest Neighbor	nn	1
Needleman-Wunsch	nw	2
Particle Filter	prtf	4
Path Finder	pthf	1
Streamcluster	sc	1

Earlier versions of OpenCL required explicit synchronization at the host-side, e.g., using `clFinish(...)` (see Fig. 3.4) or `clWaitForEvents(...)`, to achieve consistency [114]. Synchronization with the host induces a significant overhead, however, because the devices' command queues can no longer be processed in parallel. Because fine-grained SVM maintains consistency, host-side synchronization is not required anymore. However, we still need to ensure that CPU and GPU execute kernels in lock step, i.e., when launching a sequence of kernels like `clEnqueueNDRange(kernelA,...);` `clEnqueueNDRange(kernelB,...);`, the CPU should not begin executing `kernelB` before the GPU finished executing `kernelA` and vice versa. Otherwise, results from `kernelA` that `kernelB` depends on might not be ready when one device races ahead. This can lead to erroneous results. As shown in Fig. 3.5, we utilize events to express these dependencies. For each device a ring buffer is allocated that stores one event for each enqueued kernel (`eventCPUDone` and `eventGPUDone`, respectively). If work items are assigned to the CPU, the kernel is enqueued on the CPU (l.4). The execution of the kernel depends on an event that is emitted when the previous kernel that was enqueued to the GPU completes execution (`eventGPUDone[curr-1]`). In case no work items were assigned to the CPU, the event that indicates completed execution of the current kernel launch on the CPU is emitted immediately so that no deadlocks occur (`eventCPUDone[curr]`, l.6). Kernel launches on the GPU are performed analogously. In Rodinia-SVM, we replaced all calls to `clEnqueueNDRange(...)` with our `clEnqueueNDRangeFused(...)` implementation. Furthermore, the co-scheduling methods that we detail in Section 3.6 are also applied by `clEnqueueNDRangeFused(...)`.

3.5.3 Overheads of Fine-Grained SVM

Figure 3.6 shows execution time results for all benchmarks of the Rodinia Benchmark Suite (version 3.1, listed in Table 3.1) in two variants: the original OpenCL 1.2 version as well as our OpenCL 2.0 port where all device-side buffers were replaced by fine-grained SVM allocations as explained in Section 3.5.1. In both variants, kernels are

executed on the GPU only (the fused kernel launch of Section 3.5.2 is not used) on a Intel Core i7-6700T (Skylake). Kernel compilation times are omitted[7]. The results show that the convenience of being able to pass host-side pointers directly into kernels comes at a cost. In particular, short-running benchmarks (100ms and less) are significantly slowed down, e.g., `ge` takes almost 3.5× longer (112ms instead of 32ms) when executed on fine-grained SVM instead of device-side buffers. Benchmarks that run 100ms or more in the OpenCL 1.2 version only take 1.14× longer on average (geometric mean). Longer-running benchmarks that alternate between kernel execution and host-side computations like `hw` and `sc` even benefit from fine-grained SVM (1.9× and 1.48× speedup, respectively), because with OpenCL 1.2 they explicitly need to synchronize with the host and invoke transfers after every kernel execution. However, the geometric mean execution time increase over all benchmarks for the OpenCL 2.0 versions compared to the OpenCL 1.2 version is 1.51×. The overheads stem from the fact that the OpenCL 1.2 device-side buffers used in the Rodinia benchmarks are already allocated as *zero copy* buffers on fused CPU-GPU architectures[8], i.e., instead of allocating separate host-side and device-side memory, both buffers are mapped to the same shared physical memory. Consequently, data transfers between host-side and device-side buffers do not actually transfer data, but only translate pointers and initiate the OpenCL 1.2 runtime system to establish consistency between CPU and GPU. OpenCL 2.0's fine-grained SVM adds overhead compared to zero copy buffers, because consistency is not only established explicitly using transfers (e.g., at the beginning and end of a computation often consisting of multiple enqueued kernels), but continuously when kernels are executed.

Ultimately, these overheads have to be considered when implementing OpenCL 2.0 programs to decide whether to use fine-grained SVM or not. However, fine-grained SVM does not only provide the convenience of shared pointers, but also enables new features like cross-device atomics. In the following we will present co-scheduling methods that exploit these new features and evaluate them using Rodinia-SVM.

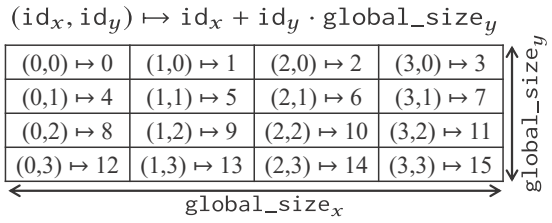


Figure 3.7: For co-scheduling, multi-dimensional IDs are mapped to one-dimensional IDs

3.6 Our Co-Scheduling Methods

Let us define two types of co-scheduling methods, namely (1) *device-side co-scheduling*, where work group scheduling is performed during execution of the respective kernel by the executing devices themselves, and (2) *host-side co-scheduling*, where work groups are assigned to CPU and GPU using the host-side OpenCL API only (outside of the kernels). In the following we will present two device-side and one host-side co-scheduling methods. All methods leverage OpenCL 2.0's fine-grained SVM to achieve consistency while executing kernels on CPU and GPU in parallel.

When enqueuing an OpenCL kernel using the OpenCL host API function `clEnqueueNDRangeKernel(...)`, the `global_size` parameter specifies how many work items should be launched by the OpenCL runtime system on a specific device (see Fig. 3.3). `global_size` can be given as an up to three-dimensional array. In this case, work items are assigned a global ID for each dimension by the OpenCL runtime system. For co-scheduling, we project multi-dimensional kernel IDs onto one-dimensional IDs. An example for the two-dimensional case is given in Fig. 3.7.

In our co-scheduling methods, we launch a subset of the total work items on CPU and GPU and then proceed to schedule the remaining work items based on the observed performance. The main idea behind the device-side methods is to treat the work of a kernel as a bag-of-tasks that contains independent work groups. Initially, only a few work groups are launched (enough to fully utilize CPU and GPU). The work items of these work groups

[7] Kernel compilation can be avoided using `clCreateProgramWithBinary(...)`
[8] using `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR` flags

```
1 typedef struct
    global_work_state_struct {
2   atomic_uint workDone;
3   size_t globalWork;
4 } global_work_state;
```

Figure 3.8: A `global_work_state` is shared between work items using fine-grained SVM to realize device-side scheduling

```
1 __kernel void kernel(...) {
2   PREAMBLE
3   ... // --- original kernel code ---
4   POSTAMBLE }
```

Figure 3.9: The device-side methods add a preamble and postamble to each kernel that implement the co-scheduling methods

act as workers that autonomously acquire and process work from the bag-of-tasks. To implement this scheme, the device-side methods utilize a `global_work_state` struct that is stored in SVM and shared between CPU and GPU (see Fig. 3.8). `globalWork` is the total amount of times the body of an enqueued kernel needs to be executed (equal to the `global_size` parameter passed to `clEnqueueNDRangeKernelFused(...)`). In all methods, the kernel is executed `globalWork` times in total. `workDone` is an atomic counter that keeps track of how many work items were executed. It is used to calculate work item IDs and to decide whether another work group needs to be scheduled, i.e., while `workDone` $<$ `globalWork`. Furthermore, all device-side methods add a preamble or postamble macro to each kernel as shown in Fig. 3.9. The specific preamble and postamble implementations are presented below. Please note that, e.g., modifying atomic variables, calculating work item IDs or handling corner cases, results in lengthy code that we simplified in our presentation below for comprehensibility[9].

3.6.1 Atomic Counting

In the *atomic counting* method, each work item acts as a worker that loops over the original kernel code. Initially, multiple same-sized work groups are launched (`clEnqueueNDRangeKernelFused(...)`) and execute in parallel (e.g., one work group per CPU core and multiple ones on GPU). No further work groups are launched during kernel execution. Each work item sequentially executes the kernel body repeatedly for different global IDs. Work items that belong to the same work group execute the kernel body in lock step as shown in Fig. 3.10. This way, they can share an atomic counter to derive their global IDs and iterate through all IDs that constitute the global work at `local_size` granularity. As detailed in Fig. 3.11, the atomic counter `workDone` (initially zero) is used to assign group IDs to work items of the same group:

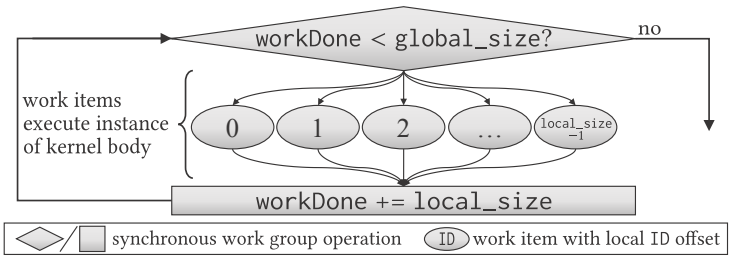


Figure 3.10: A single work group executes in lock step (*atomic counting*). Multiple work groups execute in parallel

Before each execution of the original kernel code (l.7), each work group (the last work item of a work group) fetches the value of `workDone` and increments the counter by the work group size (l.4 and l.9). The while loop beginning in line 6 is executed until the total amount of work required by the respective kernel launch is done. `workDoneCpy` (defined in l.2) is a variable that stores the fetched value of `workDone` and is allocated once for all work items that belong to the same work group (once for each work group). Independent of how many work groups execute in parallel, `workDoneCpy` will take the values of 0, `get_local_size()`, 2·`get_local_size()`, ..., `globalWork`−`get_local_size()`, each exactly once for a single work group that enters the while loop (`globalWork` is an integer multiple of `local_size`, see Fig. 3.3). Accessing the atomic counter only once per iteration of a work group (instead of, e.g., once per work item) reduces contention during the atomic operations,

[9] The full implementation of all approaches is available at: `https://git.scc.kit.edu/CES/Rodinia-SVM`

```
1 __kernel void kernel(...) {
2   local unsigned int workDoneCpy;
3   if (get_local_id() == get_local_size()-1)
4     workDoneCpy = atomic_fetch_add(workDone, get_local_size());
5   barrier(CLK_LOCAL_MEM_FENCE);
6   while (workDoneCpy < globalWork) {
7     ... // --- original kernel code ---
8     if (get_local_id() == get_local_size()-1)
9      workDoneCpy = atomic_fetch_add(workDone, get_local_size());
10    barrier(CLK_LOCAL_MEM_FENCE); }}
```

Figure 3.11: In *atomic counting*, work groups loop over the original kernel code until the total amount of work is done

```
1 __kernel void kernel(...) {
2   ... // --- original kernel code ---
3   if (get_local_id() == get_local_size()-1) {
4     int workDoneCpy = atomic_fetch_add( workDone, get_local_size());
5     if (workDoneCpy < globalWork) {
6       ndrange_t child_ndrange = ndrange_1D(workDoneCpy, get_local_size(),
            get_local_size());
7       enqueue_kernel(get_default_queue(), CLK_ENQUEUE_FLAGS_NO_WAIT,
            child_ndrange, ^{kernel(...);}); }}}
```

Figure 3.12: The *device-side enqueuing* method enqueues additional work groups using device-side queues

but work items of the same work group need to synchronize after each iteration (thus, execute in lock step). Synchronization is achieved using a barrier. It ensures that every work item of the same work group sees the same value of `workDoneCpy` at all times. This way `workDoneCpy` can be used to derive work item IDs, i.e., `get_global_id()` is redefined as `workDoneCpy` + `get_local_id()`. Ultimately, the original kernel body is executed exactly once for each work item ID 0, 1, 2, ..., `globalWork`−1.

3.6.2 Device-Side Enqueuing

The *device-side enqueuing* method does not define a preamble, but only a postamble as detailed in Fig. 3.12. Similarly to the atomic counting method, it uses the atomic counter `workDone` to keep track globally of how many times the kernel body was executed. Again, only as many work groups are launched initially as needed to fully utilize CPU and GPU (using `clEnqueueNDRangeKernelFused(...)`). The main difference to the atomic counting method is how work is processed by the work items. Instead of looping, a single work item executes the kernel body only once. After executing the kernel body, additional work groups may be launched by the work items itself using OpenCL 2.0's device-side enqueuing. As shown in Fig. 3.12, the work item with the highest ID inside a work group (l.3) launches another work group by enqueuing the current kernel into the device-side queue (l.7).

In OpenCL 2.0, Kernels are enqueued to the device-side command queue using the Clang [61] *block syntax*, a non-standard C extension by Apple Inc. (also known as *closure* in other programming languages) that allows to define functions that can access variables outside their scope (belonging to a captured environment). In our case (l.7) the block `^{kernel(...);}` defines a function that only calls the current kernel with the (captured) arguments that were passed to the initial kernel call from the host-side API. This may seem overly complex for our use case, however, potential alternatives like function pointers are not supported in OpenCL 2.0 and function calls are always inlined [44].

Line 6 defines the parameters of the enqueued kernel, i.e., the global ID offset (`workDone`), the total amount of work items to be launched (`get_local_size()`) and the work group size (`get_local_size()`), respectively. Effectively, work item ID calculation does not have to be redefined as in atomic counting, but `get_global_id()` will return the correct IDs 0, 1, 2, ..., `globalWork`-1 for exactly one work item each. We also evaluated variants of this method, e.g., enqueuing larger amounts of work items than single work groups per `enqueue_kernel(...)` call. However, OpenCL 2.0's device-side enqueuing in general introduces too much overhead (caused by runtime evaluation of the block syntax) to be suitable for co-scheduling as we show in Section 3.7.1.

3.6.3 Host-Side Profiling

In contrast to the device-side co-scheduling methods, the *host-side profiling* method does not apply any modifications to the executed kernels and work items behave exactly the same as in standard OpenCL. Host-side profiling utilizes the OpenCL host-side API, only. Similar to the Inspector-Executor paradigm, the performance behavior of a specific kernel is characterized in an initial phase. Afterwards, this characterization is used to schedule all following executions of the same kernel. Upon the first execution of a kernel k, only a fraction of the total work items (`profiling_size`) is executed for profiling as shown in Fig. 3.13. The `profiling_size` is split with half of it executing on the CPU and the other half on the GPU. The execution time of the profiling depends on the specific kernel. OpenCL events are used (1) to synchronize both devices with the host program once profiling finishes and (2) to obtain the execution times of the work items executed on CPU (timeCPU) and GPU (timeGPU), respectively (using the OpenCL API call `clGetEventProfilingInfo(...)`). A ratio $r'_k \in [0,1]$ of work items to distribute to the CPU is then determined using these measured execution times as follows:

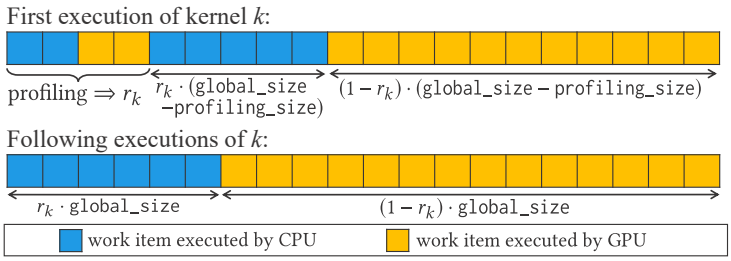


Figure 3.13: At the first execution of a kernel k, host-side profiling determines a ratio r_k to distribute work items

$$r'_k = 1 - (\text{timeCPU}/(\text{timeCPU} + \text{timeGPU}))$$

This ratio is slightly adjusted to obtain the final ratio r_k. Low percentages of work items on GPU showed to be detrimental to the performance compared not using it at all, while following executions on the GPU performed slightly better than the initial profiling in our experiments:

$$r_k = \begin{cases} 1, & r'_k > 0.8 \quad \text{(all CPU)} \\ \min(0, r'_k - 0.05), & \text{else} \quad \text{(mixed CPU/GPU)} \end{cases}$$

Finally, $r_k \cdot$ `global_size` and $(1 - r_k) \cdot$ `global_size` determine the amount of work items executed on CPU and GPU, respectively, for following executions of k (see Fig. 3.13, the values are rounded to multiples of the work group size). r_k is also used to distribute the remaining work items after profiling. The amount of work items to use for profiling is parameterized. In our experiments we achieved the best compromise between accuracy of the determined ratio and overhead of the profiling run when 50% of `global_size` was used for profiling when a kernel k was executed for the first time.

Figure 3.15: Speedup of the co-scheduling methods applied to Rodinia-SVM, on a fused CPU-GPU architecture with shared LLC. Results are relative to performing the optimal choice for each kernel of *either* executing on CPU *or* GPU (*xor-Oracle* is 100%)

3.7 Experimental Evaluation

The following results were obtained using a Intel Core i7-6700T (Skylake) fused CPU-GPU architecture with 32 GB of main memory. The Intel Core i7-6700T features a quad-core CPU and the HD Graphics 530 GPU. CPU and GPU share 8 MiB of last level cache (maximum for Skylake). All benchmarks were compiled using GCC version 7.2.1 and the Intel SDK for OpenCL Applications version 2017 R1. They were executed on CentOS Linux release 7.4.1708 with the Intel OpenCL 2.0 CPU/GPU driver package SRB5.0 (Linux kernel 4.7.0.intel.r5.0). To minimize execution time variance, hyper-threading was disabled and CPU frequency scaling set to 'performance' (which sets the highest frequency to all cores and effectively disables turbo boost). The Rodinia-SVM benchmarks were executed using the default inputs from the Rodinia Benchmark Suite for reproducible and comparable results. Results report the average of 10 executions of the respective benchmark with a standard deviation $< 2\%$ of the average, and do not include kernel compilation times[10].
In the following, we first show that device-side enqueuing causes too much overhead to be suitable for co-scheduling. Then, we evaluate our co-scheduling approaches, and finally show that cache coherency is a major performance bottleneck.

3.7.1 Device-Side Enqueuing

In this section we evaluate device-side enqueuing on a subset of the Rodinia-SVM benchmarks that result in the highest overheads when device-side enqueuing was applied. We execute the benchmarks in two versions: First we execute the kernels on the CPU only without applying any co-scheduling method. Then, we execute the benchmarks again with the device-side co-scheduling method of Section 3.6.2 applied (still CPU only). However, we immediately launch all work items (the total `global_size`) when the kernels are launched from the host-side API. Effectively, co-scheduling is never actually performed, i.e., the if statement in line 5 of Fig. 3.12 always evaluates to 'false', i.e., the postamble of the device-side enqueuing method is never executed.

Figure 3.14: Device-side enqueuing adds significant overhead, even when no kernel is enqueued. The overheads stem from the kernel call in the block syntax

Figure 3.14 shows the execution time increase of the device-side enqueuing method relative to execution without any co-scheduling method applied. Note that even though the co-scheduling code is not executed, the execution times increase significantly, up to almost $6\times$ for `sc`. The overheads disappear, as soon as we remove the kernel call from the block syntax in line 7 of Fig. 3.12 (e.g., by replacing `kernel(...)` with a `printf`). This means that runtime processing of the block syntax (capturing the environment) is performed even when that part of the code is not executed, and that it introduces high overheads, which render device-side enqueuing unsuitable for implementing co-scheduling methods. These results may surprise, but are in line with results published by Intel, where a naive port of an iterative implementation of *Sierpiński Carpet* to a recursive implementation using device-side enqueuing resulted in a $186\times$ execution time increase (2050ms instead of 11ms) [55]. Due to this cost, we exclude device-side enqueueing from further experiments.

3.7.2 Co-Scheduling Results of Rodinia-SVM

Figure 3.15 shows evaluation results for the co-scheduling approaches atomic counting and host-side profiling, and execution on CPU-only as well as GPU-only. The results are shown as speedups over the optimal per-kernel choice

[10] Kernel compilation can be avoided using `clCreateProgramWithBinary(...)`

of whether to execute the kernel either on CPU or GPU (clairvoyant xor-Oracle, see Section 3.3 for a discussion compared to a program-fixed ratio as determined by state-of-the-art approaches designed for fused CPU-GPU architectures without shared LLC). All speedups are relative to xor-Oracle (100%) and given in percent (of the relative performance achieved). The geometric mean (gmean) shows that on average execution on GPU-only performs worst (67.5%), mainly because two of the benchmarks (`mc` and `prtf`) perform very badly when their kernels are executed only on the GPU (they contain long-running loops). With 77.6% performance of xor-Oracle on average, execution on CPU-only performs better than GPU-only or with atomic counting. In other words, however, xor-Oracle on average achieves a 1.48× and 1.29× speedup over CPU-only and GPU-only, respectively, by using the most suitable compute device for each kernel.

When using both compute devices in parallel using the co-scheduling methods, one would expect to achieve a considerable speedup over the xor-Oracle that only uses one compute device at a time. As our results show, however, this is rarely the case (which we will analyze further in the following section). At best, atomic counting achieves 110.4% of xor-Oracle's performance (`hw`). On average it achieves 74.8% and thus performs better than GPU-only, but worse than CPU-only. One problem of atomic counting is that some kernels perform very badly on a particular device. Even when only a few work groups are launched initially, their execution times dominate the kernel's overall execution time (e.g., in `mc` and `prtf`). Additionally, atomic counting adds logic, and thus overhead, to the kernels itself.

Host-side profiling, on average, achieves 96.8% of xor-Oracle's performance and a speedup of 1.43× and 1.25× over GPU-only and CPU-only, respectively. It also performs considerably better on average than atomic counting (1.29× speedup), mainly because it only adds overheads to the very first kernel execution (when profiling) and does not add any code to the kernels. The overhead of profiling is especially evident in `md` that only executes a single kernel once, where host-side profiling performs worst over all benchmarks (64.9% of xor-Oracle). At maximum, host-side profiling achieves a 122.5% of xor-Oracle's performance in `hw`, but only in one other benchmark (`lc`) is another considerable performance benefit over xor-Oracle achieved (116.4%). Note that a host-side profiling implementation that tries to select the best device instead of distributing the work would incur similar overheads without any resulting speedups over xor-Oracle.

In summary, host-side profiling performs best over all methods and is on average competitive to the clairvoyant and thus hypothetical xor-Oracle. However, in most benchmarks it does not benefit from executing kernels on CPU and GPU in parallel compared to executing on the most-suitable single compute device, only.

3.7.3 Cache Performance Bottleneck

To analyze why executing kernels on both compute devices in parallel on fine-grained SVM does on average not provide a considerable performance benefit over executing the kernels on the most-suitable device only, we measured cache metrics using CPU-internal hardware performance counters. A subset of the Rodinia-SVM benchmarks was selected, for which host-side profiling was utilized to distribute work items to CPU and GPU for all kernels ($\forall k : 0 > r_k < 1$). These benchmarks potentially benefit most from utilizing both devices in parallel. Furthermore, the selected benchmarks synchronize with the host after each kernel execution (the same as in their original versions) which allows us to measure the performance counters for the kernel executions, only. We use the ratios r_k from the previous section for all kernels k, without performing the profiling step of the host-side profiling method.

Figure 3.16: Cache performance metrics (all levels, measured on CPU) when executing kernels in parallel on CPU and GPU relative to executing the same work item distribution sequentially (first on CPU, then on GPU; = 1 on y-axis)

First, all benchmarks are executed while using the devices sequentially, i.e., for each kernel we execute the work items assigned to the CPU first, synchronize with the host, and then execute the work items assigned to the GPU. For this *device-sequential* execution, the total cache misses and cache stalls (all levels) that are encountered by the

CPU are measured[11]. Then, all benchmarks are executed while using both devices in parallel (as in the previous section) and the same measurements are performed. In both measurements the CPU (and GPU) performs the same amount of work, but in the device-sequential case the CPU has more idle time.
Fig. 3.16 shows the measured cache metrics from the device-parallel execution relative to the device-sequential execution (= 1 on y-axis). For `hys`, `km` and `sc`, the cache misses do not increase (`hys` even benefits from device-parallel execution). This means that there are no cache conflicts like false or true sharing that impair the performance. However, the cache-related stalls increase considerably by up to 1.75× and 1.64× on average. A similar effect has previously been observed under simulation for cache-coherent fused architectures without a shared LLC [81]. The authors demonstrated that the amount of data probes sent by the highly-parallel GPU to the shared cache directory occupied the directory bandwidth which considerably slowed down the memory bandwidth that can be sustained by the cache hierarchy. Our results demonstrate the existence of a similar cache coherency bottleneck when fine-grained SVM is used on the Intel fused CPU-GPU architecture, even when CPU and GPU share an inclusive LLC. Further research is required to analyze and resolve this bottleneck (in software or hardware) to fully benefit from co-processing on fused CPU-GPU architectures.
For `hw` and `lc`, a similar increase in cache-related stalls cannot be observed. These results are in line with the speedup results shown in Fig. 3.15: `hw` and `lc` (group 1) benefit considerably from co-scheduling over the xor-Oracle, while `hys`, `km` and `sc` (group 2) do not. The main difference between these two groups of benchmarks is that the kernels of group 1 are considerably longer ($>$ 100 lines of code on average) than the kernels of group 2 ($<$ 30 lines of code on average). Therefore, the kernels of group 1 perform considerably more operations per work item than the kernels of group 2.

3.8 Conclusion and Implications for Predictable Execution

This work presented the first investigation of collaborative execution of computational kernels on a fused CPU-GPU architecture with a shared LLC using fine-grained SVM. We contributed two novel device-side co-scheduling methods that perform scheduling within the kernel code. It was shown that device-side enqueuing introduces considerable overhead stemming from the evaluation of the block syntax that is used in device-side enqueuing of kernels (up to 6× execution time increase), too much to be suitable for implementing co-scheduling methods.
Our host-side co-scheduling method achieved 96.8% of the clairvoyant and thus hypothetical xor-Oracle's performance on average (optimal per-kernel choice of exclusive CPU or GPU usage) and a speedup of 1.43× and 1.25× over execution on GPU only and CPU only, respectively. It also provided a 1.29× speedup over 'atomic counting', the best device-side co-scheduling method, because it does not add overhead to kernel execution once profiling is done. This makes our host-side co-scheduling method the most competitive practical scheme to date. We further showed that cache coherency is the major performance bottleneck in current fused CPU-GPU architectures with a shared LLC. It was shown that when CPU and GPU execute kernels in parallel on an Intel architecture, cache-related stalls observed on the CPU can increase by up to 1.75× while cache misses remain the same compared to executing the same work on the CPU and only then on the GPU (while the CPU is idle).
However, some benchmarks benefited considerably from collaborative execution on CPU and GPU (up to 1.23× speedup) compared to using the most suitable device. It depends on the memory access patterns of the kernels whether cache coherency becomes a performance bottleneck or not. In future work, it will be crucial to categorize the memory access patterns of kernels and design optimizations to alleviate this performance bottleneck for even more effective co-scheduling of kernels on fused CPU-GPU architectures. It becomes evident that the trend of processor integration in high-performance architectures is a two edged sword: it can eliminate data transfers to private memories of heterogeneous compute devices and enable co-computation of kernels by, e.g., CPU and GPU, resulting in a high performance within a limited power and are budget (which is crucial, e.g., for embedded

[11] There are no publicly documented interfaces to access Intel GPU performance counters when not using OpenGL

systems). At the same time, the potential for resource conflicts (and the complexity thereof) increases. While these conflicts can most certainly be resolved for average-case performance, it will be more challenging for future research to resolve them for predictable performance. The presented cache coherency bottleneck adds a shared last level cache between CPU and GPU to the growing list of microarchitectural features that can benefit average-case performance, but lead to resource conflicts of such a complexity that they are virtually infeasible to analyze for execution time guarantees.

This chapter presented novel co-scheduling approaches for fused CPU-GPU architectures in a case study on how performance is achieved in an off-the-shelf platform. It provided further evidence that high-performance architectures, which were designed for average-case performance, are not suitable for hard real-time systems that require execution time guarantees. Thus, the following chapters take a different approach to obtain *predictable* performance and base on a system that is already amenable to WCET analysis. As motivated in Chapter 1, such a system lags years behind current platforms like the one discussed in this chapter in terms of its architectural design. The focus of the following chapters will therefore be to achieve high performance *and* WCET guarantees by introducing runtime-reconfigurable accelerators.

4 Runtime Reconfiguration under WCET Guarantees

The target of this[1] and the following chapters is to achieve timing-analyzable performance by employing hardware accelerators that speed up the tasks' most compute-intensive parts, so called *computational kernels* (also known as hotspots) that are comprised of one or more nested loops. When implementing these accelerators as application-specific integrated circuits, the system would lack flexibility with respect to revised standards or new algorithms. Instead, using a runtime-reconfigurable architecture (which employs an FPGA, see Section 2.3) maintains a high flexibility and even allows for reconfiguring the accelerators at runtime, thereby increasing the performance as well as the computing efficiency (compared to a static set of accelerators) at the cost of a more complex timing analysis. The aim of this chapter is to enable guaranteed reconfiguration delays for configuring accelerators onto the reconfigurable area (which were previously unavailable). The following chapters will base on the guaranteed reconfiguration delays to achieve guaranteed WCETs of tasks that employ runtime reconfiguration of accelerators. Existing work on runtime reconfiguration in the context of real-time systems implicitly *assumes* that the process of reconfiguration itself complies with timing guarantees [16, 27, 29, 36, 93], e.g., the time it takes to configure a hardware accelerator on the reconfigurable fabric (*reconfiguration delay*) is assumed constant and free from conflicts with other system components that could affect WCET guarantees. The realization of a runtime reconfiguration controller that fulfills these assumptions and that is amenable to WCET guarantees is so far unavailable. However, guaranteed reconfiguration delays are crucial to realize runtime-reconfigurable real-time systems.
The novel contributions of this chapter are as follows:

- A runtime reconfiguration controller called "Command-based Reconfiguration Queue" (CoRQ) that provides guaranteed latencies for its operations and supports timing analysis for WCET guarantees. It was released as an open-source project, including examples and benchmarks[2].
- We show that conflicts while accessing a shared main memory during reconfiguration can lead to a slowdown of more than 21× in reconfiguration bandwidth. In contrast, CoRQ guarantees constant reconfiguration delays even under heavy system bus load.

4.1 Challenges for a Guaranteed Reconfiguration Delay

A straight-forward approach of improving WCET guarantees of a kernel using runtime reconfiguration with the constraints of timing-analyzability and reasonable implementation effort is the *stalling* approach (which will be detailed in Chapter 5). Software-only execution, i.e., without any accelerators, is shown in Fig. 4.1 (top). As shown in Fig. 4.1 (middle), a task that reconfigures an accelerator using stalling, stalls its execution for the whole reconfiguration delay. At most one reconfiguration can be performed by the reconfiguration port at any time. Once all reconfigurations have completed (at the end of the reconfiguration delay, Fig. 4.1 (a)), the task proceeds execution in software and executes the reconfigured hardware accelerators in every iteration of the kernel. A task that requests reconfiguration of accelerators using stalling can be analyzed for WCET guarantees using established timing analysis techniques by adding the reconfiguration delay (see Fig. 4.1 (a)) to the WCET of the basic block that requests the reconfiguration. The assumption is that the reconfiguration delay can be determined statically, which is reasonable for the stalling approach because the task's memory accesses and reconfiguration cannot interfere on

[1] The work presented in this chapter was originally published in [28]
[2] Available at: `https://git.scc.kit.edu/CES/corq`

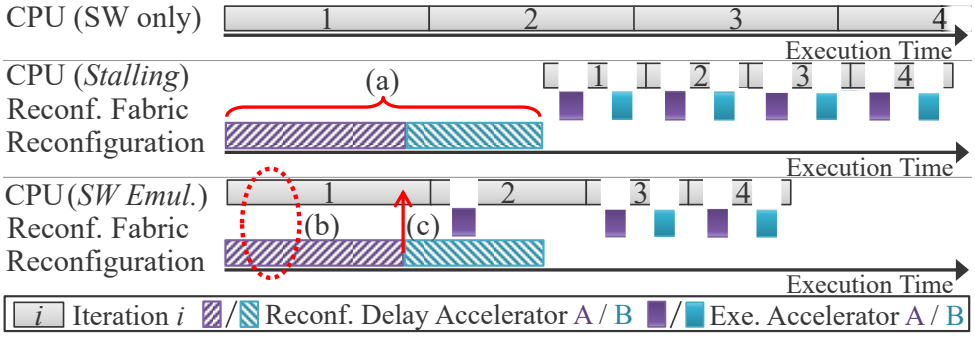


Figure 4.1: Timelines of executing a Kernel using Software only, Stalling and Software Emulation

main memory or a shared system bus. However, stalling is not state-of-the-art in reconfigurable systems, because the CPU remains idle during reconfiguration.

An approach that enables the CPU to perform useful operations in parallel to reconfiguration is *software emulation*, i.e., (1) accelerators are configured as early as possible in the control flow graph (CFG) and execution proceeds in parallel so that a considerable amount of reconfiguration delay has already passed at the point in time when the accelerators are actually needed and (2) in case execution of an accelerator is requested that is not yet configured, functionally-equivalent software is executed (see Fig. 4.1 (bottom)). Software Emulation is an established technique in average-case optimizing reconfigurable systems, because it provides considerable performance improvements. For real-time systems, however, software emulation poses new challenges:

- As memory transfers can be initiated simultaneously by the reconfiguration of accelerators and by tasks running on the CPU (see Fig. 4.1 (b)), it needs to be ensured that assumptions about memory access delays during static timing analysis of the guaranteed WCET bound (see Section 2.2) capture potential conflicts on main memory or a shared system bus.

- Even when the reconfiguration delay of an accelerator would be a statically-known constant value, the worst-case state of the task's execution on the CPU is unclear: how far did the task proceed (in the worst case) during the reconfiguration delay? In other words, from what point is it safe to assume during static timing analysis that, e.g., Accelerator A is readily configured on the reconfigurable fabric and available to be invoked (see Fig. 4.1 (c), this question will be the focus of Chapter 5)? Usually, reconfiguration of multiple accelerators is requested at once (but configured sequentially). The information that a specific accelerator has been reconfigured and is available to speed up execution should be obtainable by the task without interrupts that would complicate timing analysis.

- If program execution is faster or takes a different path than the worst-case path, a reconfiguration request could become obsolete because the requested accelerator will not be executed anymore (see Fig. 4.2). In real-time systems, the possibility of an already occupied reconfiguration port can lead to delays that are hard to analyze and therefore introduce pessimism in the resulting WCET bound. Therefore, it is crucial to be able to abort reconfigurations such that one can guarantee for each reconfiguration request that the reconfiguration port is unoccupied.

It might seem that the stalling approach is the favorable way to perform reconfiguration in real-time systems due to the potentially complex analysis of software emulation. However, stalling poses similar challenges for timing analysis when scheduling multiple real-time tasks, even on a uniprocessor system: when a task that requests a reconfiguration is stalled, another task could be executed in parallel to the reconfiguration delay (see Fig. 4.1 (a) and [16]). Concerning the resulting WCET bound by analyzing either approach, it will be shown in Chapter 5 that software emulation always provides a considerable speedup at runtime, but there are cases where additional WCET overestimation compared to stalling diminishes the speedup on the WCET guarantee. Which approach

Table 4.1: CoRQ Commands with Cycles spent in EXE State

Command	**Im**mediate/**Qu**eueable	$\text{latency}_{\text{EXE}}$ [1]
`clearQ`	Im, Qu	5
`stopQ`	Im, Qu	0
`resumeQ`	Im	0
`abortReconf`	Im	5
`setBaseAddr`	Qu	1
`configBitsExt` [2]	Qu	—
`configBitsInt` [2]	Qu	$6+\lceil B/4 \rceil$
`stallCPU`	Qu	1
`unstallCPU`	Qu	1
`sendGPIO`	Qu	1
`sendIRQ`	Qu	1

[1] discussed in Section 4.2.1 [2] detailed in Section 4.2.2, B - size of bitstream [byte]

is beneficial, eventually depends on several parameters of the accelerated kernel (e.g., reconfiguration delay and speedup of the accelerators employed) and therefore both approaches should be supported by a reconfiguration controller for real-time systems. In the following section, we will introduce our reconfiguration controller *CoRQ*, and explain how it addresses the challenges that we observed and supports the stalling approach as well as software emulation in a predictable way.

4.2 Enabling Runtime Reconfiguration in Real-Time Systems with CoRQ

The focus of our reconfiguration controller *Command-based Reconfiguration Queue* (CoRQ) is to enable the CPU to issue sequences of reconfiguration requests, provide guaranteed reconfiguration delays and relieve the CPU from managing accelerator availability. CoRQ provides commands to inform the CPU of finished reconfigurations in a predictable way; the CPU never has to poll or be interrupted to obtain the information that an accelerator has become available (following a reconfiguration). CoRQ processes 32-bit commands and can be instantiated with an internal memory to store *bitstreams* (configuration data for the reconfigurable fabric). Commands are issued by the CPU using load/stores over the system bus (see Fig. 4.3). They are either executed immediately or enqueued in an internal FIFO queue (denoted as *immediate* or *queueable* commands, respectively, in the following). Table 4.1 shows all 11 currently supported commands grouped by category (immediate or queueable). The immediate commands are used to control CoRQ itself (stop/resume processing enqueued commands, clear queue, reset) and abort a running reconfiguration. Queueable commands relieve the CPU from managing reconfigurations, i.e., they configure bitstreams (from internal or external memory), provide information about available accelerators through a general-purpose interface (or send an interrupt to the CPU), and can even stall/unstall the CPU to implement the stalling approach. In the following we illustrate how stalling and software emulation can be realized with CoRQ.

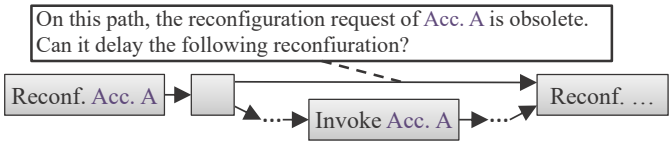


Figure 4.2: Control-flow graph that shows how one reconfiguration request can delay a following reconfiguration, thus impairing timing analysis

Reconfiguring a single accelerator in the stalling approach (see Section 4.1, Fig. 4.1 upper timeline) can be per-

```
1  stopQ          (Im: Stop processing commands from queue)
2  stallCPU       (Qu: Stall CPU)
3  setBaseAddr    (Qu: Set main memory base address)
4  configBitsExt  (Qu: Reconfigure from main memory)
5  sendGPIO       (Qu: Reset accelerators)
6  unstallCPU     (Qu: Unstall CPU)
7  resumeQ        (Im: Process enqueued commands)
```

Listing 4.1: CoRQ commands used to realize the stalling approach

```
1  clearQ         (Im: Ensure command queue is empty)
2  abortReconf    (Im: Ensure free reconfiguration port)
3  configBitsInt  (Qu: Reconfigure from internal memory)
4  sendGPIO       (Qu: Store info 'Accelerator A available')
5  configBitsInt  (Qu: Reconfigure from internal memory)
6  sendGPIO       (Qu: Store info 'Accelerator B available')
```

Listing 4.2: CoRQ commands used to realize the software emulation approach

formed using the sequence of commands shown in Listing 4.1. First, processing commands from the queue is stopped (immediately), otherwise the following command (Line 2) would stall the CPU before the `unstallCPU` command could be enqueued. All following commands (including stall CPU) are queueable. Assuming that the main memory is idle while stalling the CPU, one can use it to configure bitstreams even under timing guarantees. First, the base address of the bitstream is set and then `configBitsExt` instructs CoRQ to configure a bitstreams relative to this base address (Lines 3 and 4). This way, the whole 32-bit address space can be addressed using 32-bit wide commands. Afterwards, `sendGPIO` is executed to trigger CoRQ's GPIOs, e.g., to reset the configured accelerator and ensure it is in a sane state before using it. Once these commands are processed, the CPU is resumed. Finally, `resumeQ` (Line 7) is used to start processing the enqueued commands.
A reconfiguration of two accelerators while utilizing software emulation (executing software in parallel, see bottom timeline of Fig. 4.1) can be performed using the commands shown in Listing 4.2. In this case, neither processing queued commands is stopped nor the CPU is stalled. Therefore, the CPU proceeds executing software after issuing the commands to CoRQ. In this example we assume that a previous reconfiguration request could still occupy the reconfiguration port and obsolete commands could be in the queue (see Fig. 4.2). To be able to guarantee the reconfiguration delay, it needs to be ensured that no earlier reconfiguration requests are still pending. Therefore, at first all remaining commands are cleared and reconfiguration (if any) is aborted (Lines 1 and 2). Afterwards, a bitstream from internal memory is configured. This way, loading the bitstream does not conflict with memory accesses from the CPU to main memory. Once reconfiguration completes, `sendGPIO` is executed (Line 4) to notify the CPU that the first accelerator has become available. This can be done by writing into a memory-mapped register that the CPU can read or by writing to a lookup table that is automatically queried before executing an accelerator. This enables the CPU to use each configured accelerator immediately once it is configured (see Fig. 4.1 (bottom)), without waiting for the whole set of commands to have finished processing by CoRQ. Afterwards, a second accelerator is configured (Lines 5 and 6).
These two examples illustrate that stalling as well as software emulation can be realized by using CoRQ with simple sequences of commands issued by the CPU.

4.2.1 Command Execution

CoRQ processes commands using a finite state machine (FSM) consisting of three states: Fetch from queue (FE), Decode (DEC) and Execute (EXE) (see Fig. 4.4). Fetching a command takes a single cycle, the DEC state takes

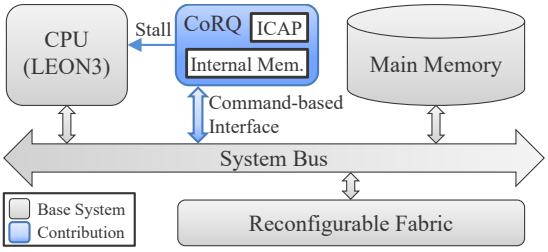


Figure 4.3: Example of how CoRQ is attached to a System on Chip to enable runtime reconfiguration under timing guarantees

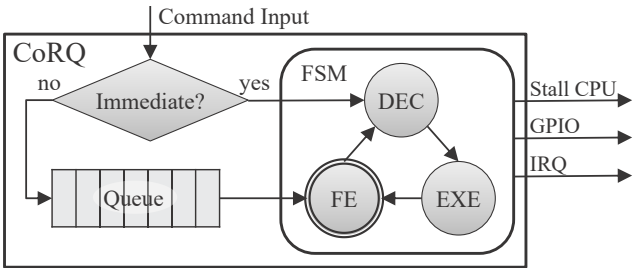


Figure 4.4: High-level view of how CoRQ processes commands

two cycles and the latency of EXE depends on the command (see Table 4.1). Immediate commands control CoRQ itself, and thus have priority over commands from the queue. After being identified as immediate (which takes one cycle), these commands reset the FSM (potentially aborting a queueable command in EXE) and directly enter DEC. In sum, executing either an immediate or a queueable command takes $3 + \text{latency}_{\text{EXE}}$ cycles.
Enqueueing a command takes 2 cycles for identifying it as queueable and writing it to the queue. Commands can simultaneously be enqueued to and fetched from the queue. The realization of this simultaneous access (with a double-ported FIFO) incurs an additional delay of 2 cycles for commands to become visible to the FSM if the FIFO was empty.

4.2.2 Guaranteed Reconfiguration Delay

It is possible to load bitstreams from arbitrary addresses, however, accessing the system bus and a shared main memory (especially DDR) can incur memory access delays that are hard to bound for WCET guarantees. Therefore, guaranteeing reconfiguration delays when using CoRQ-external memory (`configBitsExt`) is outside the scope of this thesis. Reconfiguration delays are guaranteed when the CoRQ-internal memory is used (`configBitsInt`). The CoRQ-internal memory is implemented using SRAM (so-called Block RAMs on Xilinx FPGAs). This way, the `configBitsInt` command can feed one word of the bitstream in each cycle to the reconfiguration port and utilize its full bandwidth (see Section 4.3). Additionally, the `configBitsInt` command requires 5 setup cycles and a single cycle at completion. Thus, $\text{latency}_{\text{EXE}} = 6 + \lceil B/4 \rceil$ cycles, with B being the size of the bitstream in bytes (see Table 4.1). Including the latency of FE and DEC, *configuring a single bitstream from CoRQ-internal memory (`configBitsInt`) is guaranteed to take exactly* $9 + \lceil B/4 \rceil$ *cycles*.

4.2.3 Analyzing Sequences of Commands

In the examples of Section 4.2, the latency of command sequences is simply the sum of the latencies of the queueable commands: configuring a single bitstream from main memory using stalling (see Listing 4.1 and Fig. 4.1 (middle)) results in a latency of $t_{\texttt{stallCPU}} + t_{\texttt{setBaseAddr}} + t_{\texttt{configBitsExt}} + t_{\texttt{sendGPIO}} + t_{\texttt{unstallCPU}} + t_{\texttt{resumeQ}} = 4 + 4 + t_{\texttt{configBitsExt}} + 4 + 4 + 3 = 19 + t_{\texttt{configBitsExt}}$ cycles. This is the latency after `resumeQ` reaches CoRQ. At this point the immediate command `stopQ` was already executed (taking 3 cycles) in parallel to filling the queue with commands, and therefore it does not add to this latency. As mentioned before, we do not provide guarantees for `configBitsExt`.
Configuring two bitstreams using software emulation (see Listing 4.2 and Fig. 4.1 (bottom)) results in a latency of $t_{\texttt{clearQ}} + t_{\texttt{abortReconf}} + t_{\texttt{configBitsInt}} + t_{\texttt{sendGPIO}} + t_{\texttt{configBitsInt}} + t_{\texttt{sendGPIO}} = 8 + 8 + (9 + \lceil B_1/4 \rceil) + 4 + (9 + \lceil B_2/4 \rceil) + 4 = 42 + \lceil B_1/4 \rceil + \lceil B_2/4 \rceil$. This latency starts once the immediate command `clearQ` reaches CoRQ and is running in parallel to the CPU that sends the commands following `clearQ` to CoRQ. Executing previous commands always takes at least as long as the delay for enqueueing the current command, therefore enqueueing the commands does not add to the delay. If it can be guaranteed that there are no pending reconfigurations, `clearQ`

Table 4.2: Ressource Utilization

	LUTs	FlipFlops	BRAM
LEON3 CPU (standard config.)	8,144	3,450	14
CoRQ	398	546	1
Internal Mem. of CoRQ (384 KB)	233	6	96
Available on VC707	303,600	607,200	1,030

and `abortReconf` can be omitted. In this case, enqueueing the first command to the empty queue would incur an additional latency of 4 cycles, resulting in a total delay of $30 + \lceil B_1/4 \rceil + \lceil B_2/4 \rceil$ (see Section 4.2.1).

4.3 Experimental Evaluation

We implemented a synthesis flow for partial reconfiguration and evaluated CoRQ based on a Gaisler LEON3 design (GRLIB GPL 1.4.1, also see Fig. 4.3) targeting the Xilinx VC707 board (Virtex-7 FPGA)[3]. We used a LEON3 design provided by Gaisler that instantiates a single LEON3, uses the DDR3 on the VC707 as main memory and runs at 100 MHz. CoRQ was added to the AHB system bus and a signal was connected to the LEON3 to enable stalling, no further changes were made to the SoC. For evaluation purposes, we simply reconfigure patterns of flashing the VC707's LEDs. The resource utilization is shown in Table 4.2.

The reconfiguration port (called *Internal Configuration Access Port* (ICAP) in Xilinx devices) can process 4 byte each cycle at maximum 100 MHz on the VC707. Therefore, the theoretical maximum reconfiguration bandwidth is 381.47 MiB/s[4] . We reconfigure 25 partial bitstreams of $B = 57,248$ bytes each, which together takes a minimum of 357,800 cycles when assuming the theoretical maximum reconfiguration bandwidth without overheads. Using CoRQ, these reconfigurations take 358,036 cycles[5] which corresponds to a reconfiguration bandwidth of 381.22 MiB/s. This means that CoRQ is only 0.066 % (or 236 cycles) slower than the theoretical maximum.

In the following we evaluate the impact of system bus conflicts on the reconfiguration bandwidth. Figure 4.5 shows the reconfiguration bandwidth results as measured by the CPU. Note that measuring itself adds an overhead, therefore, the measured reconfiguration bandwidth is always lower than the analytical bandwidth of CoRQ. The results were obtained for reconfigurations using the CoRQ-internal memory (Int. Mem.), as well as main memory over the shared AHB system bus (Main. Mem.). 'Stalling' leaves the CPU idle during reconfiguration, whereas 'Polling' means that the CPU repeatedly reads CoRQ's status register to check whether reconfiguration has completed (producing traffic on the AHB). 'Bus Load' uses a simple DMA unit that repeatedly initiates maximum length (256 words) AHB burst transactions to provoke system bus and main memory conflicts during reconfiguration. The small variance in measurements when using CoRQ-internal memory ($< 1\%$) stems from the overhead of measuring. The CPU's bus accesses (e.g., for fetching instructions and reading the timers) conflict with the DMA. CoRQ's commands itself always have exactly the same latency when using internal memory.

When reconfiguring over main memory, accesses from CPU, the DMA and CoRQ are in conflict. We can observe a strong variance in reconfiguration bandwidth between the measurements. The measurement under DMA bus load reports only 4.69% of the Stalling bandwidth. This shows that reconfiguration controller design is crucial in runtime-reconfigurable real-time systems. Simply utilizing a shared memory for reconfiguration can lead to a slowdown of more than $21\times$ in reconfiguration bandwidth.

[3] Project incl. benchmarks available at: `https://git.scc.kit.edu/CES/corq`
[4] More precisely: $(4 \cdot 1024^{-2})/10^{-8} = 381.4697265625$ MiB/s (= 400 MB/s)
[5] Sum of latencies of the individual commands (see Section 4.2): $4 + 25 \cdot (9 + \lceil 57,248/4 \rceil) + 4 + 3$ cycles
[6] Average of 50 measures, maximum error $< 1\%$

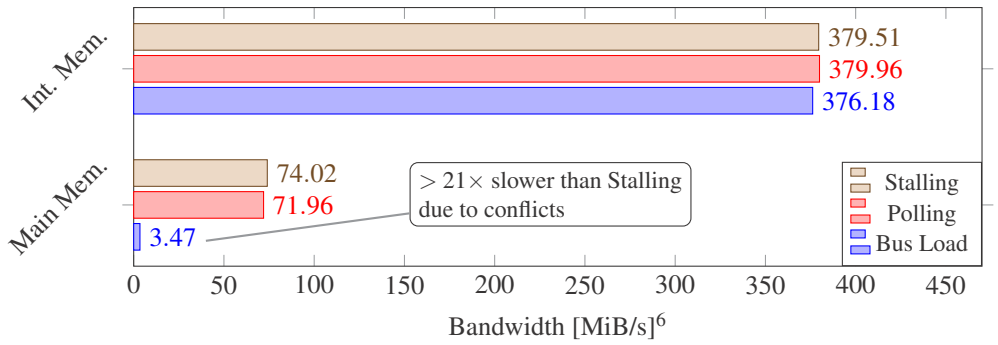


Figure 4.5: Reconfiguration bandwidth measured by the CPU, revealing a high variance when using main memory

4.4 Conclusion

This chapter discussed challenges for timing-analyzable runtime reconfiguration in systems that require WCET guarantees. It presented how these challenges can be addressed, and introduced CoRQ: a reconfiguration controller for real-time systems that provides guaranteed reconfiguration delays for the stalling and software emulation approaches. In the work presented in [88], CoRQ formed the basis to design a reconfiguration controller that enables preemptable runtime reconfiguration in Xilinx FPGAs, i.e., reconfigurations can be preempted (keeping reconfiguration progress) to avoid priority inversion in the presence of multi-priority real-time tasks instead of being aborted (and loosing the progress made so far). It was used to design a Xilinx Zynq-based multi-priority real-time system, where tasks of different priority levels can request reconfigurations.
The following chapter bases on the reconfiguration delay guarantees provided by CoRQ and introduces an analysis that enables reconfiguration of accelerators that speedup WCET guarantees in a runtime-reconfigurable processor.

5 WCET Analysis of Tasks on Runtime-Reconfigurable Processors

To escape the scarcity of timing analyzable performance features that was motivated in Chapter 1, this chapter[1] introduces timing analysis of tasks on runtime-reconfigurable processor designs in which the *core instruction set architecture* (cISA) of the processor core is extended by *custom instructions* (CIs). These CIs initiate the execution of accelerators on the reconfigurable fabric. Figure 5.1 shows a system with a reconfigurable instruction set processor (generalized from [12, 47, 102] and Section 2.4). It consists of an in-order RISC CPU with a reconfigurable fabric, scratchpad memory (SPM) and separate data and instruction caches (D$ and I$). With commercial platforms like the Xilinx Zynq SoC, which couples an ARM Cortex A9 multi-core with a Xilinx reconfigurable fabric on a single chip, reconfigurable processors have become off-the-shelf devices. In contrast to these, we specifically choose an in-order pipeline to be able to obtain precise execution time bounds using state-of-the-art static timing analysis. Predictable performance is achieved with our novel models for analyzing tasks that utilize reconfigurable CIs. These processor designs enable speedups even for applications that contain kernels with only short execution times in the range of 10–100 cycles when running on the fabric. CIs were detailed in the context of the reconfigurable processor *i*-Core in Section 2.4, their most-important properties for the context of this chapter are summarized as follows: CIs are specified by (multi-cycle) datapaths, which are implemented as configurations on the reconfigurable fabric. A configured CI takes a certain area share of the fabric. The fabric can accommodate several CIs at once, constrained by its total area (see [92] for an overview of area models). The time required for reconfiguring a CI at runtime depends on the size of the configuration and is called *reconfiguration delay* (it can take several milliseconds); a CI ready for execution is referred to as *available*. A CI which is not yet readily configured or was replaced by another CI is *unavailable*. In contrast to cISA instructions, a CI can be unavailable when it is due for execution. As introduced in the previous chapter, two common approaches exist to deal with this problem in reconfigurable processors, which optimize the average case on a best effort basis (see Fig. 4.1): *stalling* [48, 108], i.e., halting execution until the pending reconfiguration finishes, and *software emulation* [11, 26], i.e., branching to CI-equivalent software which can be executed on the cISA (see Fig. 5.2).
The main contribution of this chapter is a timing analysis for environments in which faster paths (e.g., containing hardware-accelerated CIs) through a kernel body become successively available during execution of the kernel (e.g., software emulation is utilized for unavailable CIs and reconfigurations are performed in parallel). A reconfig-

[1] The work presented in this chapter was originally published in [29]

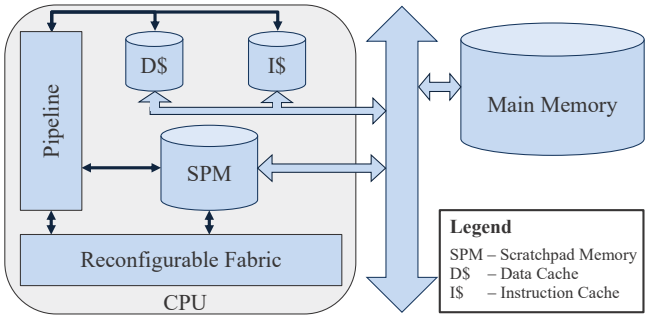


Figure 5.1: System on Chip with a reconfigurable processor

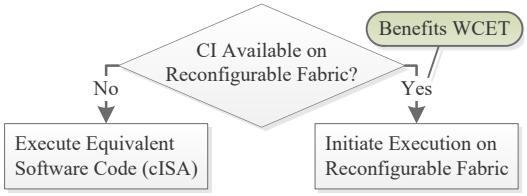


Figure 5.2: Software Emulation entails testing whether a specific reconfigurable CI is currently configured (*available*).

urable processor design amenable to this timing analysis is presented, and it is shown how the resulting information can be utilized to obtain precise WCET bounds of tasks.
The novel contributions of this chapter are as follows:

- Timing analysis of tasks on a reconfigurable instruction set with support for multiple execution contexts and comparison of measures to deal with reconfiguration delays: *stalling* the core pipeline or running equivalent software code until configurations finish (*software emulation*). To our knowledge, this is the first time that runtime reconfiguration is supported in an analysis for WCET guarantees.
- Identification and analysis of a *timing anomaly* of runtime reconfiguration, i.e., a situation where executing iterations of a kernel faster than worst-case time during reconfiguration can extend the execution time of the whole program. The timing anomaly is safely bounded during timing analysis.
- Description of key requirements to design reconfigurable instruction set processors that support timing guarantees.

We argue that a reconfigurable instruction set gives application designers more control over the microarchitecture than a fixed instruction set, which benefits timing analysis. Our evaluation results show that with precise analysis of proven reconfigurable processor design, runtime instruction set reconfiguration can be an enabling feature to provide timing-analyzable performance.

5.1 Related Work

Significant amount of work on reconfigurable instruction set processors has been performed. The demonstrated benefits are code size reduction, lower power consumption and increased average-case performance [42]. Current research in reconfigurable instruction set processors is moving towards heterogeneous reconfigurable multi-core architectures [24, 45, 50]. However, worst-case timing analysis of parallel tasks is still new ground even on general-purpose multi-core architectures [6, 90].

5.1.1 WCET-Optimizing Instruction Set Architectures

Little work on instruction set adaptation has been performed in respect to WCET. In [112] an instruction set selection for WCET optimization on an ASIP is performed. This approach performs WCET estimation using timing schema [76]. Timing schema uses a syntax tree representation of the program under analysis. Inner nodes represent the control flow and leaf nodes are the basic blocks of the program. The WCET is estimated in a bottom-up fashion with simple recursive rules. The proposed instruction set selection targets instruction set extension for applications known at design time. Reconfiguration is not considered in this approach.
MCGREP [104] is a two-stage pipelined, micro-programmed processor design without caches. Every instruction has a constant delay, independent of the execution history. The two ALUs of this processor design can be reconfigured to perform an application specific instruction in every cycle using a microcode to improve instruction throughput. MCGREP's design allows a straight-forward timing analysis without the requirement of complex models. However, its evaluation assumes a single-cycle load delay and compares against microprocessors at 40 MHz with their cache switched off. Requiring the absence of caches for timing predictability is too restrictive, as LRU caches are well understood in timing analysis and allow a predictable processor design with memory hierarchy [107]. Additionally, the two-stage pipeline may limit the frequency of the processor. The scalability of this design in performance-demanding scenarios remains questionable.
In [100] an integration of the real-time processor CarCore [101] and the MOLEN reconfigurable custom computing unit [102] is presented. The integration focuses on guaranteeing hard real-time constraints for memory accesses of processor core and reconfigurable hardware. Timing analysis of binaries utilizing reconfiguration is

not addressed. As a consequence, measured execution times are evaluated while the effects on WCET bounds remain uninvestigated.

5.1.2 Runtime Reconfiguration in Hard Real-Time Systems

If supported, runtime reconfiguration is the responsibility of the task scheduler in state-of-the-art hard-real-time systems [3, 23, 36, 54, 70], but reconfiguration within a currently executing task is not considered. Ref. [36] tackles the problem of scheduling access to the reconfiguration access port for fixed-priority task sets with hard deadlines. ReconOS [3] is an operating system that provides a unified programming model for threads running in software and threads mapped to reconfigurable hardware. Previous versions were based on the eCos[2] real-time kernel. Ref. [23] presents mapping and scheduling of task graphs to a uniprocessor system with reconfigurable units, which minimizes utilization of fabric area. Reconfigurations are either performed before, or as special tasks within the schedule. However, during execution of a task, its assigned reconfigurable unit is not reconfigured.
In [54] a per-task instruction set selection is performed using reconfiguration to meet timing constraints. A periodic task graph with deadlines is scheduled and the schedule is partitioned into configurations. Each configuration is assigned an instruction set to optimize the tasks' WCET. Their approach assumes that the schedule's WCET reduction directly corresponds to the per-task cycle reduction due to a CI that is chosen for a subset of the tasks. But this is only the case when there is no conditional execution of tasks. When having alternative execution paths, adding a CI into the current WCET path may result in another path becoming the WCET path. In this case, the gain of the CI on the overall WCET is the delta between the old and the new WCET path, which could be as small as 1 cycle, independent of the average performance improvement due to the CI. So the overall WCET gain can be significantly less than the gain in the old WCET path. A similar effect was already reported by [94] when assigning a variable to scratchpad memory for WCET reduction. Due to this effect, it is not possible to apply the inter-task techniques in [54] on intra-task level.
In sum, state-of-the-art techniques either do not consider runtime reconfiguration [112], introduce reconfigurable architectures without investigating effects on WCET bounds [100], or runtime reconfiguration is the responsibility of the real-time scheduler, where reconfiguration for an already running task is prohibited [3, 23, 36, 54, 70].

5.2 Motivational Example

As discussed in Section 5.1, state-of-the-art techniques only perform reconfigurations when switching from one task to another [3, 23, 36, 54, 70]. Within a single task however, the benefits of having a reconfigurable fabric are not exploited. To show the opportunities that are missed by such limitations, we use an H.264 video encoder as a motivational example in Fig. 5.3. For each input frame, the encoder goes through a sequence of kernels with different requirements of CIs to be configured onto the fabric. When configuring CIs for the whole task before it executes, the reconfigurable area has to be divided among the kernels (Figure 5.3 (a)). Performing reconfiguration *before* each kernel allows every kernel to use the whole fabric area at the cost of reconfiguration delay (Figure 5.3 (b)), i.e., the start of the kernel is delayed (*stalling*) until its reconfiguration is completed. As the total reconfiguration delay per task increases, the question whether the idle time of the CPU during reconfiguration (stalling) can be used more effectively is posed. Instead of waiting until the reconfiguration of all CIs for a kernel finishes, the kernel can be started immediately using *software emulation* (see Fig. 5.2 and Fig. 4.1). As soon as reconfiguration for a CI finishes, it is utilized within the kernel and the reconfiguration delay was effectively used to make progress (Figure 5.3 (c)). Software emulation leads to considerable runtime benefits at the cost of implementing equivalent software code for a CI. E.g., the runtime of `LoopFilter`, the shortest running kernel in the H.264 video encoder, is reduced by 20% using software emulation compared to stalling on a system running

[2] `http://ecos.sourceware.org/`

the reconfigurable fabric at 100 MHz, the core pipeline at 800 MHz and a reconfiguration bandwidth of 100 MB/s (see Section 5.6 for a detailed discussion).

Software emulation provides measurable runtime benefits which we make accessible for lower WCET bounds. A pessimistic timing analysis approach would simply assume an infinite reconfiguration delay and thus assume that every CI is executed in its cISA implementation for WCET estimation. While this would produce a safe WCET bound and enable speedups using CIs over cISA at runtime, the guaranteed WCET would be worse than not using reconfiguration at all. Therefore, precise modeling of reconfigurable CIs is required for timing analysis.

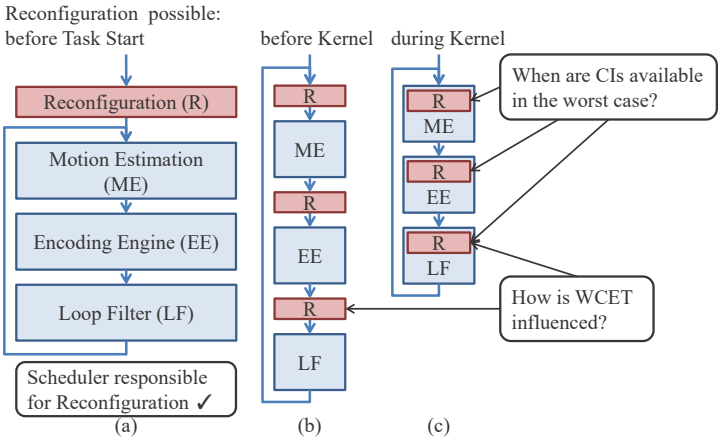


Figure 5.3: Sequences of kernels, e.g., in the H.264 Encoder, are well-suited for runtime reconfiguration, but raise new issues in timing analysis

In the following section, we introduce timing analysis fundamentals which target non-reconfigurable processors or the cISA of a reconfigurable processor, and form the basis for our novel extensions for analysis of reconfigurable CIs presented in Section 5.4.

5.3 Timing Analysis Background

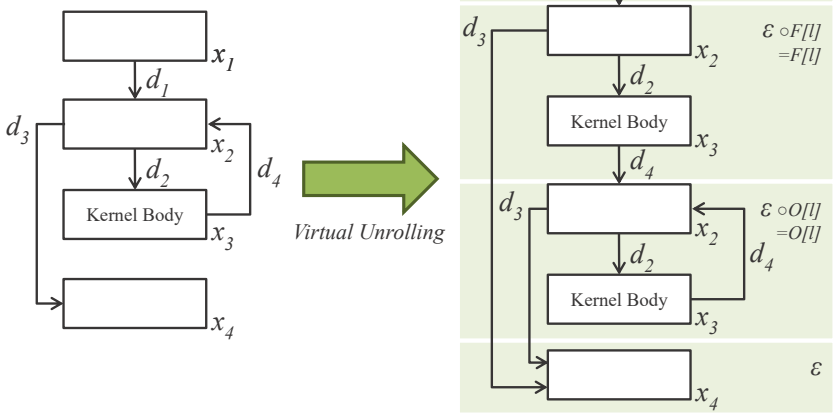
Timing analysis to derive WCET guarantees of tasks was introduced in Section 2.2. This section revisits global bound analysis using the Implicit Path Enumeration Technique (IPET) to provide additional details on multi-context path analysis as employed by the approach presented in this chapter. For computing guaranteed time bounds for tasks on a reconfigurable instruction set processor, an additional reconfiguration analysis pass that generates IPET constraints after the microarchitectural analysis is introduced in this chapter. When using software emulation and performing reconfigurations in parallel, timing analysis needs to determine the worst-case point in time at which a CI becomes available. The aim of our reconfiguration analysis is to provide information to the global bound analysis about when it is safe to assume a CI to be available, i.e., when to use the path that uses the hardware CI instead of equivalent software to effectively reduce the guaranteed WCET bound. In the following section, we introduce the background on IPET-based multi-context path analysis.

5.3.1 Path Analysis

As any ILP-formulated problem, global bound analysis using IPET consists of two parts: the objective function and its constraints (details in Section 2.2). For WCET analysis, the objective function determines the CPU cycles executed on a path in the task's control flow graph (CFG). To find the WCET path, it needs to be maximized. Variables in the objective function represent the execution count of a single basic block (x_i) in the CFG and are weighted with the execution cycles of that basic block (c_i), which is determined in the microarchitectural analysis. For a program with N basic blocks, the objective function is given as [64]:

$$\max_{x \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i x_i$$

The constraints restrict the variables by modeling the control flow and capturing relative execution counts of basic blocks. The more infeasible paths can be excluded with constraints, the tighter the WCET bound will get. *Program*



(a) Virtually Unrolling a Kernel

$$x_1 = d_1$$
$$x_2 = d_1 + d_4 = d_2 + d_3$$
$$x_3 = d_2 = d_4$$
$$x_4 = d_3$$

(b) Basic program structural constraints (single context)

$$x_3 \leq 200 \cdot d_1$$

(c) Program functionality constraint (single context) for an upper bound of 200 kernel iterations

$$x_1^{\varepsilon} = d_1^{\varepsilon}$$
$$x_2^{F[l]} = d_1^{\varepsilon} = d_2^{F[l]} + d_3^{F[l]}$$
$$x_3^{F[l]} = d_2^{F[l]} = d_4^{F[l]}$$
$$x_2^{O[l]} = d_4^{F[l]} + d_4^{O[l]} = d_2^{O[l]} + d_3^{O[l]}$$
$$x_3^{O[l]} = d_2^{O[l]} = d_4^{O[l]}$$
$$x_4^{\varepsilon} = d_3^{F[l]} + d_3^{O[l]}$$

(d) Program structural constraints after virtual unroll (multiple contexts)

Figure 5.4: IPET constraint generation for single contexts and multiple contexts after *virtual unrolling*

structural constraints can be derived automatically from the CFG, *program functionality constraints* need to be user-specified or provided by further analysis passes.

Basic Constraints

An overview of how to generate IPET constraints was given in Section 2.2, a brief example is described in the following. Besides the variables x_i representing the execution counts of basic blocks, variables d_i for execution counts of edges in the CFG are used. E.g., consider the loop kernel in Fig. 5.4 (a). The loop header (represented by x_2) can be entered from outside using the edge represented by d_1 or from a previous iteration using d_4. The same basic block can be left when the loop condition becomes false and the kernel is exited using d_3 or it can proceed to another iteration when the loop condition is true using d_2. Therefore, $x_2 = d_1 + d_4 = d_2 + d_3$ (see Fig. 5.4 (b)). An upper bound of 200 for the number of kernel iterations can be given by the program functionality constraint $x_3 \leq 200 \cdot d_1$ (see Fig. 5.4 (c)).

Execution Context

The simple constraints presented so far only consider a single execution *context*, i.e., only one abstract microarchitectural state is considered at the beginning of each basic block. For a specific basic block this context needs to include any additional delay that may occur on any path to this basic block. In a loop, e.g., the first iteration will encounter much more cache misses than following iterations, but the cache misses of the first iteration need to be taken into account for all the following iterations. Consequently, the resulting WCET of the task will be very pessimistic. Therefore, more fine-granular analyses considering different paths to reach a basic block separately have been developed [87, 98].

Following the definition in [98], a context is a sequence (denoted by * in the formula) of first (C) and recursive (R) calls of functions as well as first (F) and following/other (O) iterations of loops. The set $\mathcal{T}$ of all contexts for a program $\mathcal{P}$ is [98]:

$$\mathcal{T} := \{C[c], R[c], F[l], O[l] : c \in \text{calls}(\mathcal{P}), l \in \text{loops}(\mathcal{P})\}^*$$

With $\text{calls}(\mathcal{P})$ and $\text{loops}(\mathcal{P})$ being the sets of all calls and all loops in $\mathcal{P}$, respectively.

Analyzing Basic Blocks in Multiple Contexts

Contexts allow separate timing analysis of basic blocks for the different paths to reach them. Analysis starts with the empty context ε. When a function is called or a loop body is executed, then the context changes. It can be considered as a stack: when a function is called or a loop is entered, this information is appended using the '$\circ$' operator. Upon exit, the information is removed. Recursive calls and following loop iterations replace (first) calls or iterations on top of the stack, respectively. For example, a basic block inside a loop $l \in \text{loops}(\mathcal{P})$, which is reached by calling $c \in \text{calls}(\mathcal{P})$, entering l and executing it repeatedly would be in the context $C[c]O[l]$. A more detailed explanation can be found in [98], the theoretical background in [71].
In the CFG, the context influence of a loop is represented by *virtually unrolling* the loop once, as shown in Fig. 5.4 (a). Using this representation, multi-context IPET constraints can be generated as seen in Fig. 5.4 (d). In general, each basic constraint in Section 5.3.1 is generated for each distinguished context of the designated basic block. E.g., x_3 in Fig. 5.4 can be entered using d_2 and exited using d_3 in context $F[l]$ as well as $O[l]$. Additionally, the constraints capture context changes performed using the '$\circ$' operator, e.g., when entering, reentering or exiting a loop.

5.4 Timing Analysis Extensions for Runtime-Reconfigurable Processors

The main contributions of this chapter are the path analysis extensions to obtain precise WCET bounds of tasks on runtime-reconfigurable processors and are presented in Section 5.4.2. The following section introduces properties, which we assume the microarchitecture to provide and which we exploit during timing analysis.

5.4.1 Microarchitectural Analysis

The input to microarchitectural analysis (see Section 2.2) is the reconstructed CFG from the binary under analysis. Its output is a WCET bound per basic block and context (see Section 5.3.1) incorporating cache misses/hits, memory access latencies, pipeline stalls, and other delays which can occur in the system. To determine the WCET of a basic block including CIs, the CI latencies need to be known and possible influences on the rest of the microarchitecture –especially on core pipeline and caches– need to be accessible to the respective analysis passes. The delay for initiating and performing the reconfiguration of a CI needs to be analyzable statically. Additionally, reconfigurations in parallel to execution may not void any timing guarantees of the microarchitecture, e.g., the delay for the CPU to perform bus accesses. We assume to be able to initiate a sequence of CI reconfigurations using the core pipeline (e.g., by accessing a device on the bus) and then either use stalling or software emulation while reconfigurations take place. The requested CIs for an upcoming kernel and the order of configurations need to be obtainable by analyzing the binary. This information is used to generate constraints for path analysis as described in the following section.
An implementation of a *reconfiguration controller* which provides these properties is CoRQ, as presented in Section 5.5.

5.4.2 Path Analysis Constraints for Software Emulation

Using software emulation, CI functionality can be executed on two separate paths: the CI itself or functionally-equivalent software, depending on whether the CI is available or not (see Fig. 5.2). In the following we always initiate a sequence of reconfigurations of CIs immediately before entering a kernel. Kernel execution and reconfigurations are performed in parallel (see Figs. 4.1 and 5.6). With the reconfiguration delay for each CI known (in cycles of the core pipeline), we can determine the total delay for a CI to become available in the sequence. Once the CI is available, the program path that uses the CI will be taken for all of its invocations in the remaining kernel iterations. The main challenge is to precisely analyze at which point during kernel execution these path changes

will happen (from software emulation to hardware-accelerated CI) by CIs becoming available successively: How far will the kernel have progressed in the worst case when reconfiguration of a CI finishes and what exactly is the worst case?

Assumptions

For analyzing tasks that use reconfigurable CIs for worst-case timing with manageable complexity, we apply the following assumptions.

- We assume that software emulation of a CI always has a longer delay than the CI itself. In case the CI took longer than the software emulation, it would not make sense to use a CI anyway. The result of this assumption is that we know that software emulation is executed when the invocation of a CI lies on the worst-case path, unless the respective CI is explicitly annotated available (which results in a lower WCET).
- CIs currently executing in software emulation are never moved to hardware and we conservatively assume the availability of a CI not to change during a kernel iteration, even when reconfiguration finishes early in the iteration and the CI is the last executed instruction. This assumption eases analysis and is safe (the WCET of a single kernel iteration is reduced when a CI becomes available), but may introduce pessimism.

Worst-Case CI Availability

In the following, the aim is to bound the worst-case number of kernel iterations u_k for every CI k, in which the CI is unavailable. During these iterations, the CI invocations need to be executed using software emulation. To obtain a safe bound, we need to determine under which circumstances the execution time is maximized when a CI becomes available after its constant reconfiguration delay. For this, consider the timelines in Fig. 5.5 for a kernel with 6 iterations in total. Iterations without the CI available are yellow for executing faster than WCET and orange for executing them in WCET. After the marked configuration delay for the specific CI ("Reconfiguration Finish"), every following kernel iteration (green) makes use of the CI path (instead of software emulation) for all of its invocations (possibly multiple per iteration). Clearly, these iterations need to run in worst-case time to maximize the execution time after the configuration delay. The remaining questions to *maximize the total execution time* of the kernel are:

(i) Given an upper bound of kernel iterations. In the worst case, how many of these iterations are run after the reconfiguration delay?

(ii) What is a safe time bound for iterations that execute before finishing reconfiguration?

First, let us consider question (i) and assume *no iteration of the kernel can overlap* the point at which the configuration finishes. Let $\mathrm{WCET}_{\mathrm{avail}}$ be the WCET of one kernel iteration with the CI available, r the reconfiguration delay for the CI and m the remaining iterations after executing with software emulation during r. Under these assumptions, the execution time becomes:

$$\text{Execution Time} = r + m \cdot \mathrm{WCET}_{\mathrm{avail}}$$

As m is the only variable in this equation and every constant is positive, the equation is maximized when m is maximized. Under the assumption that no iteration of the kernel can overlap the Reconfiguration Finish, this is achieved by executing every iteration during r in worst-case time as depicted in Fig. 5.5 (b). Counterintuitively, this means that the execution time is *increased* when *fewer iterations* are executed with the CI unavailable. We call this property *minimum progress* (during the reconfiguration delay). To finally answer (i) and (ii), iterations overlapping the Reconfiguration Finish need to be considered. As the last iteration of the kernel with the CI unavailable can

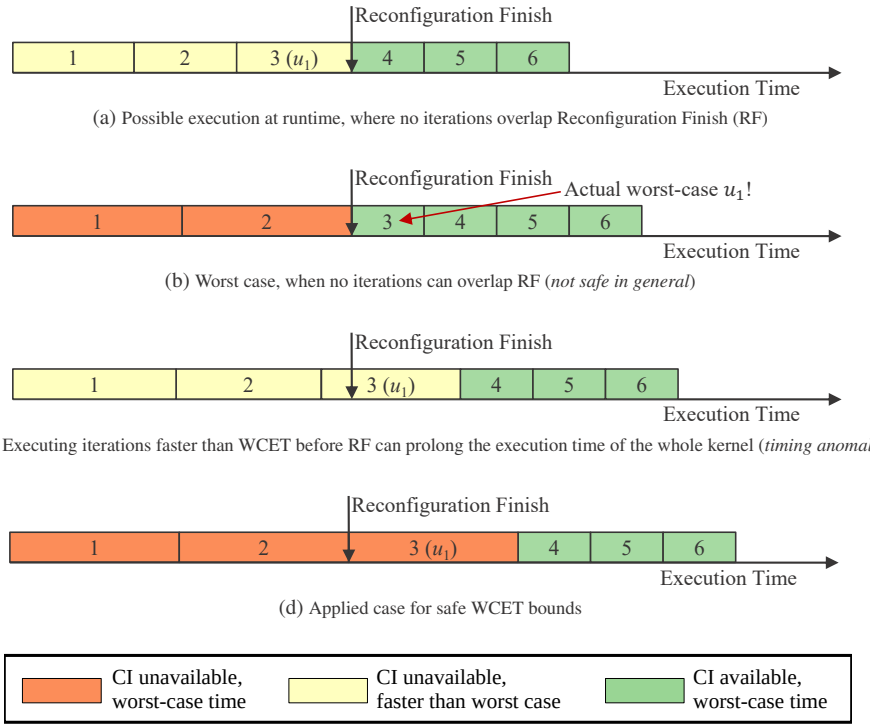


Figure 5.5: Different cases for execution times of kernel iterations. Executing all iterations in WCET does not necessarily bound the total WCET of the kernel, because the worst-case number of iterations in which CI_1 is unavailable (u_1) can be mispredicted (timing anomaly in (b)). For safe bounds, an additional iteration needs to be considered that assumes CI_1 unavailable (like in (d))

now reach into the part of the timeline in which the CI is already available, this iteration can prolong the execution time. This can lead to a case, where executing iterations faster than worst-case time during reconfiguration extends the execution time of the whole kernel. We call this the *timing anomaly of runtime reconfiguration*. An example is depicted in Fig. 5.5 (c). For bounding this timing anomaly, the case shown in Fig. 5.5 (d) needs to be applied in timing analysis, i.e., *maximum overlap* (of one kernel iteration).

Summarizing, worst-case execution during reconfiguration is safely bounded when minimum progress and maximum overlap are combined: All iterations are executed in worst-case time, as this results in minimum progress during the reconfiguration delay. An additional full iteration starting after the Reconfiguration Finish is assumed to execute with the CI unavailable in worst-case time for bounding a potentially overlapping iteration. To generate safe constraints this case is always applied to bound the timing anomaly at the potential cost of overestimation.

For a scenario with multiple CIs in a kernel, it can analogously be argued for the worst case of CI_i to become available after CI_{i-1} when considering the reconfiguration finish of CI_{i-1} as point zero on the timeline and accounting for the additional iteration (potential timing anomaly) of CI_{i-1}. In the following section we will formally express these considerations for multiple CIs.

Basic Constraints

First, the analysis of the worst-case number of iterations a CI is unavailable needs to be formalized. Then, IPET path constraints can be generated that model the information obtained from the analysis. Without loss of generality, suppose that the CIs for the following kernel are configured in the sequence $CI_1, \ldots, CI_n$. We denote r_i as the delay to configure CI_i. With existing timing analysis and the properties of Section 5.4.1, we can determine $WCET_i$, the WCET of one kernel iteration with $CI_1, \ldots, CI_{i-1}$ available (and $CI_i, \ldots, CI_n$ still unavailable) by modeling the CI availability using IPET path constraints. Consider the conditional branch to CI-equivalent software (see Fig. 5.2) in the case a CI should be invoked, but is unavailable. As shown in Fig. 5.6, for every CI invocation j in the binary,

the CI and its software emulation reside on separate paths in the CFG, which are immediately joined after the CI functionality is executed. Let x_{SW_j} be the variable representing the first basic block of the software emulation path of invocation j. A constraint of $x_{\mathrm{SW}_j} = 0$ is used to annotate the CI invoked by j to be available, because it forces the path analysis to exclude this path and account for the hardware CI path using x_{CI_j} only. For determining WCET_i, we generate a constraint for every invocation of $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ to exclude software emulation. WCET_0 is the special case of not generating any CI-specific constraints and effectively always executing the software emulation for every CI to be configured.

Suppose we would know u_k, the number of iterations in which CI_k is unavailable (and therefore CI_s, $s > k$ unavailable), but all previous CIs (if any) CI_t, $t < k$ are available. The total number of iterations CI_i is unavailable is $\sum_{k=1}^{i} u_k$. Let $u_0 = \mathrm{WCET}_0 = 0$, then we can define the remaining reconfiguration time needed for CI_i after CI_{i-1} became available as:

$$s_i := \sum_{k=1}^{i} r_k - \sum_{k=0}^{i-1} u_k \cdot \mathrm{WCET}_k \tag{5.1}$$

In other words, this is the time to configure $\mathrm{CI}_1, \ldots, \mathrm{CI}_i$ minus the time already spent in the first $\sum_{k=0}^{i-1} u_k$ iterations. Formally, we can define u_i recursively as follows:

$$u_i := \begin{cases} \lceil s_i / \mathrm{WCET}_i \rceil + 1, \text{ if } s_i > 0 \\ 1, \text{ if } s_i + \mathrm{WCET}_{i-1} > 0 \wedge s_i \leq 0 \\ 0, \text{ else} \end{cases} \tag{5.2}$$

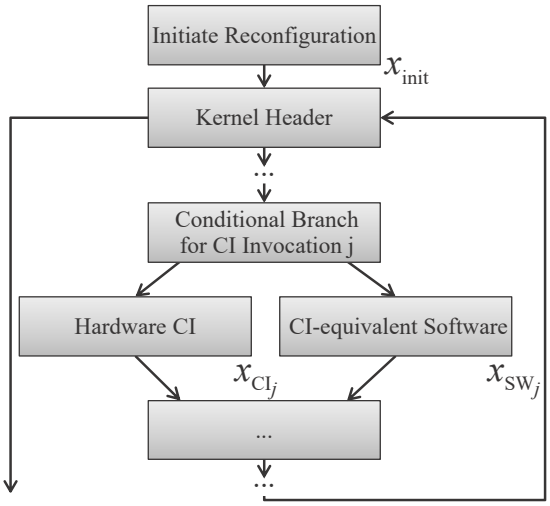


Figure 5.6: CFG of a Kernel invoking a CI with Software Emulation

If s_i becomes ≤ 0 (the second and third case of u_i), the time $\sum_{k=1}^{i} r_k$ until CI_i becomes available, is already covered by the time spent in the iterations in which $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ are unavailable. As discussed in Section 5.4.2, an additional iteration for iterations overlapping the point in time the reconfiguration finishes can become necessary, however. This is the case, if CI_i became available in the additional iteration of CI_{i-1} (the second case, e.g., iteration 3 in Fig. 5.5 (d)). If CI_i became available in the previous iteration (e.g., iteration 2 in Fig. 5.5 (d)), the additional iteration of CI_{i-1} already covers the additional iteration of CI_i such that CI_{i-1} and CI_i become available in the same iteration. It would be safe but pessimistic to not differentiate between the second and third case and always add an additional iteration. For a single CI_1, the equation becomes $u_1 = \lceil r_1 / \mathrm{WCET}_1 \rceil + 1$ (see Fig. 5.5 (d)).

Assuming no invocations of CI_i are contained in a nested loop inside the kernel, constraints can directly be generated that restrict the number of kernel iterations in which CI_i is unavailable in hardware and the software emulation needs to be executed. The limitation that CI invocations cannot be contained in nested loops will be removed in Section 5.4.2. For a single invocation j of CI_i inside the kernel, let $x_{\mathrm{SW}_{i,j}}$ be the variable representing the number of executions of the first basic block in CI_i's software emulation. Let x_{init} be the number of executions of the basic block which initiates reconfiguration before entering the kernel as shown in Fig. 5.6. Finally, include the previous reconfiguration analysis into global bound computation, the following constraint is generated:

$$x_{\mathrm{SW}_{i,j}} \leq \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}} \tag{5.3}$$

This constraint ensures that the software emulation of invocation j of CI_i ($x_{\mathrm{SW}_{i,j}}$) is accounted for at most as often in the worst-case path as CI_i was determined to be unavailable in iterations of the kernel under analysis, i.e., it adds information to the analysis that aims to reduce the estimated WCET bound. Even though we can determine the exact number of kernel iterations which start with CI_i unavailable, the constraint is generated as an inequality because it might happen that the worst-case path in the kernel does not even include invocation j of CI_i. Generating a constraint $x_{\mathrm{SW}_{i,j}} = \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}}$ with $u_i > 0$ would force the timing analyzer to include invocation j of CI_i in its path analysis, possibly hindering it from finding the real worst-case path.

Conditional Execution

In the following, pessimism in the analysis of conditional execution of CIs is removed. As discussed previously, the constraints generated by Eq. (5.3) correctly allow the timing analyzer to find worst-case paths not containing every CI invocation. However, for the upper bounds on how often the software emulation of a CI invocation needs to be executed (because the hardware is not yet available), the maximum possible number of iterations making use of this invocation is always considered. Even when the specific CI invocation is not part of the worst-case path for *all* iterations of the kernel. This may be very pessimistic: consider a CI which is unavailable for 10 iterations of a kernel in total. Furthermore, the CI is invoked only 5 times during the 10 iterations of its unavailability in the worst-case path. Then, the CI needs to be invoked only 5 times using software emulation during the whole execution of the kernel. So far however, the generated constraints force the analyzer to assume 10 invocations of the CI need to be executed in software emulation (the total iterations in which the CI is unavailable). As only 5 iterations exist in which the CI is invoked and unavailable, the constraints will force 5 iterations of the kernel to emulate the CI in software even though the hardware is already available. This pessimism can be removed by performing an extended analysis of WCET_i and determining for every invocation j of an unavailable $\mathrm{CI}_i, \ldots, \mathrm{CI}_n$ whether this invocation is part of the worst-case path, i.e., $x_{\mathrm{SW}_{i,j}} > 0$ (in the analysis of WCET_i). When generating the constraint for the number of software emulations of invocation j of CI_i, u_k is only added to the sum if this invocation is part of the worst-case path that defines WCET_k. We define:

$$\mathrm{inp}(i,j,k) := \begin{cases} 1, \text{ invocation } j \text{ of } \mathrm{CI}_i \text{ is part of the worst-case path of } \mathrm{WCET}_k \\ 0, \text{ else} \end{cases} \tag{5.4}$$

$\mathrm{inp}(i,j,k)$ is obtained from the solved ILP which was used to determine WCET_k by simply testing whether $x_{\mathrm{SW}_{i,j}} > 0$. For every invocation of a CI the exact number of times the software emulation is executed when entering the kernel in the worst-case path is obtained from this analysis. The following inequality defines the updated constraint that makes use of this information:

$$x_{\mathrm{SW}_{i,j}} = \sum_{k=1}^{i} \mathrm{inp}(i,j,k) \cdot u_k \cdot x_{\mathrm{init}} \tag{5.5}$$

This constraint is generated for every invocation of all CIs instead of the constraint in Eq. (5.3).

Loop Nests

As mentioned before, the constraints generated so far do not support CIs that are contained in loop nests. This limitation is removed in the following. When considering an invocation j of CI_i which is contained in a nested loop inside the kernel, the constraints generated by Eq. (5.3) would result in an unsafe (too low) upper bound for the number of times the software emulation is executed, because j is assumed to be executed at most once per iteration. In a nested loop, however, j can be executed multiple times per iteration, at maximum as often as the basic block its conditional branch for software emulation is contained in (see Fig. 5.6). For brevity, let $\mathrm{nf}(\mathrm{CI}_i, j)$ (nesting factor) be the statically known product of upper loop bounds for every level of loop nest to reach the basic

block which contains invocation j of CI_i from the kernel iteration top level and $\mathrm{nf}(\mathrm{CI}_i, j) = 1$ if it is not contained in a loop nest.
We obtain the following constraint:

$$x_{\mathrm{SW}_{i,j}} \leq \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}} \tag{5.6}$$

When including the analysis of conditional execution, we obtain the equality:

$$x_{\mathrm{SW}_{i,j}} = \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} \mathrm{inp}(i,j,k) \cdot u_k \cdot x_{\mathrm{init}} \tag{5.7}$$

Using Eq. (5.7), constraints for all invocations of CI_i are generated. After generating constraints for all invocations of $\mathrm{CI}_1, \ldots, \mathrm{CI}_n$, the global WCET bound analysis of a task including reconfigurable CIs can be performed for a single context (e.g., not considering cache effects).

Multiple Contexts

For extending the constraints of the previous section for multi-context timing analysis that enables treating the first iteration of a loop differently from the following iterations (e.g., for precise analysis of cache effects), we need to generate the constraint of Eq. (5.7) for every context the kernel can appear in. Let $t(x_{\mathrm{init}}) \subseteq \mathcal{T}$ be the subset of execution contexts x_{init} can appear in (see Section 5.3.1 for a definition of contexts). Again, the reconfiguration delay of a CI needs to be expressed as worst-case iterations of the kernel. However, in a multi-context analysis, a loop l can be in its first $F[l]$ and other $O[l]$ iterations. Therefore, the basic constraint in Eq. (5.3) becomes:

$$x_{\mathrm{SW}_{i,j}}^{\vartheta \circ F[l]} + x_{\mathrm{SW}_{i,j}}^{\vartheta \circ O[l]} \leq \sum_{k=1}^{i} u_k^{\vartheta} \cdot x_{\mathrm{init}}^{\vartheta} \quad \forall \vartheta \in t(x_{\mathrm{init}}) \tag{5.8}$$

Note that the number of iterations in which CI_i is unavailable u_i^{ϑ} is now also context-dependent as denoted by the superscript $\vartheta \in t(x_{\mathrm{init}})$, because iterations of the kernel now have different delays depending on the context the kernel is entered in. Therefore, we need to redefine u_i^{ϑ} in the following. The reconfiguration of CI_1 starts in the first iteration of the kernel, therefore in context $\vartheta \circ F[l]$. The worst-case delay for one iteration of the kernel is now context-dependent and especially dependent on whether the iteration is the first one of the other iterations of the kernel. We denote the first iteration, in which all CIs to be reconfigured are unavailable, as $\mathrm{WCET}_1^{\vartheta \circ F[l]}$. Following iterations in parallel to reconfiguration are all in context $\vartheta \circ O[l]$. $\mathrm{WCET}_1^{\vartheta \circ O[l]}$ denotes the worst-case time bound of an iteration in this context in parallel to configuring CI_1. $\mathrm{WCET}_i^{\vartheta \circ F[l]}$ and $\mathrm{WCET}_i^{\vartheta \circ O[l]}$, the WCET of one first or other kernel iteration with $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ available and $\mathrm{CI}_i, \ldots, \mathrm{CI}_n$ unavailable when the kernel is entered in context ϑ, can be determined analogously to the single context analysis explained in Section 5.4.2.
Now let us consider u_1^{ϑ}, the number of iterations in which CI_1 is unavailable. To account for the first iteration of the kernel, its delay is subtracted from the reconfiguration delay of CI_1 and u_1^{ϑ} is accordingly increased by 1. Together with the additional iteration to bound the timing anomaly discussed in Section 5.4.2, there are now at least 2 iterations with CI_1 unavailable. The formula directly resembles the single context definition in Eq. (5.2) when inserting the values for CI_1, u_1^{ϑ} becomes:

$$u_1^{\vartheta} := \begin{cases} \left\lceil \left(r_1 - \mathrm{WCET}_1^{\vartheta \circ F[l]} \right) \Big/ \mathrm{WCET}_1^{\vartheta \circ O[l]} \right\rceil + 2, \\ \quad \text{if } r_1 - \mathrm{WCET}_1^{\vartheta \circ F[l]} > 0 \\ 2, \text{ else} \end{cases} \tag{5.9}$$

As in the single context analysis, the remaining reconfiguration delay after the time spent in the first $\sum_{k=0}^{i-1} u_k^{\vartheta}$ iterations needs to be determined for determining u_i^{ϑ}. However, now a different context needs to be considered for the first iteration than for the others. Therefore, the context-sensitive extension of s_i is defined as follows:

$$s_i^{\vartheta} := \sum_{k=1}^{i} r_k - \left(\mathrm{WCET}_1^{\vartheta \circ F[l]} + (u_1^{\vartheta} - 1) \cdot \mathrm{WCET}_k^{\vartheta \circ O[l]} + \sum_{k=2}^{i-1} u_k^{\vartheta} \cdot \mathrm{WCET}_k^{\vartheta \circ O[l]} \right) \tag{5.10}$$

Finally, u_i^{ϑ} for $i > 1$ becomes:

$$u_i^{\vartheta} := \begin{cases} \left\lceil s_i^{\vartheta} \Big/ \mathrm{WCET}_i^{\vartheta \circ O[l]} \right\rceil + 1, \text{ if } s_i^{\vartheta} > 0 \\ 1, \text{ if } s_i^{\vartheta} + \mathrm{WCET}_{i-1}^{\vartheta \circ O[l]} > 0 \wedge s_i^{\vartheta} \leq 0 \\ 0, \text{ else} \end{cases} \tag{5.11}$$

As in the single-context case, u_i^{ϑ} for $i > 1$ can become 0 when CI_{i-1} and CI_i become available in the same iteration. For including the analysis of conditional CI execution and loop nests as explained in Section 5.4.2 and Section 5.4.2, $\mathrm{inp}(i,j,k)$ also needs to be extended for contexts as it is dependent on the WCET of one kernel iteration. $\mathrm{inp}(i,j,k)$ determines whether invocation j of CI_i is part of the worst-case path defining WCET_k. Thus, the context-aware $\mathrm{inp}^{\vartheta}(i,j,k)$ is defined as

$$\mathrm{inp}^{\vartheta}(i,j,k) := \begin{cases} 1, \text{ invocation } j \text{ of } \mathrm{CI}_i \text{ is part of the worst-case path of } \mathrm{WCET}_k^{\vartheta \circ F[l]} \text{ or } \mathrm{WCET}_k^{\vartheta \circ O[l]} \\ 0, \text{ else} \end{cases} \tag{5.12}$$

Including the analysis of conditional CI execution and loop nests analogously to the single-context constraint in Eq. (5.7), the final constraint for multi-context analysis is obtained:

$$x_{\mathrm{SW}_{i,j}}^{\vartheta \circ F[l]} + x_{\mathrm{SW}_{i,j}}^{\vartheta \circ O[l]} = \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} \mathrm{inp}^{\vartheta}(i,j,k) \cdot u_k^{\vartheta} \cdot x_{\mathrm{init}}^{\vartheta} \tag{5.13}$$

Using this equation, constraints for all invocations of $\mathrm{CI}_1, \ldots, \mathrm{CI}_n$ can be generated and global WCET bound analysis of a task can be performed including reconfigurable CIs with multiple contexts.

5.4.3 Stalling vs. Software Emulation

Modeling software emulation with parallel reconfiguration needs to make pessimistic assumptions to achieve a safe worst-case bound and an additional analysis needs to be performed to generate path analysis constraints. Therefore, it needs to be determined in which cases the approach is competitive or superior to stalling under timing guarantees. For clarity, we will only consider a single execution context in the following, but multiple contexts in the evaluation.

When stalling, execution halts for the duration of reconfiguring all CIs required in the upcoming kernel. Afterwards, every kernel iteration is executed in its worst-case execution bound with all CIs available, let us denote this worst-case bound as WCET_{n+1}. Therefore when stalling, the WCET for a kernel with an upper bound of I iterations (while neglecting the time taken to exit the kernel) is:

$$\sum_{k=1}^{n} r_k + I \cdot \mathrm{WCET}_{n+1} \tag{5.14}$$

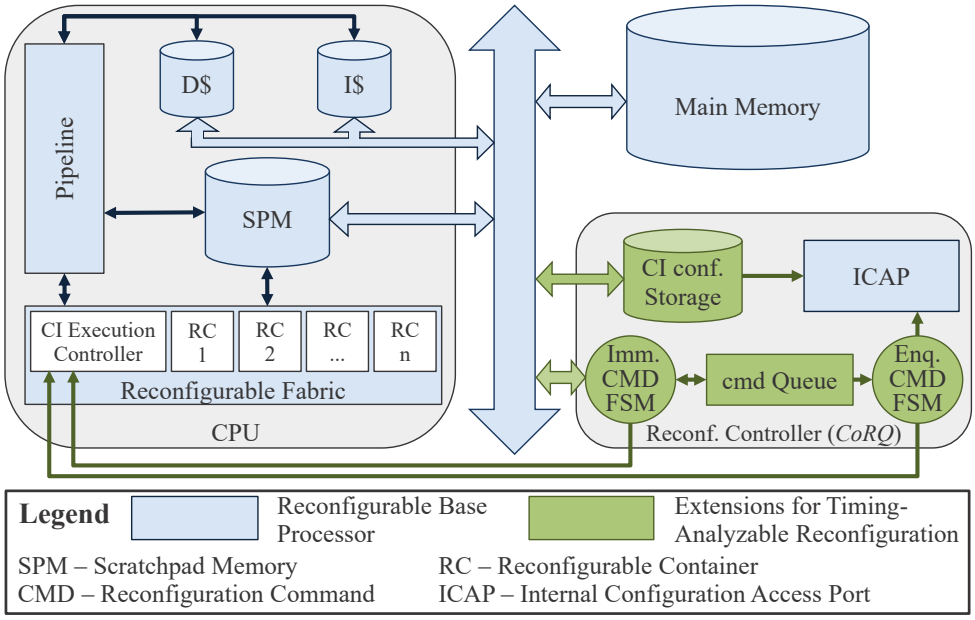


Figure 5.7: Overview of System on Chip used for Evaluation

From the considerations for generating path analysis constraints for the software emulation approach in Section 5.4.2, the WCET of a kernel can be bounded. For brevity, the basic constraints of Section 5.4.2 are discussed in the following. The upper bound of kernel iterations in which reconfiguration takes place is $\sum_{k=1}^{n} u_k$. Furthermore, during u_k iterations of the kernel each iteration has an upper execution time bound of WCET_k. The remaining $I - \sum_{k=1}^{n} u_k$ iterations have an upper bound of WCET_{n+1} each. Therefore, we obtain the following execution time bound for all kernel iterations:

$$\sum_{k=1}^{n} u_k \cdot \mathrm{WCET}_k + \left(I - \sum_{k=1}^{n} u_k \right) \cdot \mathrm{WCET}_{n+1} \tag{5.15}$$

For the software emulation approach resulting in a lower worst-case time bound, the inequality of (5.15) $<$ (5.14) needs to be satisfied. When simplifying this inequality, the following test is obtained:

$$\sum_{k=1}^{n} u_k \cdot (\mathrm{WCET}_k - \mathrm{WCET}_{n+1}) < \sum_{k=1}^{n} r_k \tag{5.16}$$

It can be noted that Inequality (5.16) is independent of the total iterations of the kernel, whether software emulation is beneficial over stalling can be decided by analyzing the first $\sum_{k=1}^{n} u_k$ iterations only. For a specific iteration included in u_k, $\mathrm{WCET}_k - \mathrm{WCET}_{n+1}$ denotes the additional time the software emulation takes because some CIs are unavailable. The software emulation approach results in a lower time bound for a kernel if and only if the total additional time remains lower than the total reconfiguration time. The practical implications of this analysis are investigated in the evaluation.

5.5 Runtime-Reconfigurable Processor Infrastructure for Timing Guarantees

For evaluating this work, the reconfigurable processor *i*-Core was employed, which was presented in Section 2.4. For the context of this chapter it is important to remember that a CI is executed by the CI Execution Controller in a protocol similar to other multi-cycle instructions like division and directly accesses register operands or the

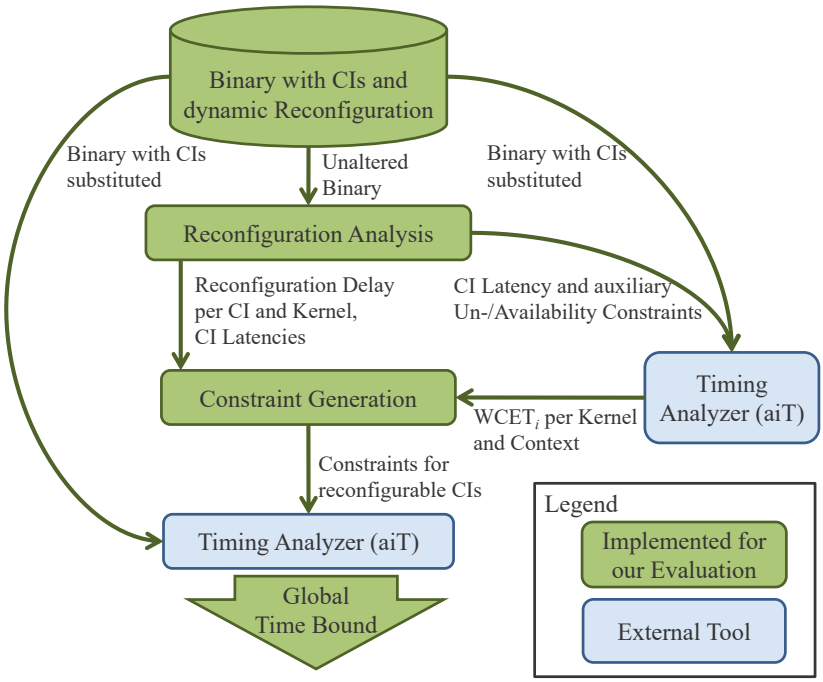


Figure 5.8: Evaluation toolflow

non-cacheable SPM only (it follows the generalized architecture shown in Fig. 5.7). This way, in microarchitectural analysis, a CI is just another multi-cycle instruction, which does not influence data cache analysis. The reconfigurable fabric is partitioned into *containers* of identical size [92]. A CI can be configured into any set of reconfigurable containers (possibly multiple) and potentially replaces a currently configured CI [11].
We further employ CoRQ (see Chapter 4) as the reconfiguration controller to perform timing analyzable reconfiguration using the integrated configuration access port (ICAP). CoRQ is accessible by the core pipeline as a memory-mapped on-chip bus device and processes reconfiguration commands that enable guaranteed reconfiguration delays (see Fig. 5.7, details in Chapter 4). For achieving predictability, we statically select which CIs to reconfigure at which program points and in what order. The resulting reconfiguration sequences are fed as commands to CoRQ by a sequence of stores to its address. CoRQ's internal bitstream memory is filled with all configurations required by the task over the bus at task load.

5.6 Experimental Evaluation

5.6.1 Implementation and Setup

The static timing analysis flow as described in Section 5.3 was split in several steps as depicted in Fig. 5.8.

(i) Reconfiguration analysis is performed on the compiled binary, giving absolute reconfiguration delays per CI.

(ii) AbsInt aiT [2] is used to determine $\text{WCET}_i^{\vartheta}$ (see Section 5.4.2) for all kernels. As aiT is closed-source software, we could not directly integrate support for CIs. Therefore, every CI opcode in the binary was substituted by an ADD opcode and a constraint in aiT's AIS2 Language to set the delay for the new ADD instruction to the delay of the specific CI (e.g., Fig. 5.9 (a)). aiT outputs an XML report, which we parsed to determine every u_i^{ϑ} for every kernel and generate the constraints described in Section 5.4.2 in AIS2 (e.g., Fig. 5.9 (b)).

(iii) Our generated constraints were used to calculate the global WCET bound using aiT.

```
instruction 0x40001238 additionally takes:  ((36*def("cISA_freq_mul"))-1) cycles;
```

(a) Example for Setting the CI Latency

```
flow sum:  point(0x40001244) == (2*67) point(0x400011a0);
```

(b) Example for a Generated Constraint for CI Availability

Figure 5.9: Generated Constraints in aiT's Format (AIS2)

Table 5.1: Kernels and Custom Instructions (CI) in the H.264 Encoder

`CI Name` and Short Description	Working Set	CLoC[4]
`MotionEstimation` Kernel		
`SATD`: Sum of Abs. Transf. Differences	16×16 px	123
`SAD`: Sum of Abs. Differences	16×16 px	24
`EncodeMacroBlock` Kernel		
`MC_Hz`: Motion Compens. Interpol. Horiz.	4 px	51
`IPred_HDC`: Intra Prediction Horiz.	16×16 px	35
`IPred_VDC`: Intra Prediction Vert.	16×16 px	19
`DCT`: Discrete Cosine Transf.	4×4 px	76
`HT2x2`: Hadamard Transform	2×2 px	12
`HT4x4`: Hadamard Transform	4×4 px	111
`LoopFilter` Kernel		
`LoopFilter`: In-Loop Deblock. Filter	4 px	82

The analysis is performed offline and runs on a workstation; it does not induce any runtime overheads. We evaluated our analysis with an H.264 encoder application, which uses 9 CIs covering the most compute-intensive kernels shown in Table 5.1. Every kernel configuration requires the whole CI containers. Therefore, before entering a kernel, reconfigurations for all containers are initiated to meet the kernel's CI requirements. It contains complex control flow with numerous decisions and nested loops. Most of the properties tested in the Mälardalen WCET Benchmarks[3] are covered, e.g., Discrete Cosine Transform is contained in both. For evaluating the overestimation of the static analysis, we executed the same binary obtained from BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1 in our SystemC-based cycle-accurate simulator which models the reconfigurable system shown in Fig. 5.7. Before performing the evaluation, we calibrated aiT and our simulator by harmonizing hardware parameters and verifying the results of test-cases, e.g., load-store sequences.

5.6.2 Results

In the following, the influences of stalling and software emulation on WCET bounds of a single kernel are analyzed. Afterwards, the overestimation and WCET reduction when using reconfigurable CIs on the whole H.264 encoder is analyzed.

Software Emulation vs. Stalling on a Single Kernel

For the analysis we use results obtained from performing timing analysis on a binary that executes the `LoopFilter` kernel of H.264 on 99 macroblocks (QCIF resolution). `LoopFilter` is the kernel of lowest complexity in our H.264 encoder, it contains a single CI and allows detailed analysis of worst-case CI availability. The guaranteed time bounds are compared to results obtained by executing the same binary in our simulator. Table 5.2 gives an overview of the parameters investigated. f_{fabric} stays constant at 100 MHz and we choose multiples of it for f_{CPU} which resemble realistic setups (rounded to the next power of two). E.g., the LEON3 processor which we

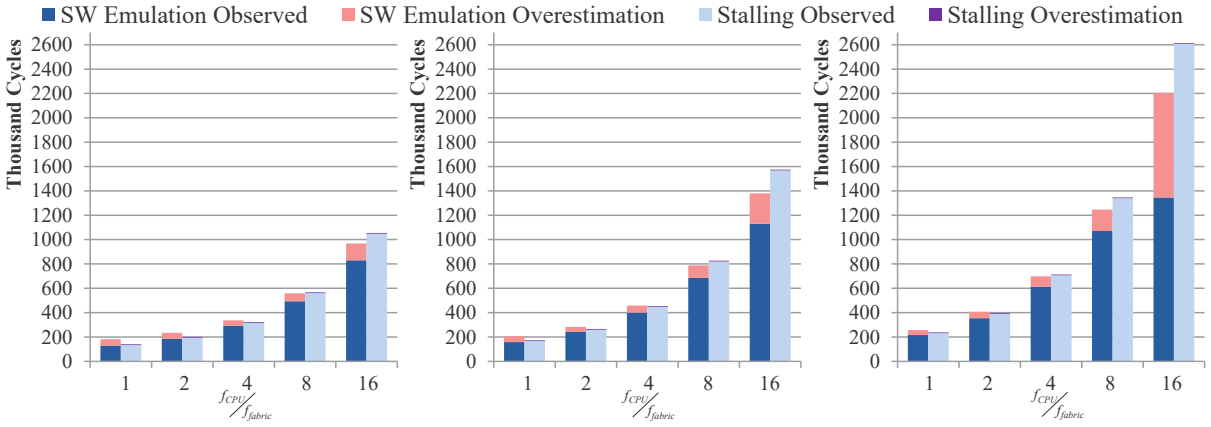
[3] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`
[4] C Lines of Code that are replaced by utilizing a hardware CI (without comments or whitespace)

Parameter [Unit]	Symbol	Values
CPU frequency [MHz]	f_{CPU}	100, 200, 400, 800, 1600
Fabric frequency [MHz]	f_{fabric}	100
Configuration Port Frequency [MHz]	f_{ICAP}	25, 50, 100

Table 5.2: Parameters investigated

$f_{\text{CPU}}/f_{\text{fabric}}$	**1**	**2**	**4**	**8**	**16**
u_1 at $f_{\text{ICAP}} = f_{\text{fabric}}$	3	4	6	11	21
u_1 at $f_{\text{ICAP}} = \frac{1}{2} \cdot f_{\text{fabric}}$	4	6	11	21	41
u_1 at $f_{\text{ICAP}} = \frac{1}{4} \cdot f_{\text{fabric}}$	6	11	21	41	81

Table 5.3: CI Unavailability (u_k) obtained during WCET bound estimation for `LoopFilter`



(a) $f_{\text{ICAP}} = f_{\text{fabric}}$ (b) $f_{\text{ICAP}} = \frac{1}{2} f_{\text{fabric}}$ (c) $f_{\text{ICAP}} = \frac{1}{4} f_{\text{fabric}}$

Figure 5.10: Observed Runtimes and Guaranteed WCET Bounds for `LoopFilter`

extended for reconfigurable CIs is advertised as running at 400 MHz when implemented as an ASIC, its successor the LEON4 is advertised running at 1500 MHz. The commercially available Xilinx Zynq-7000 SoC couples an ARM Cortex A9 at 866 MHz with a Xilinx 7-Series reconfigurable fabric.

All results are measured in cycles of the CPU pipeline, therefore they are determined by the relation of f_{CPU}, f_{fabric} and f_{ICAP}. Figure 5.10 (a) shows the results for $f_{\text{CPU}}/f_{\text{fabric}} \in \{2^0 = 1, \ldots, 2^4 = 16\}$ and $f_{\text{ICAP}} = f_{\text{fabric}} = 100$ MHz. This corresponds to running the Internal Configuration Access Port (ICAP) at its maximum frequency, and results in a reconfiguration bandwidth of 400 MB/s when configuring 32 bit of data every cycle. This reconfiguration bandwidth is possible when using a dedicated on-chip configuration storage (e.g., as available in CoRQ, see Chapter 4), we utilize the BRAM resources of Xilinx FPGAs in our prototype. When increasing the minimum evaluated CPU pipeline frequency of 100 MHz by a factor of c for a fixed ICAP frequency, the reconfiguration delay, measured in CPU cycles, increases by the factor c as well. Additionally, the runtime benefit of hardware CIs compared to software emulation decreases. According to our prediction in Section 5.4.3, software emulation results in a lower time bound than stalling for $f_{\text{CPU}}/f_{\text{fabric}} \in \{8, 16\}$, but not for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4\}$. The time bounds obtained for the kernel shown in Fig. 5.10 (a) reflect these predictions. Software emulation results in 30.28%, 16.39% and 4.48% higher time bounds than stalling for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4\}$, respectively. For $f_{\text{CPU}}/f_{\text{fabric}} \in \{8, 16\}$ software emulation results in 1.38% and 8.25% lower time bounds. When considering the observed worst-case runtime, however, software emulation always takes less time than stalling, i.e., 2.84%, 4.57%. 7.28%, 11.98% and 20.85% less for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4, 8, 16\}$, respectively. The reason for this discrepancy is that for analyzing software emulation for the worst-case time bound, pessimistic assumptions about CI availability need to be made as detailed in Section 5.4.2. In contrast, we know exactly at which point in a kernel a CI is available when stalling: directly from the beginning, after stalling for a statically known amount of cycles. Therefore, overestimation for software emulation ranges from maximal 40.17% with $f_{\text{CPU}} = f_{\text{fabric}}$ to 13.25% with $f_{\text{CPU}}/f_{\text{fabric}} = 8$, while the overestimation for stalling is never above 4.54%, again maximal with $f_{\text{CPU}} = f_{\text{fabric}}$.

Figure 5.10 (b) and Fig. 5.10 (c) show the results for $f_{\text{ICAP}} = 50\ \text{MHz} = \frac{1}{2} \cdot f_{\text{fabric}}$ and $f_{\text{ICAP}} = 25\ \text{MHz} = \frac{1}{4} \cdot f_{\text{fabric}}$, i.e., a reconfiguration bandwidth of 200 MB/s and 100 MB/s, respectively. This corresponds, e.g., to systems making use of cheaper but slower flash memory for the CI Storage instead of SRAM-based memory and therefore requiring a lower f_{ICAP}. The overall trend is that overestimation is lower with slower memories. In software emulation the pessimism of assuming an additional iteration of CI unavailability (see Section 5.4.2), has less influence on the guaranteed time bound as there are more iterations with the CI unavailable in total. At $f_{\text{CPU}}/f_{\text{fabric}} = 16$ and $f_{\text{ICAP}} = \frac{1}{4} \cdot f_{\text{fabric}}$ ($f_{\text{CPU}} = 64 \cdot f_{\text{ICAP}}$), overestimation rises again and reaches its overall maximum of 63.73%. As seen in Table 5.3, timing analysis guarantees that a maximum of 19 iterations (99 total iterations minus $81 = u_1$ plus 1, as discussed in Section 5.4.2) of the kernel need to be executed after reconfiguration delay at this point. In the observed runtime however, all iterations are finished in software during reconfiguration.
When stalling, overestimation also decreases slightly with slower memory as the reconfiguration delay takes a bigger share on the overall execution time and does not introduce overestimation. As more iterations in software emulation can be executed during reconfiguration and overestimation decreases, the resulting time bound is only 1.07% higher than stalling at $f_{\text{CPU}}/f_{\text{fabric}} = 4$, $f_{\text{ICAP}} = \frac{1}{2} \cdot f_{\text{fabric}}$ and already 2.02% lower at $f_{\text{ICAP}} = \frac{1}{4} \cdot f_{\text{fabric}}$. The observed execution time for software emulation is 10.33% and 13.11% lower than for stalling, respectively.
In sum, software emulation benefits from slow reconfiguration bandwidths or high CPU frequencies. In our results, to reduce the guaranteed time bound over stalling, the CPU needs to run at least at eight times the ICAP frequency due to pessimism. In the observed runtime, software emulation is always beneficial over stalling.

Overestimation

In this section, we analyze the influence of CIs on overestimation of WCET bounds for the H.264 encoder encoding 20 frames in QCIF resolution. Higher resolutions would increase the number of iterations per kernel and therefore reduce the relative effects of the reconfiguration delays on the total execution time. All results were taken with $f_{\text{ICAP}} = f_{\text{fabric}}$ and several values for $f_{\text{CPU}}/f_{\text{fabric}}$.
Figure 5.11 shows the percentage of overestimation for a general-purpose version of our H.264 encoder without any CIs (cISA execution only), and several alternatives of using reconfigurable CIs. As the runtime in the general-purpose case is unaffected by the fabric frequency, the amount of overestimation is constant at 38.04%. Introducing additional control flow by inserting the conditions for software emulation without actually configuring CIs –denoted as Software Emulation (always unavailable)– increases the overestimation slightly to 39.93%.
As mentioned in Section 5.6.2, the pessimism for bounding the timing anomaly when using software emulation is highest when reconfiguration of CIs takes only few iterations of a specific kernel. Overestimation reaches its maximum for software emulation when $f_{\text{CPU}} = f_{\text{fabric}}$ with 47.95% and its minimum at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ (maximum iterations during reconfiguration) with 11.89%. In sum, in our results the overestimation is less for software emulation than for the general purpose CPU when the pipeline frequency is twice the fabric frequency or higher. When using stalling, overestimation reaches its maximum of 17.97% when $f_{\text{CPU}} = f_{\text{fabric}}$ and its minimum of 6.66% when $f_{\text{CPU}}/f_{\text{fabric}} = 16$. It is generally lower than when using software emulation (see also Section 5.4.2) or cISA instructions only.
We can observe that increasing $f_{\text{CPU}}/f_{\text{fabric}}$ results in lower overestimation for both approaches for dealing with reconfiguration delay. This has two reasons:

(i) Increasing $f_{\text{CPU}}/f_{\text{fabric}}$ results in more kernel iterations which can be executed during the reconfiguration delay of a CI using software emulation. This means that the pessimism of assuming an additional iteration in software to bound the timing anomaly (see Section 5.4.2) has a lesser share on the total iterations and therefore a lesser effect on the timing bound.

(ii) Increasing $f_{\text{CPU}}/f_{\text{fabric}}$ also increases the share of execution time (measured in CPU cycles) spent on the fabric. The execution time on the fabric does not introduce overestimation and therefore the total overestimation decreases.

Software emulation is affected by (i) and (ii), while stalling is only affected by (ii). Therefore, increasing $f_{\text{CPU}}/f_{\text{fabric}}$ has a stronger effect on the overestimation of software emulation than of stalling.

Using the analysis of Section 5.4.3, we use a combination of software emulation and stalling to choose the more beneficial approach per kernel. As a result, we apply software emulation at $f_{\text{CPU}}/f_{\text{fabric}} = 8$ for two out of three kernels and stalling for the other one. $f_{\text{CPU}}/f_{\text{fabric}} = 16$ is equivalent to software emulation only. It turns out that while overestimation is higher than using stalling in these cases, the resulting time bound is lower. This is achieved by our models guaranteeing that the reconfiguration delay can be hidden effectively. All other cases are equivalent to stalling only.

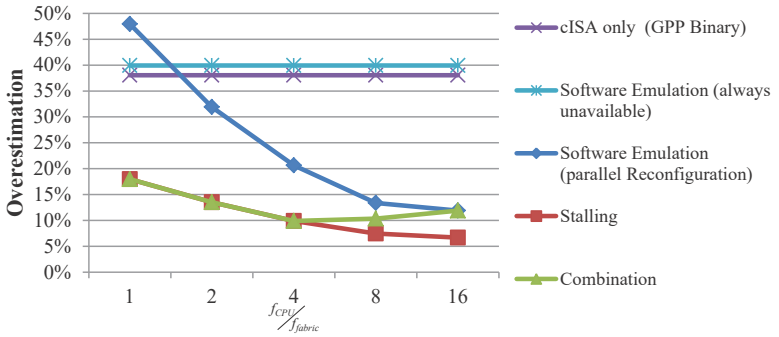


Figure 5.11: H.264 overall overestimation without CI Invocations (*cISA only*) and different alternatives of invoking CIs. *Software Emulation (always unavailable)* introduces CI Invocations, but never executes them in hardware. *Combination* chooses either Software Emulation or Stalling per kernel to optimize the timing bound (see Section 5.4.3).

Speedup

Figure 5.12 shows the speedup obtained by using runtime reconfiguration compared to execution on the cISA only. The left graph shows the speedup obtained in the guaranteed runtime, and the right graph shows the speedup of the observed runtime. As in the previous section, all results in this section are obtained with $f_{\text{ICAP}} = f_{\text{fabric}}$. The speedup in the guaranteed runtime is higher than in the observed runtime for stalling and the combination of stalling and software emulation from the previous section. The reason for this effect is that in addition to the actual speedup introduced by CIs, the overestimation is reduced as discussed in Section 5.6.2. Therefore, the speedup in the predicted runtime is on average 24.43% higher (minimum 17.01% at $f_{\text{CPU}}/f_{\text{fabric}} = 1$, maximum 29.42% at $f_{\text{CPU}}/f_{\text{fabric}} = 16$) than for the observed runtime when stalling. Software emulation results in a 11.5% higher speedup in the predicted runtime than in the observed runtime on average (minimum -6.70% at $f_{\text{CPU}}/f_{\text{fabric}} = 1$, maximum 23.37% at $f_{\text{CPU}}/f_{\text{fabric}} = 16$).

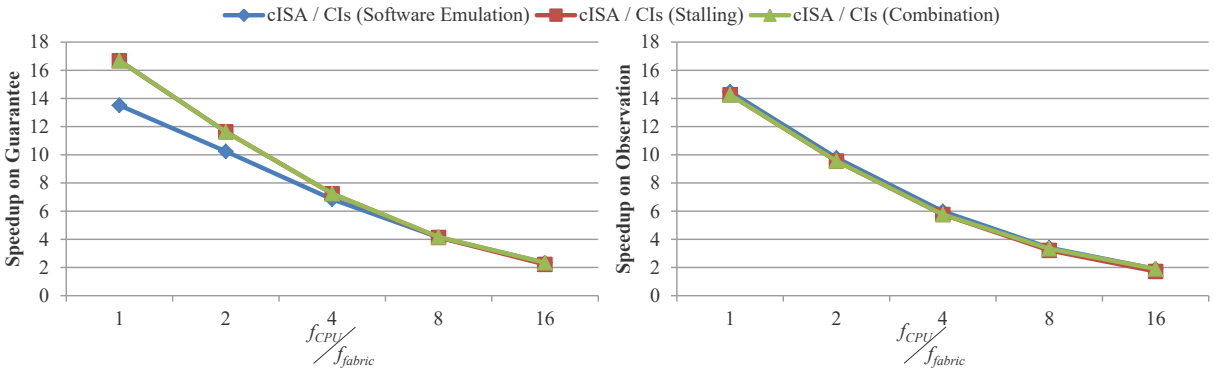


Figure 5.12: H.264 overall speedup on the guaranteed time bound (left) and the observed runtime (right)

Figure 5.13 takes stalling as a baseline and compares the guaranteed and observed results of software emulation and the combination of both approaches to it. Software emulation is always beneficial for the observed runtime with an average reduction of 4.8%, a minimum of 1.58% with $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and a maximum of 10.48% with $f_{\text{CPU}}/f_{\text{fabric}} = 16$. For the guaranteed WCET bound, however, software emulation is beneficial only when the CPU pipeline runs faster than the fabric at a factor of $f_{\text{CPU}}/f_{\text{fabric}} = 8$ or more, because of overestimation (see Section 5.6.2). Using software emulation for suitable kernels and stalling for others, the combination does not increase the runtime over stalling in any case. In cases where software emulation is beneficial for some kernels, the combination achieves the same or better guaranteed runtime reduction than using software emulation only.

5.7 Conclusion

This chapter presented a novel timing analysis approach for tasks on runtime-reconfigurable processors, it supports static analysis of runtime reconfiguration of multiple custom instruction (CIs) and multiple execution contexts (e.g., as used for precise worst-case analysis of cache effects). In the evaluation the precision of the analysis and the benefit of using CIs on WCET reduction as well as reduced overestimation was shown. We compared the effects on safe estimated WCET bounds of executing CI-equivalent software (software emulation) to halting execution (stalling) during the reconfiguration delay. In the observed worst case, software emulation was always beneficial over stalling. However, in the estimated time bound, software emulation was superior only when the CPU pipeline frequency was higher than the fabric frequency by a factor of eight or more as stalling can be analyzed more precisely. An analysis to choose either stalling or software emulation per kernel was introduced and evaluated to combine the benefits of both approaches. In sum, we have shown that runtime instruction set reconfiguration can be an enabling feature to provide timing-analyzable performance.

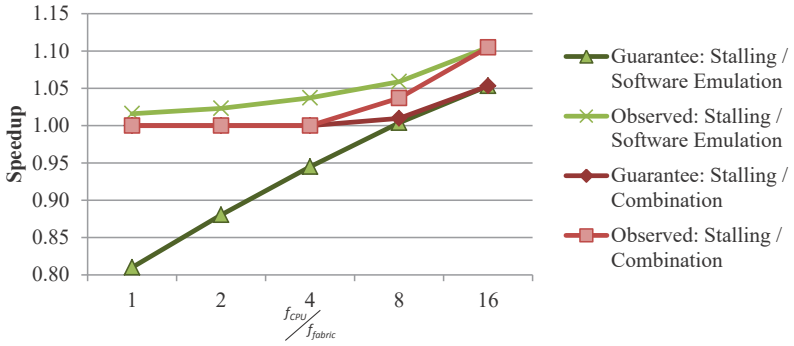


Figure 5.13: Speedup of Software Emulation and Combination over Stalling in H.264

In this chapter, the set of CIs to configure for each kernel was assumed given. It turns out, however, that in cases where there are more CIs to choose from than fit onto the reconfigurable area it is an NP-hard problem to choose the WCET-optimizing set of CIs. This problem is further complicated by the fact that computations (like the ones performed by CIs) can be implemented in hardware using different alternatives that choose a tradeoff between area requirements and resulting latency. The following chapter presents an optimal and a heuristic solution to selecting WCET-optimizing sets of CI implementations.

6 WCET Optimization using Reconfigurable Custom Instructions

The previous chapter has shown instruction set extensions by reconfigurable *custom instructions* (CIs) to be an effective means to achieve predictable performance. CIs were detailed in the context of *i*-Core in Section 2.4, their most-important properties for the context of this chapter are summarized as follows: CIs initiate execution of hardware accelerators configured on a reconfigurable fabric that is tightly coupled to a processor core (see [97] for an overview of reconfigurable architectures). An application binary in such an architecture provides directives to a *reconfiguration controller* (like CoRQ, see Chapter 4) to configure the CIs' accelerators onto the reconfigurable fabric. Reconfigurations are performed for the requirements of an upcoming *kernel* (also known as hot spot), i.e., a compute-intensive part of the application, e.g., a loop nest. In the previous chapter it was shown that –additional to a considerable speedup– the overestimation of a task's WCET can be reduced by moving calculations from software code to hardware CIs. CIs typically implement functionality that corresponds to several hundred instructions when executed on the CPU pipeline, possibly including conditional branches and other control flow. While analyzing instructions for worst-case latency may introduce pessimism due to, e.g., pipeline hazards or instruction cache misses, the latency of the hardware accelerators –executed on the reconfigurable fabric– is precisely known.
In this chapter[1] an approach of *selecting WCET-optimizing sets of CIs* for computational kernels that seamlessly integrates into state-of-the-art timing analysis is proposed. While this chapter does not target the reduction of overestimation of a task's WCET bound or resolving the problem of timing anomalies (like the one discovered in Section 5.4.2) in this work, an effective approach is presented to statically select sets of reconfigurable CIs to optimize a task's WCET bound and advance research on timing-analyzable high-performance architectures. *One main problem* in selecting WCET-optimizing CIs is *the instability of the worst-case path*, i.e., when reducing the

[1] The work presented in this chapter was originally published in [27]

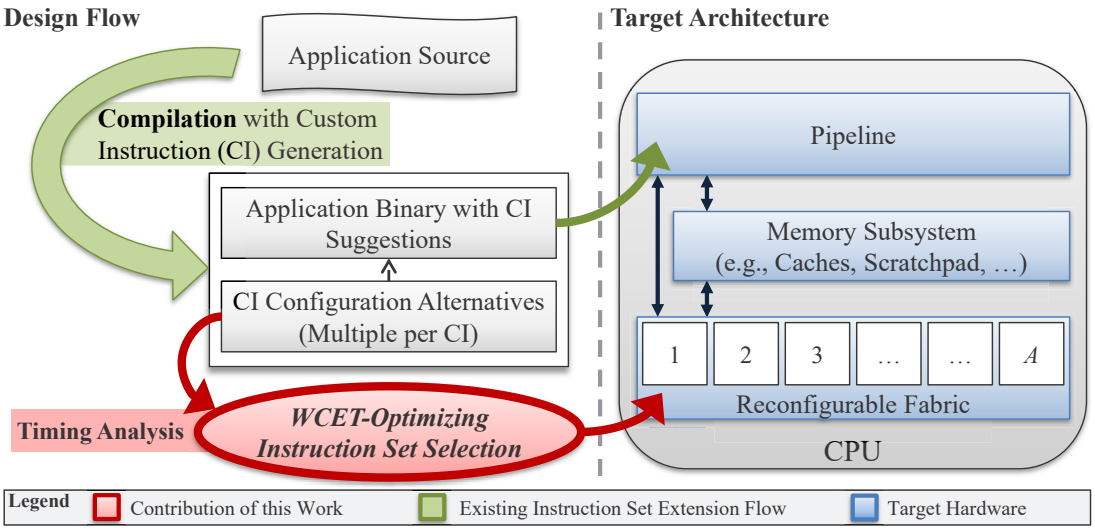


Figure 6.1: Toolflow performing *WCET-Optimizing Instruction Set Selection* integrated with timing analysis. As input to our approach we take application binary with suggestions where to place custom instructions as well as different implementation alternatives per custom instruction, differing in resource requirements and latency.

latency of the worst-case path by inserting a CI, a whole different path can become the new worst-case path. Therefore, WCET bound estimation is an integral part of WCET-optimizing CI selection. Figure 6.1 shows our envisioned toolflow. CI selection, also referred to as *instruction set selection*, is the second of the two main steps in the so-called *instruction set extension problem* [42]. The first step is the *CI generation* that is performed when compiling the application source code. In this step, kernels are identified in the application and partitioned into segments of code to execute in software and segments to execute in hardware. For the segments to execute in hardware, several alternatives that differ in resource demands as well as latencies are generated and then synthesized into configurations for the reconfigurable fabric. CIs provide an assembly-level interface to execute the hardware segments. Which CIs are implemented in hardware instead of the original software code and how much resources to allocate per CI is determined by the CI selection according to an optimization goal, e.g., average-case performance. Several approaches to CI generation exist that can provide CIs and implementation alternatives as an input to CI selection [42]. Different from existing CI selection approaches that target average-case performance, our novel WCET-optimizing selection requires the application binary, as it is the only way to be able to obtain precise WCET bound estimates (see Section 2.2 and [106]). To obtain a finished binary with generated CIs while keeping the flexibility to execute the original software, we introduce *CI super blocks* which will be detailed in Section 6.2. Effectively, the selection step of the instruction set extension problem is moved from the compiler to the timing analyzer in this work, i.e., *post-compilation*. This is achieved by extending the analysis of the conditional jump that either jumps to the hardware CI, if configured, or the original software code, otherwise, which was introduced in the previous chapter. The result is an effective technique that considerably reduces the guaranteed WCET bound compared to the original task that does not use CIs.
The novel contributions of this chapter are:

- Modeling the WCET-optimizing instruction set selection problem with support for global program flow information and reconfiguration delay by extending state-of-the-art models used in timing analyzers like AbsInt aiT [2] or OTAWA [7].
- An optimal solution that effectively reduces the search space by mapping selection candidates to *weak compositions of an integer*, i.e., the algorithm recursively generates all distributions of reconfigurable fabric area to CIs while adhering to area constraints. Recursion subtrees corresponding to distributions of area that cannot be utilized in CI implementations are pruned early. In our evaluation we show that less than 1% of all possible 570,240 selections need to be evaluated when optimizing the `EncodeMacroBlock` kernel as part of the H.264 encoder with our optimal search algorithm.
- A heuristic solution that performs a maximum number of WCET estimates linear in the partitions of area available for configuring CIs on the reconfigurable fabric. It reduces the runtime of optimization down to 11.18% of the optimal search algorithm in the before-mentioned `EncodeMacroBlock` kernel, the most-complex kernel evaluated. Its results produced maximum 2.52% lower speedups on the WCET than optimal in our evaluations.

We show that previous work targeting optimization of the worst-case path, e.g., instruction cache locking or scratchpad memory allocation of program code, share similarities with the WCET-optimizing instruction set selection problem, but cannot be adapted to obtain optimal solutions. For introducing runtime instruction set reconfiguration as an enabling feature to provide timing-analyzable performance, novel models and solutions are required.

6.1 Related Work and Motivation

WCET-optimizing instruction set selection bears resemblance to other static optimizations targeting the worst-case path like instruction cache locking or scratchpad memory allocation of program code. In this section, the

differences of these problems to WCET-optimizing instruction set selection are pointed out. Additionally, state-of-the-art solutions to instruction set selection specifically are discussed and their shortcomings explained.

Caches are used to effectively reduce the average memory access latency of a CPU. It is very difficult to predict whether a memory access can be served by the cache (cache hit) or needs to be served by main memory (cache miss). WCET analysis always needs to consider a cache miss when it cannot guarantee a cache hit. This typically leads to overestimation of the WCET bound. *Cache locking* is a software-controlled mechanism to load code segments into the cache and prevent them from being evicted. Several works utilize instruction cache locking to reduce overestimation resulting from cache analysis and thus lowering the WCET bound [40, 66, 78]. Similarly, the instruction cache can be replaced by allocating program code directly to predictable scratchpad memory [39]. Even though these techniques are complementary to instruction set selection, the question arises whether the same algorithms can be applied. Similar to instruction set selection, the instruction cache locking and program code allocation problem entail WCET estimation to determine the worst-case path and using this information to select code segments that can be most profitably sped up for lowering the WCET bound. However, both need to choose between two alternatives for a code segment only: utilizing the fast memory (i.e., locking it in the cache or allocating it in scratchpad memory) or main memory. Instruction set selection has several alternatives to choose from: the original software or different CI implementations for the same functionality with different degrees of parallelism and therefore different delays as well as resource requirements. Even with extensions for evaluating multiple alternatives to choose from (e.g., the different CI implementations), existing algorithms for cache locking would remain unsuitable for our problem. In [40] and [66] the problem is modeled similarly using *Execution Flow Graphs* and *Execution Flow Trees*, respectively. However, the execution flow is modeled on the level of function calls. As this work targets kernels, the aim of this chapter is to model the function-internal control flow.

In [78] as well as [39] function-internal control flow is modeled similarly to the instruction set selection presented in [112], which in turn is an ILP formulation of a WCET estimation technique called *timing schema* [76]. Timing schema is a tree-based WCET estimation technique (see [38] for an overview of estimation techniques). In current timing analyzers, it was succeeded by the more powerful Implicit Path Enumeration Technique (IPET) [64], which was introduced in Section 2.2.1. Timing schema is still commonly used in state-of-the-art WCET optimization approaches however, because it is computationally cheap and it enables WCET optimization to be modeled as a single ILP (as opposed to the combinatorial problem that we present in Section 6.3). In timing schema, the estimation is calculated by building a representation which generally corresponds to the abstract syntax tree of the program and traversing it bottom-up by simple recursive rules. Infeasible path information cannot efficiently be applied, because the recursive rules are local to program statements [38]. This can lead to imprecise WCET estimates as shown in the simple example in Fig. 6.2: the rules are unable to capture the global information that the `true` case of the `if` statement can appear maximum 5 times in the worst-case path. In this example, timing schema produces an estimate based on a program path that executes the `true` case 100 times and therefore this case seems to be the most profitable candidate to be optimized. However, this path never appears in an actual execution of the program. State-of-the-art timing analyzers can correctly determine that the `false` case dominates the WCET in the example in Fig. 6.2 using value analysis and generating constraints for IPET. Therefore, when utilizing a computationally cheap, but imprecise, WCET estimation technique like timing schema during WCET optimization, the allocated resources may not even be utilized in the final WCET bound that is obtained using a timing analyzer. Additionally, state-of-the-art timing analyzers support powerful annotation languages to provide global path information [59] (the impact on WCET optimization is evaluated in Section 6.6.3). Thus, we propose to extend state-of-the-art timing analysis using IPET to support WCET optimization, as opposed to treating WCET optimization and timing analysis as two separate processes.

In [112] WCET-optimizing instruction set selection for *instruction set extensible processors* is performed. These processors contain custom functional units that can be configured to implement frequently used instruction patterns for speedups by exploiting instruction level parallelism and operator chaining [111]. According to the processor

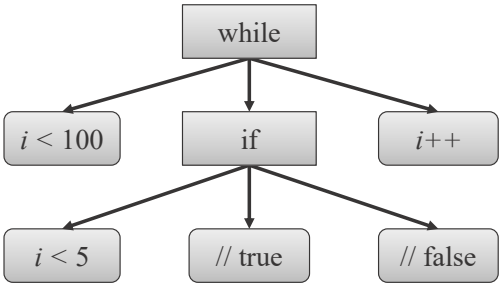
Timing Schema Rules (excerpt):

- T(`if` (Exp) T `else` F) = T(Exp) + max (T(T), T(F))
- T(`while` (Exp) Body = T(Exp) + $n \cdot$ (T(Exp) + T(Body))

Program Code:

```
int i = 0;
while (i < 100) {
  if (i < 5)
    ..; // t_T = 8
  else
    ..; // t_F = 4
  i++; }
```

Syntax Tree:



Bottom-up Calculation (with T(Exp) = 1)**:**

(i) t_{if} = T(`if` $(i < 5)$ T `else` F) = T$(i < 5)$ + max $(t_{\text{T}}, t_{\text{F}}) = 1 + \max(8,4) = 9$
$\Rightarrow$ `true` case explicitly determined as worst-case path.

(ii) T(`while` $(i < 100)$ Body) = T$(i < 100) + 100 \cdot (\text{T}(i < 100) + t_{\text{if}} + \text{T}(i{+}{+}))$
$= 1 + 100 \cdot (1 + 9 + 1) = \underline{1101}$

$\Rightarrow$ Decision: optimize `true` case, e.g., using cache locking or a custom instruction.

Actual WCET:
T$(i < 100) + 101 \cdot \text{T}(i < 5) + 5 \cdot t_{\text{T}} + 95 \cdot t_{\text{F}} + 100 = \underline{622}$
$\Rightarrow$ In contrast to the result obtained by Timing Schema, the `false` case dominates the WCET. Optimizations relying on timing schema would therefore allocate resources on the wrong path.

Figure 6.2: Simple example that shows how WCET optimization approaches that rely on Timing Schema perform suboptimal decisions

model used in that work, the presented heuristic assumes a uniform cost per selected pattern (i.e., occupation of one custom functional unit). The WCET-optimizing instruction set is selected per task, i.e., during task execution the instruction set is fixed. Therefore, the cost of configuring a selected pattern is not taken into account in their approach. In this chapter, dynamic reconfiguration of custom instructions with varying area demands is targeted (1 up to A units of the reconfigurable fabric area). For evaluating the profit of an instruction on reducing the WCET estimate, its required area demands as well as its reconfiguration delay need to be factored in. The impact of reconfiguration delay on WCET optimization is evaluated in Section 6.6.2.
In summary, state-of-the-art WCET optimization approaches model program flow at the level of function calls, rely on the imprecise timing schema, do not consider reconfiguration delay while evaluating the profits of potential decisions or support binary decisions only (either optimize a certain path or not). In the following, all of these shortcomings are resolved.

6.2 System Model

Similar to the timing analysis presented in the previous chapter, the optimization presented in this chapter is applied to the reconstructed control-flow graph (CFG) of an application in binary form, as it is the only way to obtain safe and precise WCET estimates [106]. To enable the WCET-optimizing selection of CIs, additional compile-time information is required: potential CIs and their possible configurations to choose from (see [42] for an overview). The *granularity* of a CI, i.e., the amount of software it replaces, depends on the specific target architecture. In our evaluation, CIs replace 12 to 123 lines of C code (see Section 6.6.1). For configuring the CIs

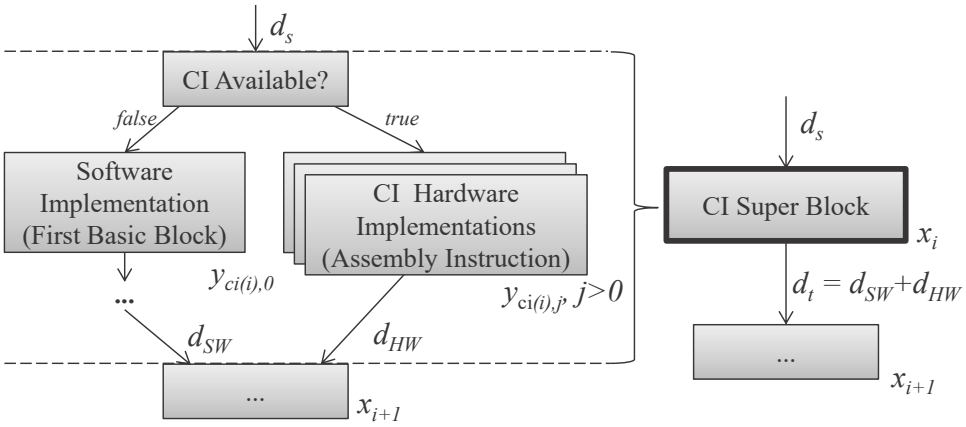


Figure 6.3: CI super block as part of a CFG

in hardware, we assume reconfigurable fabric area to be allocatable in up to A discrete units. This corresponds to the common area model of dividing the fabric area into A equally-sized *partitions*[2] like in the 1D or 2D partitioned area models in [93]. As in the previous chapter, reconfigurations are requested before beginning execution of a kernel to configure CIs that speed up the kernel's computations, as is shown in Fig. 6.4 (a). Let $\mathcal{CI}$ be the set of all CIs. We assume a specific configuration j of a CI $k \in \mathcal{CI}$ in hardware to have a constant delay $t_{k,j}$ (cycles spent in the pipeline's execution stage), to require area on the reconfigurable fabric $a_{k,j} \in [1,A]$ and to take a constant reconfiguration delay $r_{k,j}$ for configuring it on the fabric. For a constant reconfiguration delay, a constant bandwidth for transferring configuration data to the reconfigurable fabric's configuration memory needs to be guaranteed, e.g., by employing CoRQ (see Chapter 4 for details). Stalling the CPU during reconfiguration is assumed in this chapter for WCET optimization (see Fig. 4.1). Note that the resulting CI selection can directly be used in a system that employs software emulation and parallel reconfiguration at runtime after a WCET bound is obtained using the timing analysis approach of Chapter 5.

Additional to hardware configurations, a CI can be implemented using its original software code $j = 0$. The software implementation does not have a constant delay $t_{k,0}$, because it is subject to, e.g., cache and pipeline analysis in the specific context that it is executed in. It does not require fabric area nor reconfiguration delay (i.e., $a_{k,0} = r_{k,0} = 0$). For providing the flexibility to execute the original software for generated CIs, we introduce *CI super blocks* (which are a timing analysis construct that base on the conditional branch used in Chapter 5). As shown in Fig. 6.3, CI super blocks begin with a conditional branch before every CI (the actual instruction in the binary), which jumps to the functionally equivalent software code when the CI is not implemented in hardware. If a configuration for the CI is available on the reconfigurable fabric, the CI is executed instead of jumping to the software. The CI super block ends by joining paths of hardware CI and software. Multiple CI super blocks in the binary can execute the same CI k. Let $\mathcal{B}$ be the set of all blocks, i.e., basic blocks (not contained in super blocks) as well as super blocks. The function $\mathrm{ci}(i)$ determines which CI k is executed by a super block $i \in \mathcal{B}$, i.e.:

$$\mathrm{ci}\colon \mathcal{B} \to \mathcal{CI} \cup \{0\}, i \mapsto k, \text{ with } \mathrm{ci}(i) = 0 \notin \mathcal{CI} \text{ if } i \text{ is a basic block (not a super block)} \tag{6.1}$$

The context-dependent delay for executing implementation j of CI super block i is denoted as $e_{i,j}$ for hardware as well as software implementations. While CI execution on the reconfigurable fabric itself is context independent ($t_{\mathrm{ci}(i),j}$ is constant, for $j > 0$), invoking the CI from the CPU pipeline can add additional cycles, e.g., because of pipeline hazards or instruction fetch miss of the CI. Therefore, $e_{i,j} \geq t_{\mathrm{ci}(i),j}$ for $j > 0$. Consider the example of

[2] A partition directly maps to a reconfigurable container on the evaluation platform *i*-Core (see Section 2.4). The more general term 'partition' is employed throughout this chapter for consistency with commonly-used area models like presented in [93].

Fig. 6.4 (a), it provides input to the WCET-optimizing instruction set selection. In this example, two CIs were generated, one with $m_1 = 2$ and the other with $m_2 = 3$ different hardware implementations. From *microarchitectural analysis* (see Section 2.2), the worst-case bound per block which considers, e.g., cache, pipeline or branch prediction effects is obtained (see Fig. 6.4 (b)). This way, $e_{4,0}$ and $e_{6,0}$ can be unequal, even when they execute the same CI ($\text{ci}(\{4,6\}) = 1$) in the same implementation ($j = 0$). $e_{i,j}$ is the main parameter that is used to calculate WCET estimates based on a specific selection of implementations in Section 6.3. Equation (6.1) is used to concisely formulate Eqs. (6.4) to (6.8) that our WCET estimation is based on. Effectively, a CFG is obtained that is parameterized by a CI selection using CI super blocks. In the following the WCET bound estimation technique IPET (introduced in Section 2.2.1) is extended to the problem formulation of this chapter for evaluating and directing the WCET optimization.

6.3 Problem Formulation

In order to obtain precise WCET estimates that utilize global program flow information during instruction set selection, the system model of Section 6.2 and global bound calculation using IPET (see Section 2.2.1) are integrated in the following. Selecting an instruction set to optimize the WCET bound essentially means that the WCET is minimized over all possible selections, i.e., the aim is to *minimize* the *maximum* execution time. In the following, the ILP-formulation of IPET is extended for capturing the implementation alternatives of a CI $k \in \mathcal{CI}$. To this end, new variables $y_{k,j} \in \{0,1\}$ are introduced for every implementation j with $y_{k,j} = 1$ if CI k is implemented using alternative j and $y_{k,j} = 0$ otherwise. E.g., $y_{k,0} = 1$ would mean that CI_k is not implemented in hardware but utilizes its original software instead (see Section 6.2 and Fig. 6.3). The following constraint is introduced to ensure that exactly one implementation is chosen –potentially in software ($j = 0$) or hardware ($j > 0$)– with m_k being the number of hardware configurations of CI k:

$$\sum_{j=0}^{m_k} y_{k,j} = 1 \quad \forall k \in \mathcal{CI} \tag{6.2}$$

To only allow solutions that fit onto the reconfigurable fabric, the following area constraint is introduced:

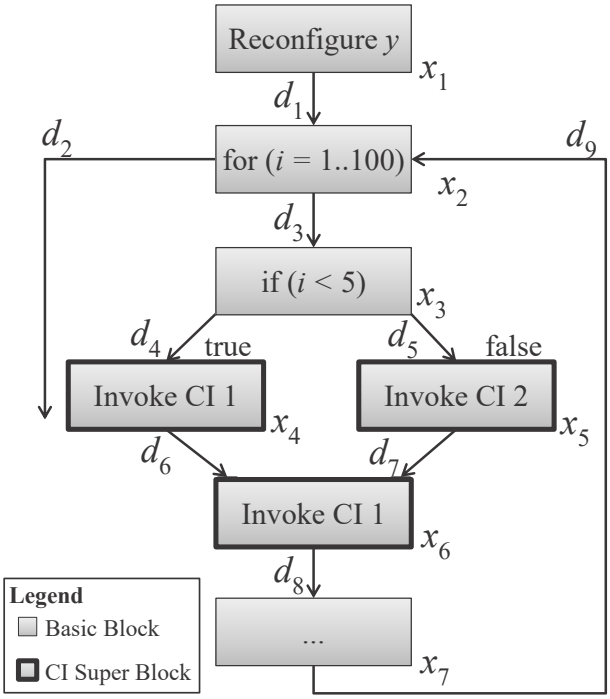
$$\sum_{k=1}^{|\mathcal{CI}|} \sum_{j=0}^{m_k} a_{k,j} y_{k,j} \leq A \in \mathbb{N}_0 \tag{6.3}$$

I.e., the sum of area on the reconfigurable fabric $a_{k,j}$ required to implement all CIs k using the selected implementation j (for which $y_{k,j} = 1$) needs to be lower than or equal to the total fabric area A. Any $y \in \{0,1\}^{|\mathcal{CI}| \times M}$, with $M = \max_{k \in \mathcal{CI}} m_k + 1$, satisfying Eq. (6.2) and Eq. (6.3) is a feasible instruction set selection. As shown in Fig. 6.4 (c), the obtained constraints are used to extend constraints generated by IPET.

The objective function for optimizing the WCET in the presence of CI super blocks is developed as follows. The system model introduced in Section 6.2 enables us to capture every implementation alternative as a single super block in the CFG (see Fig. 6.3). The total cycle contribution of CI k's super block i to the WCET bound is given as:

$$\sum_{j=0}^{m_k} e_{i,j} y_{k,j} x_i \tag{6.4}$$

(a) Input to WCET-Optimizing Timing Analysis:



Reconfigurable area:
$A = 5$
Generated CIs:
$\mathcal{CJ} = \{1,2\}, |\mathcal{CJ}| = 2$
Hardware implementations per CI:
$m_1 = 2, m_2 = 3$
Area demands ($a_{k,0}$: software):
$a_1 = (0,3,3), a_2 = (0,2,4,5)$
Reconfiguration delays:
$r_1 = (0,10,10), r_2 = (0,7,12,16)$
CI latencies on reconfigurable fabric (undefined for software: $t_{k,0} = \perp$):
$t_1 = (\perp,10,12), t_2 = (\perp,15,11,9)$

(b) Obtained from Microarchitectural Analysis:

Worst-case basic block delays:
$c_1, c_2, c_3, c_7 \in \mathbb{N}$

Invoked CIs:
$\text{ci}(\{1,2,3\}) = 0, \text{ci}(\{4,6\}) = 1, \text{ci}(\{5\}) = 2$

Worst-case CI Super Block delays (in order of invoked CI):
$e_4 = (50,10,12), e_6 = (48,10,12), e_5 = (60,18,14,12)$
($e_{k,j} \geq t_{k,j}$ for $j > 0$, because execution history-dependent, see Section 6.2)

(c) Generated Constraints:

Program Structure (by IPET):

$$1 = x_1 = d_1 \text{ (\textit{kernel entry constraint})}$$
$$x_2 = d_1 + d_9 = d_2 + d_3$$
$$x_3 = d_3 = d_4 + d_5$$
$$x_4 = d_4 = d_6$$
$$x_5 = d_5 = d_7$$
$$x_6 = d_6 + d_7 = d_8$$
$$x_7 = d_8 = d_9$$

Global Information:

$$x_3 \leq 100 \cdot d_1 \text{ (\textit{upper loop bound})}$$
$$x_4 \leq 5 \cdot d_1 \text{ (\textit{true case max. 5 times})}$$

CIs and Reconfigurable Fabric:

$$\sum_{j=0}^{2} y_{1,j} = 1, \sum_{j=0}^{3} y_{2,j} = 1$$
(exactly one configuration per CI)

$$\sum_{k=1}^{2} \sum_{j=0}^{m_k} a_{k,j} y_{k,j} \leq 5 \text{ \textit{(area constraint)}}$$

(d) Generated Combinatorial Objective Function:

$$\min_{y \in \{0,1\}^{2\times 4}} \left(\max_{x \in \mathbb{N}_0^6} \left(c_1 x_1 + c_2 x_2 + c_3 x_3 + c_7 x_7 + \sum_{j=0}^{2} e_{4,j} y_{1,j} x_4 + \sum_{j=0}^{3} e_{5,j} y_{2,j} x_5 + \sum_{j=0}^{2} e_{6,j} y_{1,j} x_6 \right) + \sum_{j=0}^{2} y_{1,j} r_{1,j} + \sum_{j=0}^{3} y_{2,j} r_{2,j} \right)$$

Figure 6.4: Simple example of how an instance of the problem formulated in Sections 6.2 and 6.3 is generated

E.g., when choosing the software implementation, the cycle contribution becomes $e_{i,0}x_i$, which directly resembles the contribution of a basic block in IPET's objective function ($\max \sum_{i=1}^{N} c_i x_i$, see Section 2.2.1). The WCET for a given selection y without accounting for reconfiguration delay can be determined as:

$$\mathrm{WCET}'(y) := \max_{x \in \mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i) \notin \mathcal{CI}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i) \in \mathcal{CI}}}^{|\mathcal{B}|} \sum_{j=0}^{m_{\mathrm{ci}(i)}} e_{i,j} y_{\mathrm{ci}(i),j} x_i \right) \tag{6.5}$$

Additionally, the reconfiguration delay induced by a selection y needs to be accounted for. Neglecting it could result in suboptimal selections in which the time spent configuring the selected CIs outweighs the time saved by performing hardware-accelerated calculations (more details in Section 6.6.2). Every CI super block utilized in a kernel is configured exactly once before entering the kernel (with zero reconfiguration delay for software implementation). Therefore, the WCET including reconfiguration delay is obtained as follows:

$$\mathrm{WCET}_r(y) := \mathrm{WCET}'(y) + \sum_{k=1}^{|\mathcal{CI}|} \sum_{j=0}^{m_k} y_{k,j} r_{k,j} \tag{6.6}$$

For every selection y, an ILP instance that determines the WCET of the kernel when reconfiguring y is obtained. E.g., when selecting the software implementation for every CI, the following objective function is obtained, which again resembles an objective function of an IPET problem instance without any CIs:

$$\max_{x \in \mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i) \notin \mathcal{CI}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i) \in \mathcal{CI}}}^{|\mathcal{B}|} e_{i,0} x_i \right) \tag{6.7}$$

Note that for every choice of y, only $\mathrm{WCET}_r(y)$ changes while **the constraints remain static once they were generated**.

Putting it all together, the WCET-optimizing instruction set selection problem becomes a combinatorial problem with the following objective function:

$$\min_{y \in \{0,1\}^{|\mathcal{CI}| \times M}} \left(\max_{x \in \mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i) \notin \mathcal{CI}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i) \in \mathcal{CI}}}^{|\mathcal{B}|} \sum_{j=0}^{m_{\mathrm{ci}(i)}} e_{i,j} y_{\mathrm{ci}(i),j} x_i \right) + \sum_{k=1}^{|\mathcal{CI}|} \sum_{j=0}^{m_k} y_{k,j} r_{k,j} \right) \tag{6.8}$$

The objective function for our example in Fig. 6.4 is shown in Fig. 6.4 (d). As there are finite choices for $y \in \{0,1\}^{|\mathcal{CI}| \times M}$ ($|\mathcal{CI}|$ and M are finite), Eq. (6.8) could be transformed into a single ILP by resolving the $\min_y(\dots)$ of Eq. (6.5) into one constraint per choice of y. However, this would result in up to $2^{|\mathcal{CI}| \cdot M}$ constraints of high complexity, which becomes practically infeasible even for small values. Also note that the ILPs only need to be evaluated per kernel and not for the whole application. Therefore, the ILPs are considerably less complex (fewer variables and constraints) than the ILP for determining the WCET of the whole application. In the following section we will show how the search space can be pruned and feasible y are generated efficiently.

6.4 Optimal Solution

In theory, up to $2^{|\mathcal{CI}| \cdot M}$ possible selections y need to be evaluated. In practice, however, the search space is considerably smaller for the following reasons:

- The number of possible hardware configurations m_k per CI k varies a lot, e.g., in our evaluation we had a minimum of 1 to a maximum of $78 = M$ implementations for CIs (including software implementation) within

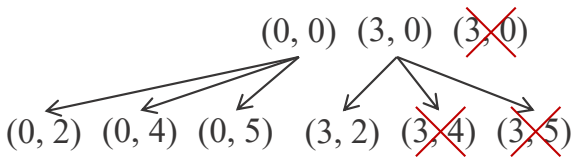
one kernel (more details Section 6.6). From these $\sum_{k=1}^{|\mathcal{CI}|}(m_k+1) \leq |\mathcal{CI}| \cdot M$ different CI implementations in total, again in practice only a small subset is relevant. For the CI with 78 different implementations, many implementations had different degrees of parallelism and latencies, but required the same amount of area and reconfiguration delay when synthesized to the reconfigurable fabric. When considering only the minimum-latency implementation per required fabric area, our algorithm was able to prune the number of implementations to 10 relevant ones. Therefore, in practice the relevant number of implementations per CI k is much smaller than m_k+1.

- Additionally, the possible selections can be pruned considerably when applying the area constraint early (see Eq. (6.3)). Let us consider the inner sum of Eq. (6.3), it models the allocation of area per CI for a specific selection as a tuple $a = (a_1, a_2, \ldots, a_{|\mathcal{CI}|})$. To prune the search space, we will find the number of unique tuples fulfilling Eq. (6.3) in the following. Having a total area of A on the reconfigurable fabric means the number of selections utilizing the *whole* fabric is equal to the number of possibilities to distribute the area to CIs such that $\sum_{k=1}^{|\mathcal{CI}|} a_k = A$ (allowing $a_k = 0$ for the software implementation). I.e., the number of selections utilizing the whole fabric is the number of so-called *weak compositions* of the integer A into exactly $|\mathcal{CI}|$ parts, which is $\binom{A+|\mathcal{CI}|-1}{|\mathcal{CI}|-1}$ [51]. The number of all unique tuples fulfilling Eq. (6.3) (which additionally allows less than A area to be distributed, i.e., $\sum_{k=1}^{|\mathcal{CI}|} a_k \leq A$), is exactly $\sum_{s=0}^{A}\binom{s+|\mathcal{CI}|-1}{|\mathcal{CI}|-1} < 2^{A+|\mathcal{CI}|}$. Effectively, a maximum of $2^{A+|\mathcal{CI}|}$ ILPs need to be solved to find the WCET-optimal selection.

Algorithm 1 Recursive Search for Optimal Selection

```
1:  y^best ← (0,...,0), WCET^best ← ∞                ▷ For brevity, global variables to obtain result
2:  function OPTSEARCH(A, k, y)                      ▷ Remaining area A, CI k, (partial) selection y
3:      if k < |CI| + 1 then                         ▷ Function was called to select CI k
4:          for a_k ← 0, A do                        ▷ For all possible possible values of a_k
5:              y'_k ← GETMINLATENCYIMPL(k, a_k)
6:                                                   ▷ Minimum latency implementation for CI k allocating exactly a_k area
7:              if y'_k ≠ 0 then                     ▷ Implementation allocating exactly a_k area exists
8:                  y' ← (y_1,...,y_{k-1}, y'_k, 0,...,0)^T   ▷ Add found implementation to current selection
9:                  OPTSEARCH(A − a_k, k + 1, y')    ▷ Branch to another recursion subtree
10:             end if
11:         end for
12:     else                                         ▷ Implementations for all CIs selected, evaluate resulting selection y
13:         if WCET_r(y) < WCET^best then            ▷ Calculate WCET bound for y, see Eq. (6.6)
14:             y^best ← y, WCET^best ← WCET_r(y)    ▷ Save so far best evaluated selection
15:         end if
16:     end if
17: end function
```

Combining both observations leads to an additional opportunity for pruning, which our optimal search algorithm shown in Algorithm 1 exploits. The algorithms recursively generates the *weak compositions* of A into exactly $|\mathcal{CI}|$ parts as tuples $a = (a_1, a_2, \ldots, a_{|\mathcal{CI}|})$. In the initial call OPTSEARCH(A, 1, y), the algorithm enumerates the possible values of $a_1 \in \{0, \ldots, A\}$ (Line 3). For every value of a_1 it tries to find the best implementation (minimum latency) of CI 1 requiring exactly a_1 area (Line 4). The recursive calls OPTSEARCH($A - a_1$, 2, y) take place only, if an implementation for a chosen a_1 is found. Otherwise, the whole recursion subtree for the value of a_1 is pruned. Every leaf of the recursion tree ($k = |\mathcal{CI}| + 1$) defines a unique selection y fulfilling Eq. (6.2) as well as Eq. (6.3) and is evaluated by solving the ILP of Eq. (6.8) (Line 12). Figure 6.5 visualizes how pruning is applied and how generated tuples correspond to selection candidates for the input provided by the example in Fig. 6.4. The effectiveness of our approach of pruning the search space by recursively generating weak compositions of A is evaluated in Section 6.6.4. While it shows effective in practice, the number of candidates to be evaluated can still grow exponentially in A and $|\mathcal{CI}|$. Therefore, a heuristic solution is presented in the following section.



Generate possibilities for $a = (a_1, 0)$ only that correspond to implementations of CI 1.

Based on a given $(a_1, 0)$, generate possibilities for $a = (a_1, a_2)$ (implementations of CI 2).

The subtree $(3,0) = (a_{1,2}, 0)$ is pruned ($\times$), because the respective implementation $j = 2$ of CI 1 requires the same area as $j = 1$, but does not provide a latency benefit, i.e., $a_{1,1} = a_{1,2} = 3 \wedge t_{1,1} < t_{1,2} \wedge r_{1,1} \leq r_{1,2}$. $(3,4)$ and $(3,5)$ are pruned, because $a_{1,j} + a_{2,j'} > A = 5$ (area constraint).
Finally, from theoretically $|\{0,1\}^{2\times4}| = 256$ possible inputs $y \in \{0,1\}^{2\times4}$ to the objective function, only

$$6 = \left| \left\{ \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}}_{\hat{=}a=(0,0)}, \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}}_{\hat{=}a=(3,0)}, \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{\hat{=}a=(0,2)}, \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}}_{\hat{=}a=(0,4)}, \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\hat{=}a=(0,5)}, \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{\hat{=}a=(3,2)} \right\} \right|$$

possible selections y need to be evaluated using timing analysis to find the optimal solution.

Figure 6.5: Visualization of how pruning is applied and how generated tuples correspond to selection candidates for the input provided by the example in Fig. 6.4. For clarity, tuples that were pruned because a chosen a_k did not correspond to a possible implementation of CI k are omitted.

6.5 Heuristic Solution

Algorithm 2 Greedy Heuristic for WCET-Optimizing Instruction Set Selection

```
1:  repeat
2:      if ∑_{k=1}^{|CJ|} a_{k,j(y_k)} = A then return        ▷ Return if reconfigurable area is fully occupied
3:      end if
4:      x ← result x which determines WCET'(y), see Eq. (6.5)        ▷ Get current worst-case path information
5:      y' ← y
6:      y ← UPGRSELECTION(y', x)        ▷ Attempt to upgrade a CI implementation
7:  until y' = y        ▷ Exit loop when unable to upgrade any CI
8:
9:  function UPGRSELECTION(y, x)
10:     profit^best ← 0, y^next ← y
11:     freePartitions ← A − ∑_{k=1}^{|CJ|} a_{k,j(y_k)}
12:     for k ← 1, |CJ| do
13:         profit ← -1, s ← 0
14:         while profit < 0 ∧ s < freePartitions do
15:             y'_k ← GETMINLATENCYIMPL(k, a_{k,j(y_k)} + s)        ▷ Try to find upgrade for CI k
16:             if y'_k ≠ 0 then        ▷ If upgrade with exactly s additional area found
17:                 y' ← (y_1, ..., y_{k−1}, y'_k, y_{k+1}, ..., y_{m_k})
18:                 profit ← profit(y'_k, y_k, x)        ▷ Defined in Eq. (6.9), profit > 0 ⇒ y'_k = y^+_k
19:             end if
20:             s ← s + 1
21:         end while
22:         if profit > profit^best then
23:             y^next ← y', profit^best ← profit        ▷ Save best y^+_k found so far
24:         end if
25:     end for
26:     return y^next
27: end function
```

We introduce a greedy heuristic that performs a number of WCET estimates linear in the number of partitions that the reconfigurable fabric area was divided in, i.e., maximal A estimates (for $A > 0$). It is shown in Algorithm 2.

The heuristic starts with implementing all CIs in software, i.e., not allocating any area of the reconfigurable fabric for CIs. For every CI, it assigns a profit which calculates the WCET reduction on the *current* worst-case path, when choosing an alternative implementation. Let $j(y_k)$ be the implementation selected for CI_k in y. We define the profit of selecting y'_k over y_k for a CI k as:

$$\mathrm{profit}(y'_k, y_k, x) := \underbrace{\sum_{\substack{i=1 \\ \mathrm{ci}(i)=k}}^{|\mathcal{B}|} (e_{i,j(y_k)} - e_{i,j(y'_k)})x_i}_{\text{latency reduction on current worst-case path}} - \underbrace{(r_{k,j(y'_k)} - r_{k,j(y_k)})}_{\text{additional reconfiguration delay}} \tag{6.9}$$

Where x provides information about the current worst-case path and is obtained by solving Eq. (6.5) and keeping the values of the variables x_i, i.e., x determines $\mathrm{WCET}'(y)$. Note that the profit can become negative if the latency reduction on the current worst-case path is smaller than the additional reconfiguration delay for the additional area. The heuristic calculates the profit for selecting the *next best implementation* y_k^+ instead of y_k for every CI k (Line 18). The *implementation* y_k^+ *for a CI* k is the implementation that can be chosen with minimum increase in the amount of area over y_k resulting in a positive profit. There might be several implementations according to this definition with the same required area. In this case y_k^+ is the implementation with minimum latency $t_{k,j}$ (and minimum j). If no such implementation y_k^+ exists (i.e., no implementation with positive profit was found), the CI is not considered for selecting a different implementation. Among the CIs for which y_k^+ exists, the algorithm greedily chooses the one with the maximum profit and upgrades y to select y_k^+ for the chosen CI k (Line 23). This process is repeated such that in every iteration the CI k with maximum $\mathrm{profit}(y_k^+, y_k, x)$ is upgraded. The algorithm terminates when no y_k^+ for any k exists anymore or insufficient area is left to be allocated for selecting y_k^+ (Line 7). In every iteration, either a CI upgrade is selected and the allocated area increased by a minimum of one or the algorithm terminates. For every iteration but the last one WCET estimate is performed. Therefore, a maximum of A WCET estimates are performed in total.

6.6 Experimental Evaluation

6.6.1 Evaluation Setup

This work is evaluated on the reconfigurable processor *i*-Core presented in Section 2.4. A CI is executed by a so-called CI Execution Controller. Its protocol is similar to other multi-cycle instructions like division and directly accesses register operands or non-cacheable Scratchpad Memory. This way, in microarchitectural analysis when determining WCETs of basic blocks, a CI is just another multi-cycle instruction that does not influence data cache analysis. The reconfigurable fabric is divided into A equally-sized partitions, complying to common models of allocating reconfigurable fabric area as assumed in our system model (see Section 6.2). The reconfiguration controller *CoRQ* (see Chapter 4) is employed with private memory to store configurations provides predictable reconfiguration of CIs. Initiating a specific configuration is done by a stores of the CPU using the memory-mapped interface of CoRQ (see Chapter 4 for more details).

Our timing analysis of tasks on reconfigurable processors has been detailed in Chapter 5 and was evaluated using the commercial timing analyzer AbsInt aiT [2]. In this work we extend WCET analysis as an integral part of WCET optimization. Therefore, the optimal search and heuristic selection algorithms were implemented as *processors* within the open-source WCET estimation framework OTAWA [7]. We extended the existing analysis support for the LEON3 CPU in OTAWA to support CI opcodes, CI super blocks with configuration-dependent latency and reconfiguration delay. Our approach was evaluated with the H.264 encoder application that was also used in the previous chapter. It uses 9 CIs covering the most compute-intensive kernels shown in Table 6.1. Multimedia applications in general are regularly subject to hard real-time constraints in the domain of computer vision. Notable

Table 6.1: Kernels and Custom Instructions (CI) in the H.264 Application

CI Name and Short Description	Working Set	#Confs.	Min. Part.	Max. Part.	CLoC[5]
MotionEstimation Kernel					
SATD: Sum of Abs. Transf. Differences	16×16 px	77	1	9	123
SAD: Sum of Abs. Differences	16×16 px	3	1	4	24
EncodeMacroBlock Kernel					
MC_Hz: Motion Compens. Interpol. Horiz.	4 px	29	1	6	51
IPred_HDC: Intra Prediction Horiz.	16×16 px	3	1	1	35
IPred_VDC: Intra Prediction Vert.	16×16 px	5	1	4	19
DCT: Discrete Cosine Transf.	4×4 px	21	1	5	76
HT2x2: Hadamard Transform	2×2 px	1	1	1	12
HT4x4: Hadamard Transform	4×4 px	17	1	4	111
LoopFilter Kernel					
LoopFilter: In-Loop Deblock. Filter	4 px	6	1	4	82

examples are advanced driver assistance systems, e.g., vehicle detection and tracking [15], but also consumer electronics, e.g., face recognition in digital cameras [109]. The H.264 encoder contains complex control flow with numerous decisions and nested loops. Most of the properties tested in the Mälardalen[3] or TACLeBench[4] WCET Benchmarks are covered, e.g., Discrete Cosine Transform is contained in all three. The H.264 decoder –that is part of TACLeBench– performs a subset of the computations performed in the H.264 encoder that is evaluated in the following. Especially the EncodeMacroBlock kernel stresses our selection heuristic (more details in Section 6.6.4), as it contains separate compute-intensive paths that share some CIs. The kernel iterates over macroblocks (MBs). Which path is executed within a kernel iteration depends on the type of MB, either *I-MB* or *P-MB*, determined by the MotionEstimation kernel, i.e., it is input dependent. I-MB and P-MB path also contain separate CIs leading to *instability of the worst-case path*, i.e., adding more partitions to the current worst-case path can result in the other path becoming the worst case. We compiled the application using BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1 and performed our selection on the encoder for a frame size of 99 MBs (QCIF resolution). At higher optimization levels, GCC emitted *irreducible loops*, i.e., complex loop structures that cannot be extracted as well-defined loop routines by the timing analyzer. Therefore, O1 provided the lowest WCET bound for the baseline executing all CI super blocks in software. The selection is performed offline and runs on a workstation with an AMD FX-6300 CPU and 12 GB of RAM. The result is used to generate a single configuration for every kernel that includes CI super blocks. The configurations are supplied to the optimized application on the target system by loading them into the private memory of CoRQ before executing the application. Before entering a kernel that includes CI super blocks, its specific configuration is triggered. The pipeline stalls for the reconfiguration delay and continues with entering the kernel once reconfiguration finishes.

The parameters evaluated were different numbers of partitions A (300 slices each on a Xilinx Virtex-7), reconfiguration bandwidths as well as relations of CPU frequency and fabric frequency $f_{\text{CPU}}/f_{\text{fabric}}$. Similar to the evaluation of the timing analysis in Section 5.6, f_{fabric} stays constant at 100 MHz and we choose multiples of it for f_{CPU} that resemble realistic setups. E.g., running the CPU at $f_{\text{CPU}} = 400$ MHz, which the LEON3 CPU is advertised as running at as an ASIC implementation, would correspond to the parameter $f_{\text{CPU}}/f_{\text{fabric}} = 4$. The successor of the LEON3, LEON4 is advertised running at 1500 MHz, corresponding to $f_{\text{CPU}}/f_{\text{fabric}} = 15$. The commercially available Xilinx Zynq-7000 SoC couples an ARM Cortex A9 at 866 MHz with a Xilinx 7-Series reconfigurable fabric, corresponding to $f_{\text{CPU}}/f_{\text{fabric}} \approx 9$. Note that while the WCET in seconds (WCET cycles$/f_{\text{CPU}}$) is anticipated to

[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[4] http://www.tacle.eu/index.php/activities/taclebench
[5] C Lines of Code that are replaced by utilizing a hardware CI (without comments or whitespace)

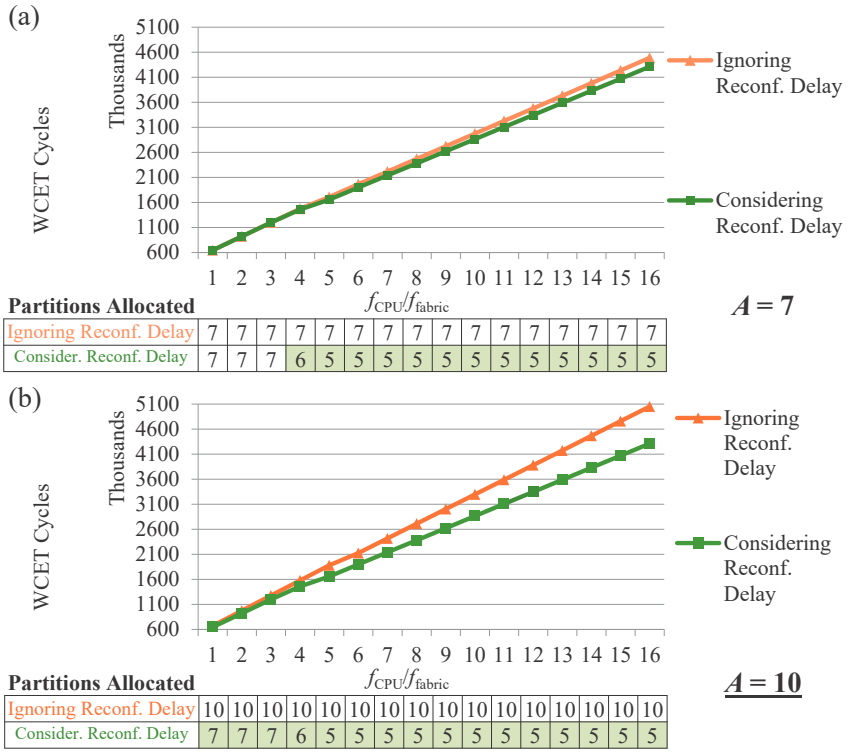


Figure 6.6: Optimal Results for the `EncodeMacroBlock` Kernel of the H.264 Encoder and different Values of f_{CPU}/f_{fabric}, A as well as a Reconfiguration Bandwidth of 200 MB/s. Comparing Results considering Reconfiguration Delay during Selection and Results not considering it.

get lower (better) with higher f_{CPU}, the WCET cycles are increasing (at a constant f_{fabric}), because hardware CIs perform less computations on the reconfigurable fabric within one CPU cycle.
In Sections 6.6.2 and 6.6.3, we focus on the effects of considering reconfiguration delay and infeasible path information on the selection result, respectively. In Section 6.6.4, the effectiveness of pruning during optimal search is analyzed and runtime as well as quality of selection results of our optimal search and heuristic algorithms compared. Note that all discussed results are upper bounds of the actual WCET. In general, it is not possible to obtain the actual WCET [106].

6.6.2 Impact of Reconfiguration Delay on WCET-Optimizing Selection

In this section we evaluate the impact of reconfiguration delay on WCET-optimizing CI selection. Figure 6.6 shows results obtained by applying our optimal search algorithm (see Section 6.4) to the `EncodeMacroBlock` kernel of the H.264 Encoder for $f_{CPU}/f_{fabric} \in [1:16]$ and reconfiguration bandwidth of 200 MB/s (half of the theoretical maximum in current Xilinx FPGAs). We compare the results obtained by considering the reconfiguration delay during selection, as in Eq. (6.8), with results obtained by ignoring it (i.e., $r_{k,j} = 0 \; \forall k \in \mathcal{CI}, j \in m_k$). The final WCET bound always includes the reconfiguration delay required to configure the selection result.
Figure 6.6 (a) shows the results for $A = 7$, i.e., the algorithm can allocate up to 7 partitions for the selection to optimize the WCET of this kernel. For $f_{CPU}/f_{fabric} \in [1:3]$, the selections and the resulting WCET bound are equal. For higher frequencies of the CPU, the WCET bound obtained by ignoring the reconfiguration delay during selection is higher than the WCET bound obtained by considering the reconfiguration delay with a maximum of 4.08 % increase at $f_{CPU}/f_{fabric} = 16$. More importantly, the lower WCET bounds are obtained with *fewer partitions*. It is not beneficial to use all 7 partitions with $f_{CPU}/f_{fabric} \in [4:16]$, because the CIs having the biggest effect on reducing the WCET bound are implemented in hardware already. Increasing the number of allocated partitions for these CIs yields diminishing returns in their latency reduction. In total, this leads to an increase of the

WCET bound, because the additional reconfiguration delay outweighs the latency reduction of the WCET path. This effect becomes even more apparent, with $A = 10$ as shown in Fig. 6.6 (b), keeping all other parameters as in Fig. 6.6 (a). In this case, ignoring the reconfiguration delay already yields a higher WCET bound by 4.02 % at $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and up to 17.14 % at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ over considering the reconfiguration delay. Furthermore, at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ only half the partitions are required when considering the reconfiguration delay (5 partitions, as compared to 10 when ignoring it). The effect of obtaining lower WCET bounds with fewer partitions when considering the reconfiguration delay during selection compared to not considering it, becomes more severe with higher reconfiguration delay (measured in CPU cycles) per allocated partition. The reconfiguration delay per partition increases when $f_{\text{CPU}}/f_{\text{fabric}}$ is increased (e.g., when using a higher-frequency CPU) or the reconfiguration bandwidth is lowered (e.g., when using cheaper memory). Additionally, raising A further would again lead to worse selections when not considering the reconfiguration delay, as more partitions would be allocated for only little CI latency improvement.

In sum, not considering the reconfiguration delay during WCET-optimizing CI selection can not only lead to suboptimal results. It can lead to *higher* WCET bounds allocating *more* partitions than required in the optimal results (considering reconfiguration delay). Existing approaches for selecting CIs to optimize the WCET, target application-specific instruction set processors (ASIPs) instead of reconfigurable processors, and therefore do not consider reconfiguration delay (see Section 6.1). While runtime reconfiguration provides the flexibility to utilize the whole fabric area per kernel and was proven to provide substantial WCET reductions [29], the reconfiguration delay needs to be considered during selection to avoid suboptimal results. Previous approaches targeting ASIPs can therefore not be applied to reconfigurable processors as the results obtained by our approach show.

6.6.3 Impact of Infeasible Path Information on WCET-Optimizing Selection

As motivated in Fig. 6.2, previous WCET-optimizing selection and allocation approaches relying on timing schema cannot utilize information about the global program flow. Therefore, global flow information provided by annotation languages in state-of-the-art timing analyzers (see [59] for an overview), which is crucial to precise WCET bounds, cannot be utilized during optimization and therefore decisions are made on imprecise WCET estimates. In the evaluations of our approach, the CFG was annotated with infeasible path information using the XML-based FFX language [17] supported by OTAWA. During WCET bound estimation, these annotations are translated into IPET constraints (see Section 2.2.1). Similar to all other IPET constraints used in our optimization approach, the constraints need to be generated once for the whole optimization process and can be reused for all WCET bound estimations.

Figure 6.7 shows results with and without infeasible path information obtained by applying the optimal search algorithm (see Section 6.4) to the `EncodeMacroBlock` kernel of the H.264 Encoder for several parameters. For evaluating the effects of infeasible path information, the path encoding I-MBs (see Section 6.6.1) is annotated as infeasible, which becomes the worst-case path only at some point when adding partitions to CIs (the exact point depends on $f_{\text{CPU}}/f_{\text{fabric}}$ and the reconfiguration bandwidth). For most selections and especially when not allocating any partitions (original software instead of hardware CIs only), the P-MB path is the worst-case path. Still, we can show that annotating the I-MB path as infeasible has a considerable effect on the resulting WCET bound. A reconfiguration bandwidth of 400 MB/s –the theoretic maximum in Xilinx Virtex-7 FPGAs– is used in Fig. 6.7 (a) for allocating $A = 5$ partitions. At $f_{\text{CPU}}/f_{\text{fabric}} = 1$, the difference between optimized WCET bound with and without infeasible path information is maximal with 12.71 % more WCET cycles when not utilizing the infeasible path information. The additional WCET cycles are a result of allocating partitions to CIs that lie on the path marked as infeasible. Our approach enables to utilize this information during optimization and therefore does not allocate any partitions to the infeasible path when this information is provided. For $f_{\text{CPU}}/f_{\text{fabric}} \in [2:4]$, the difference decreases down to 3.45 % at $f_{\text{CPU}}/f_{\text{fabric}} = 4$, because the increased speed of the CPU relative to the

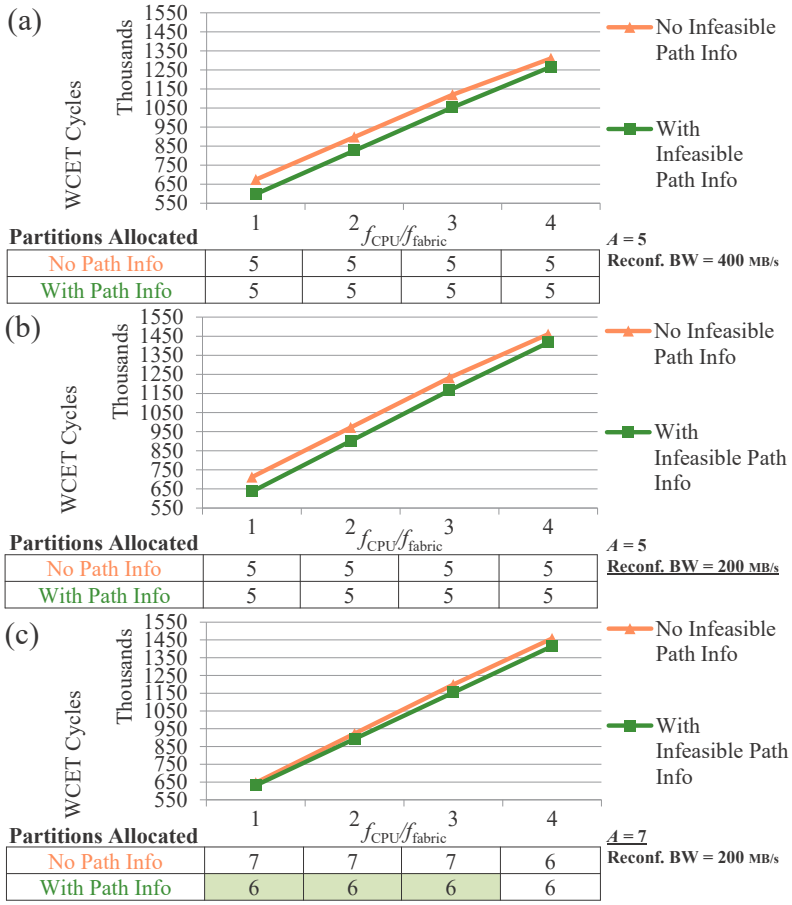


Figure 6.7: Optimal Results for the `EncodeMacroBlock` Kernel of the H.264 Encoder and different Values of f_{CPU}/f_{fabric}, A as well as Reconfiguration Bandwidth. Comparing Results utilizing Infeasible Path Information (I-MB Path marked infeasible) and not utilizing it.

fabric compensates the partition wasted on the infeasible path when not utilizing the global flow information. One might suspect that this effect is only possible at this high reconfiguration bandwidth, because the reconfiguration delay of adding an additional partition to the worst-case path while utilizing infeasible path information might be too high to reduce the WCET bound otherwise (see Section 6.6.2). However, with half the reconfiguration bandwidth (200 MB/s), the results remain valid with 11.95 % additional cycles at $f_{CPU}/f_{fabric} = 1$ and 3.08 % at $f_{CPU}/f_{fabric} = 4$ when comparing the selection not utilizing WCET path information with the selection that does. For higher values of f_{CPU}/f_{fabric} than 4, the instability of the worst-case path leads to the infeasible path not appearing for $A = 5$. Keeping the reconfiguration bandwidth at 200 MB/s and increasing A leads to an additional effect shown in Fig. 6.7 (c). The difference between the WCET bounds obtained with infeasible path information and without is generally lower, with maximal 3.90 % additional WCET cycles at $f_{CPU}/f_{fabric} = 3$. The reason is that with infeasible path information the optimal choice only allocates 6 out of $7 = A$ available partitions, the reconfiguration delay of adding an additional partition to the worst-case path is now too high to effectively reduce the WCET bound. Adding an additional partition to the infeasible path when not utilizing this information therefore adds reconfiguration delay to the WCET bound without providing actual benefit. Therefore, similar to the impact of reconfiguration delay evaluated in Section 6.6.2, utilizing infeasible path information during WCET-optimizing CI selection provides better results and can even provide *better results with fewer partitions*. Previous WCET-optimizing approaches for CI selection and memory allocation relied on timing schema and were therefore unable to utilize global flow information (see Section 6.1). However, utilizing this information is crucial to obtain good selection results.

Table 6.2: Evaluation Results `LoopFilter` Kernel, $|\mathcal{CJ}| = 1$

Measures \ A	**0**	**1**	**2**	**3**	**4**
Total Possible Selections	7	7	7	7	7
Weak Comp. of A into $\lvert\mathcal{CJ}\rvert$	1	2	3	4	5
Opt. Estimates	1	2	3	4	5
Heur. Estimates	1	1	2	3	3
Opt. Runtime [ms]	122.0	123.2	119.2	120.0	124.4
Heur. Runtime [ms]	118.0	112.4	122.4	122.4	119.2
WCET unoptimized [cycles]	4,467,172				
Opt. Speedup	1	1	19.8516	19.8516	19.8516
Heur. Speedup	1	1	19.8516	19.8516	19.8516

While our approach enables utilizing global flow information and considering the reconfiguration delay during optimization, it adds complexity by utilizing IPET over simpler techniques like timing schema to obtain WCET estimates. In the following, the quality of the results of our heuristic compared to optimal search as well as runtimes are evaluated to demonstrate the practicality of our approach.

6.6.4 Runtimes, Pruning and Quality of Heuristic Selection

Sections 6.6.2 and 6.6.3 demonstrated the importance of considering reconfiguration delay as well as global program flow information during WCET optimization. In contrast to previous approaches, our approach enables considering both types of information. However, this requires evaluating several IPET instances and therefore raises the question whether the runtimes of the optimization remain within acceptable bounds.

Tables 6.2 to 6.5 show evaluation results for all major kernels of the H.264 encoder application. We fixed $f_{\text{CPU}}/f_{\text{fabric}}$ at 4 and a reconfiguration bandwidth of 400 MB/s, as it reflects the realistic setup of running the CPU at 400 MHz (which the LEON3 processor is advertised as running at when implemented as an ASIC) and the reconfigurable fabric at 100 MHz, as well as running the configuration port at its maximum speed. The scalability and effectiveness of pruning during the optimal search as well as the heuristic is evaluated by running the algorithms for $A \in [0:21]$. Giving optimal search the freedom to allocate up to $A = 21$ partitions results in the maximum number of candidates for the most complex kernel `EncodeMacroBlock` (see Table 6.5). The maximum number of candidates is reached for lower A for the `MotionEstimation` ($A = 11$) and `LoopFilter` ($A = 4$) kernels, the additional measurements for these kernels are therefore omitted. Table 6.2 to Table 6.5 are in increasing order of kernel complexity (number of instructions and number of CI super blocks). The first line of the tables is the total number of possible selections calculated as $\prod_{k \in \mathcal{CJ}}(m_k + 1)$, i.e., the number of all combinations of configurations, plus the original software implementation, per CI without any restrictions. Weak compositions of A were explained in Section 6.4 as a technique we apply for pruning selection candidates during optimal search. The number of weak compositions of A into exactly $|\mathcal{CJ}|$ parts are calculated as $\sum_{s=0}^{A} \binom{s+|\mathcal{CJ}|-1}{|\mathcal{CJ}|-1}$ [51]. *Opt. Estimates* and *Heur. Estimates* are the number of WCET estimates calculated using Eq. (6.6) during optimization using optimal search and the heuristic, respectively. The last lines of the tables are the runtimes of the optimizations and the speedups obtained on the WCET estimate of the kernel, comparing the selection result to the software-only implementation.

Effectiveness of Pruning and Scalability of Optimal and Heuristic Selection

Table 6.2 shows the results obtained for the evaluated kernel of least complexity, `LoopFilter`. The kernel includes one CI super block only, with 7 implementation alternatives and therefore 7 possible selections in total. For only

Table 6.3: Evaluation Results `MotionEstimation` Kernel, $|\mathcal{CJ}| = 2$

Measures \ A	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Total Possible Selections	312	312	312	312	312	312	312	312	312	312	312	312
Weak Comp. of A into $\lvert\mathcal{CJ}\rvert$	1	3	6	10	15	21	28	36	45	55	66	78
Opt. Estimates	1	2	3	4	5	6	7	8	9	10	29	30
Heur. Estimates	1	1	2	3	4	5	6	7	8	9	10	11
Opt. Runtime [ms]	4360.0	4350.8	4318.4	4329.2	4360.4	4331.2	4375.2	4391.2	4398.8	4456.8	4373.2	4317.2
Heur. Runtime [ms]	4291.6	4303.2	4332.0	4317.2	4337.6	4309.6	4342.4	4361.6	4323.6	4405.6	4344.0	4328.8
WCET unoptimized [cycles]	112,173,893											
Opt. Speedup	1	4.82	5.48	9.04	21.01	24.67	25.65	26.70	26.70	26.97	26.97	27.24
Heur. Speedup	1	4.82	5.48	9.04	21.01	24.67	25.65	26.70	26.70	26.97	26.97	27.24

Table 6.4: Evaluation Results `EncodeMacroBlock` Kernel, $|\mathcal{CJ}| = 6$

Measures \ A	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Total Possible Selections	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240
Weak Comp. of A into $\lvert\mathcal{CJ}\rvert$	1	7	28	84	210	462	924	1716	3003	5005	8008
Opt. Estimates	1	2	3	4	5	299	517	816	1192	1629	2100
Heur. Estimates	1	1	2	3	4	5	6	7	8	9	10
Opt. Runtime [ms]	4568.4	4635.6	4757.6	5154.4	5923.2	7194.4	9568.8	12331.6	15721.2	19550.4	23131.2
Heur. Runtime [ms]	4540.8	4546.0	4556.4	4547.2	4561.6	4580.4	4605.2	4647.6	4747.2	4731.2	4784.8
WCET unoptimized [cycles]	13,348,251										
Opt. Speedup	1	2.43	3.35	4.66	9.84	10.19	10.44	10.55	10.62	10.69	10.69
Heur. Speedup	1	2.43	3.35	4.66	9.84	9.94	10.29	10.55	10.62	10.69	10.69

Table 6.5: Evaluation Results `EncodeMacroBlock` Kernel, $|\mathcal{CJ}| = 6$ (continued)

Measures \ A	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
Total Possible Selections	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240
Weak Comp. of A into $\lvert\mathcal{CJ}\rvert$	12376	18564	27132	38760	54264	74613	100947	134596	177100	230230	296010
Opt. Estimates	2571	3008	3384	3683	3901	4045	4130	4174	4193	4199	4200
Heur. Estimates	10	10	10	10	10	10	10	10	10	10	10
Opt. Runtime [ms]	27230.4	31072.8	34457.6	37087.2	38973.6	40324.4	41037.6	41725.2	41686.8	41751.6	41695.2
Heur. Runtime [ms]	4761.6	4740.0	4753.2	4802.0	4764.4	4792.8	4780	4592.0	4608.0	4608.4	4597.6
WCET unoptimized [cycles]	13,348,251										
Opt. Speedup	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69
Heur. Speedup	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69

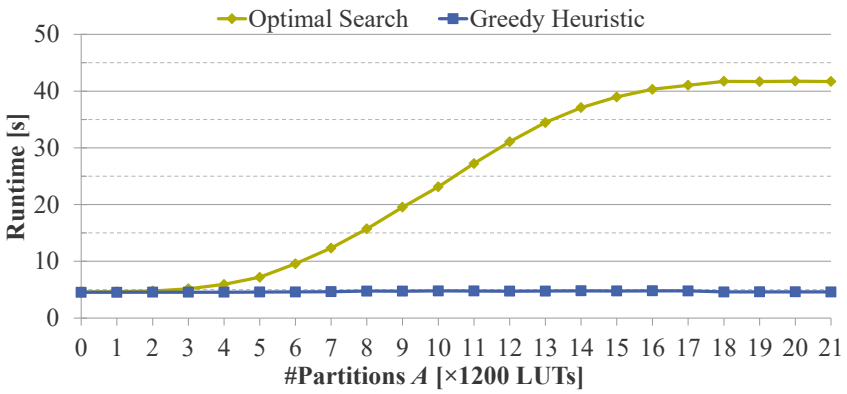


Figure 6.8: Visualization of the runtime results of the optimization approaches when applied to the `EncodeMacroBlock` kernel ($|\mathcal{CI}| = 6$)

one CI, the optimal search performs as many WCET estimate calculations as the number of weak compositions of A into exactly $|\mathcal{CI}|$ parts (denoted as *number of compositions* for the remainder of this text) until $A = 4$. For higher A the number of estimates remains constant, as no possible implementation for the CI requiring more than 4 partitions exists. The heuristic never performs more than 3 estimates while the optimal search performs maximal 5, however, the complexity of the kernel is too low to show any measurable runtime effect.

The kernel of next higher complexity is `MotionEstimation`, its evaluation results are shown in Table 6.3. It includes two CI super blocks having 4 and 78 different implementations, for a total of 312 possible selections. This is enough, to demonstrate the effectiveness of pruning by finding selections which correspond to weak compositions of A (see Section 6.4), as even at $A = 11$ the 78 possible compositions is only a quarter of the total number of possible selections. The additional pruning of the search space in our recursive search further reduces the search space to 30 candidates which is 9.62% of the total selection candidates and 38.46% of the number of compositions. The heuristic further reduces the number of WCET estimations performed to 11, 36.67% of the estimations performed by the optimal search, the runtime benefit is barely measurable, however.

`EncodeMacroBlock` is the most complex kernel in our evaluation. It contains 6 CI super blocks resulting in a total of 570,240 possible selections. The results are shown in Table 6.4 and Table 6.5. Again, finding selections which correspond to weak compositions of A prunes the search space effectively, but for $|\mathcal{CI}| = 6$ the number of possible compositions of A already grows rapidly, reaching 51.91% of the total number of estimates at $A = 21$. However, the number of estimates calculated during optimal search stays much lower with a maximum of 4200 at $A = 21$, which is 0.74% of the total number of possible selections. This is possible by pruning recursive subtrees early in our optimal search algorithm. Still, the runtime[6] of the optimal search algorithm does not scale well with increasing A as visualized in Fig. 6.8. At $A = 0$ it takes 4.5s, doubling already at $A = 6$ with 9.0s, again doubling at $A = 9$ to 18.88s. For higher values of A the runtime growth stagnates, reaching its maximum of 41.43s at $A = 21$. Especially because there may be numerous kernels within the application under optimization, these runtime values may hinder design space exploration. To reduce the optimization runtime at the potential cost of quality of the result (discussed in the following section), the heuristic can be applied. It performs a maximum of 10 estimate calculations, leading to a runtime of maximal 4.63s. 4.25s of this runtime are spend in CFG reconstruction and microarchitectural analysis, which are preparation steps to WCET bound estimation before any optimization can take place. Therefore, *solving ILPs for estimate calculations during optimization only takes a fraction of the total runtime* of the heuristic. As the dominating part of the runtime –the microarchitectural analysis– only needs to be performed once, an additional estimate required roughly under 40ms. This value is often dominated by the noise between measurements. Thus, extending IPET for precise WCET estimates during WCET optimization can be suitable for design space exploration as our results show.

[6] All runtime results were computed as the average of 10 median values from 12 measurements

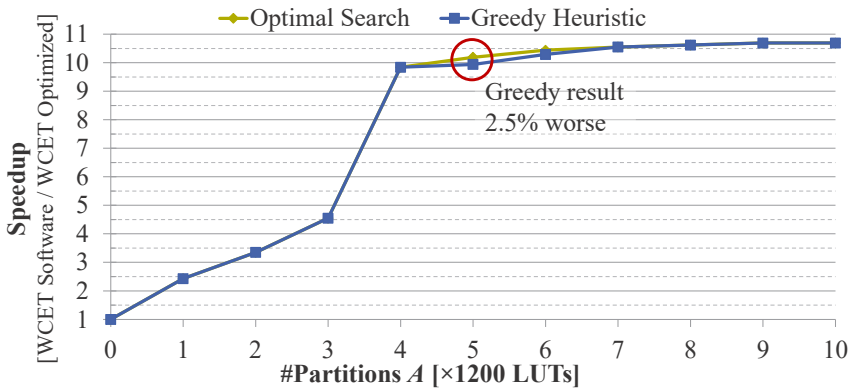


Figure 6.9: Visualization of the speedup results of the optimization approaches when applied to the `EncodeMacroBlock` kernel ($|\mathcal{CJ}| = 6$)

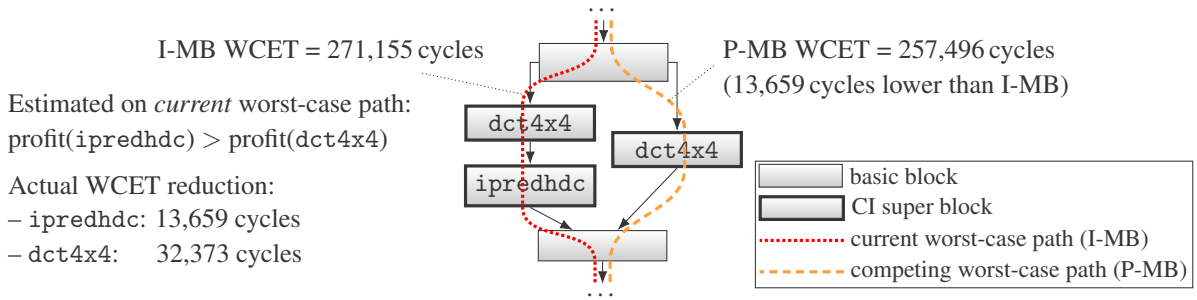


Figure 6.10: The greedy heuristic is unaware of any "competing" worst-case paths (P-MB path here). Thus, the estimated profit on the *current* worst-case path (I-MB) can be higher than the actual WCET reduction as in this case that appears when optimizing `EncodeMacroBlock` at $A = 5$ (simplified)

In sum, the pruning techniques for the optimal search algorithm have shown very effective, but can still lead to runtimes unsuitable for design space exploration. In these cases the heuristic can reduce the runtime down to 11.18% of the optimal search algorithm. However, the heuristic can lead to suboptimal results in certain cases, which we will detail in the following section.

Quality of Heuristic Selection

For the kernel of least complexity, `LoopFilter`, and medium complexity, `MotionEstimation`, our heuristic as well as optimal search always find the same solution for all values of A as shown in Table 6.2 and Table 6.3, respectively. Therefore, we focus on the `EncodeMacroBlock` kernel and the evaluation results, which exhibit heuristic selections different from optimal search as highlighted with yellow background in shown in Table 6.4 and visualized in Fig. 6.9. More specifically, the heuristic finds selections that produce 2.52% and 1.46% lower speedups at $A = 5$ and 6 than the optimal solution, respectively (while finding the optimal solution in all other cases). The reason for this is the calculation of the profit function in Eq. (6.9), which tries to estimate the effect of a CI implementation on a previously calculated WCET bound. The problem is that the profit is calculated for the *current* worst-case path. Due to the instability of the worst-case path, adding a CI implementation y'_j that benefits the WCET bound can have a smaller effect on the total bound than on the current worst-case path. However, this is not sufficient for the heuristic to make a suboptimal choice. Additionally, a CI implementation is needed that was assigned a lower profit than y'_j, but actually has a higher effect on the total WCET bound than y'_j. This can happen when a CI configuration can appear in the current worst-case path as well in the next longest path in the program. This is the case for the `dct4x4` CI within the `EncodeMacroBlock` kernel. The case is visualized in simplified form in Fig. 6.10. E.g., for $A = 5$ the heuristic chooses the suboptimal selection as follows: after allocating 4 partitions for the P-MB path, the I-MB path becomes the worst-case path. The heuristic calculates a profit of 257,496 cycles

for implementing `ipredhdc` in hardware, allocating the last partition. However, the I-MB path takes only 13,659 cycles longer than the P-MB path at this point. Therefore, implementing `dct4x4` with a profit of 46,032 cycles in hardware and effectively reducing the WCET bound by 32,373 cycles, because it appears in the P-MB as well as the I-MB path would have been the better choice. For $A < 5$, the I-MB path never becomes the worst-case path. For $A > 6$, the heuristic has enough partitions available to fully compensate for the suboptimal decision. Therefore, the heuristic finds the optimal results in these cases.

6.7 Conclusion

This chapter presented how timing analysis using IPET can be extended to perform WCET optimization on runtime-reconfigurable processors. The WCET-optimizing instruction set selection problem was formulated, i.e., selecting the WCET-optimal set of reconfigurable custom instruction implementations. Techniques for generating and pruning potential instruction set selections were discussed and realized in an optimal search algorithm. The effectiveness of pruning in our optimal search algorithm was demonstrated, it only needed to evaluate less than 1% of all possible 570,240 selections when optimizing the `EncodeMacroBlock` kernel as part of the H.264 encoder. However, as the optimal search algorithm was still not scaling well for large problem instances, a heuristic was introduced which performs maximally as many evaluations as there are partitions to allocate on the reconfigurable fabric. The heuristic was an order of magnitude faster than the optimal search for the previously mentioned kernel. Additionally, an analysis of suboptimal solutions (up to 2.52% lower speedup) obtained from the heuristic in our evaluation, showed that they are a result of competing worst-case paths during optimization that share CIs. Our problem formulation and algorithms were implemented based on the timing analyzer OTAWA, showing the seamless integration into a state-of-the-art timing analysis tool.

The consequences of utilizing timing schema in WCET optimization were shown, a WCET estimation technique that does not support global program flow information, but is still commonly used in state-of-the-art WCET optimization approaches. Our novel problem formulation of WCET optimization enables considering global program flow information such as reconfiguration delay during optimization and the importance of considering these information was demonstrated. Not considering global information can lead to higher WCET bounds that require more resources than the optimal solution.

In sum, novel WCET optimization approaches were provided and runtime instruction set reconfiguration was shown to be an enabling feature for timing-predictable performance. To the best of our knowledge, our model is the first formulation of a WCET optimization problem with support for global program flow information and we can envision applications to problems other than instruction set extension.

In the following, the static WCET optimization presented in this chapter is complemented by an online optimization that targets average-case performance while maintaining WCET guarantees.

7 WCET Guarantees for Opportunistic Runtime Reconfiguration

As presented in the previous chapters, recent works have demonstrated that runtime reconfiguration of hardware accelerators is a viable way to achieve high performance that is analyzable for execution time guarantees [16, 29, 77, 89]. The latency of a reconfigurable hardware accelerator is under direct control of an application designer. It is often precisely known, e.g., when leveraging high-level synthesis tools. Where in average-case optimizing systems the reconfigurable area is allocated to accelerators that result in the best speedup *on average*, the main constraint in real-time systems is to statically guarantee WCET bounds. Thus, reconfigurable area is allocated to accelerators that reduce the execution time of the statically-determined worst-case path of a task (like in the previous chapter). However, executing the worst-case path and completing in WCET is usually highly improbable (see Chapter 1). In fact, WCET analysis approximates the WCET of a task by an upper execution time bound and thus, the actual runtime of the task will virtually *always* be faster than the guaranteed WCET as shown in Fig. 7.1 (also see Section 2.2). Ultimately this means that (1) the average-case execution time (ACET) of a task is generally considerably lower than the WCET and (2) configuring accelerators to optimize the WCET of a task can waste reconfigurable area on program paths that might never be executed in practice. It is, however, highly desirable to achieve a high utilization of accelerators and optimize average-case execution to fulfill additional non-functional constraints like, e.g., power or thermal constrains.

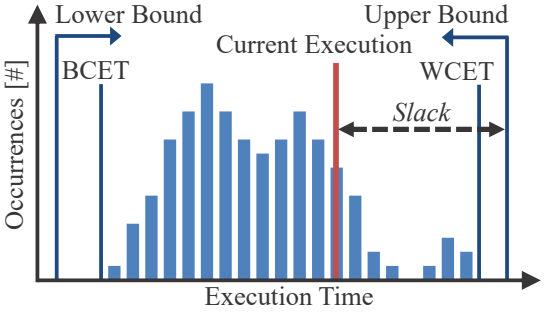


Figure 7.1: The WCET of a task is upper-bounded using static timing analysis. At runtime, *slack* towards this upper bound becomes a resource that can be leveraged, e.g., for runtime reconfiguration of accelerators on an FPGA.

The novel contributions of this chapter are as follows:

- an approach to optimize static WCET guarantees as well as runtime optimization of the ACET (maintaining WCET guarantees) using runtime reconfiguration of hardware accelerators (in the form of Custom Instructions as presented in Section 2.4 and Chapter 6)
- analysis of runtime slack bounds that enable safe reconfiguration for average-case performance under WCET guarantees
- an approach for monitoring the *runtime slack* of a task (the amount of time it executed parts of code faster than in worst-case) using simple performance counters

7.1 Related Work

While runtime reconfiguration of accelerators is an established concept in embedded systems in general [97], it gained traction in recent years for achieving performance in systems that need to fulfill hard real-time guarantees [72]. Several works in this direction are concerned with scheduling task sets that employ runtime reconfiguration [16, 54, 89]. The authors of [89] present scheduling strategies and admission tests for periodic hard real-time

tasks that occupy area on a runtime-reconfigurable fabric for the duration of their execution. Fully and partially reconfigurable fabrics are considered, where partitions allocated to tasks are restricted to be equally-sized. In [16] an overview over state-of-the-art real-time scheduling for reconfigurable systems is provided, and a scheduling framework as well as an analysis for periodic multi-priority real-time tasks is presented. The presented model divides tasks into software and hardware subtasks, where hardware subtasks model the execution of hardware accelerators on a reconfigurable fabric. Software subtasks request execution of hardware subtasks and self-suspend until the hardware subtask has finished execution. The requests for hardware subtask execution trigger runtime reconfiguration of the respective hardware accelerator onto a partitioned reconfigurable area.

Whereas the previous scheduling approaches assume that each task provides a given set of hardware accelerators that need to be configured, the work of [54] addresses the problem of finding a set of configurations (of a processor with a reconfigurable instruction set) for periodic task graphs that enable timing constraints to be met. A periodic task graph with deadlines is scheduled and the schedule is partitioned into configurations for the reconfigurable fabric. Each configuration is assigned an instruction set to optimize the tasks' WCET. Further work on finding sets of hardware accelerators that minimize the WCET of a task are available for non-reconfigurable [112] and reconfigurable processors (see previous chapter). While all of the above mentioned approaches utilize runtime reconfiguration to fulfill real-time constraints, none of the approaches considers online optimization of average-case execution once constraints are met. Thus, they are unable to optimize for dynamic workloads like signal processing or computer vision applications.

When a task executes faster than in the worst case, the execution time difference between the guaranteed WCET and the current execution, i.e., the *runtime slack* (see Fig. 7.1), becomes a resource that can be used to optimize additional constraints. Several works on runtime slack exploitation target dynamic voltage scaling to reduce the overall energy consumption of the system [62, 91]. Furthermore, opportunistic monitoring for security or reliability issues has been presented [67]. Orthogonal to these works, runtime slack was used to optimize the ACET using a complex (i.e., hard to analyze but high-performance) microarchitecture that provides a simple, timing-analyzable architecture mode [5]. In case insufficient runtime slack is detected at runtime, the architecture enters the simple mode that is the basis for WCET guarantees. However, this approach does not provide optimization of WCET guarantees.

In summary, state-of-the-art approaches either do not consider runtime slack but focus on utilizing runtime reconfiguration of accelerators for optimizing worst-case execution only, or they do consider runtime slack but not for reconfiguration of accelerators. In this chapter, both properties are achieved for the first time, i.e., an approach is presented that optimizes worst-case execution using accelerators *and* considers runtime slack as a resource for online optimization of the average-case execution using reconfiguration.

7.2 System Model

As in previous chapters, this chapter focuses on runtime-reconfigurable processor designs in which the *core instruction set architecture* (cISA) of the processor core is extended by *custom instructions* (CIs) (see Section 2.4). CIs initiate the execution of (one or more) hardware accelerators on the reconfigurable fabric. To model the use of reconfigurable CIs inside the control flow graph (CFG) of a task during WCET analysis, we base on the "stalling" model of Chapters 4 and 5, which is summarized in Fig. 7.2 and Fig. 7.3 for the context of this chapter. Each kernel in the stalling model is preceded by a basic block that initiates reconfiguration of CIs, which are used within the kernel body (x_{reconf} in Fig. 7.3). During reconfiguration the task waits, i.e., it only proceeds its execution after the reconfiguration delay has passed. The reconfiguration delay depends on the size of the configuration data and the bandwidth of the reconfiguration port. Afterwards, the CPU can execute the kernel and utilize hardware accelerators by invoking CIs as shown in Fig. 7.2 (bottom). The functionality of a CI (x_{HW_i}) is alternatively available as a software implementation (x_{SW_i}), e.g., the software implementation that the CI was derived from when

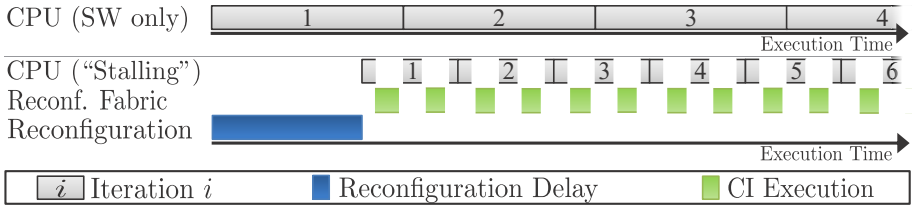


Figure 7.2: Visualization of execution without utilizing acceleration (top) and with configuring CIs before executing a kernel (bottom).

using high-level synthesis (we assume the software implementation will always have a higher latency than the hardware-accelerated CI). This way, more opportunities to generate CIs can be identified at compile time than actually fit onto the reconfigurable area of the specific target platform (as detailed in the previous chapter). The *selection* problem, i.e., choosing a subset of CIs to optimize certain criteria, was addressed for optimizing WCET guarantees in the previous chapter. At runtime, a conditional branch tests whether a specific CI was configured and is available in hardware. If this is the case, the functionality is executed using hardware accelerators, and in software otherwise.

The following section gives a high-level summary of how selection can be solved for WCET-optimizing configurations and a brief description of selection for performance-optimizing configurations based on the utilized model. Afterwards, extensions to the model are presented that enable multiple reconfigurations within a kernel to switch between different configurations at runtime.

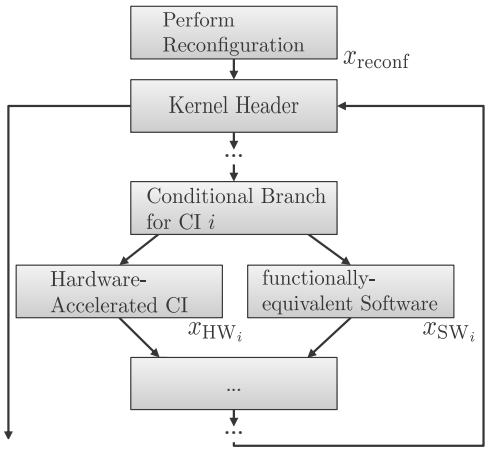


Figure 7.3: CFG of a kernel that configures and utilizes reconfigurable CIs in the stalling model. Not all utilized CIs need to be configured (constrained area), but can be executed in a functionally-equivalent software implementation.

7.3 Our Approach

The main idea of this chapter is to employ two configurations for the reconfigurable fabric: a *safe configuration* (a selection of CIs that optimizes the WCET bound like in Chapter 6) and a *performance configuration* (a selection of CIs that optimizes the ACET). At the start of a kernel, the reconfigurable area is configured using the safe configuration. During its execution the task's slack towards the WCET guarantee, i.e., the amount of time it executed parts of code faster than in worst case, is continuously sampled and accumulated. Once sufficient slack is accumulated, the performance configuration is configured onto the reconfigurable area and the average-case execution is accelerated. The performance configuration has a higher WCET bound than the safe configuration, and in the worst case it could experience a slower execution than what was guaranteed, i.e., the accumulated slack could be reduced. Therefore, special care needs to be taken for the case that the slack might be depleted. We might need to switch back to the safe configuration to avoid this case, and we need sufficient slack left not to violate the guaranteed WCET bound. With high probability, however, the execution of the performance configuration will be faster (it optimizes the average case) and the task will finish with a lower execution time than when executing in the safe configuration.

The approach comprises an offline preparation phase in which the safe and performance configurations are created and their information is annotated to the task under optimization, as well as the actual online optimization phase that performs reconfigurations within the WCET bound using runtime slack.

7.3.1 Offline Preparation

The input to the offline preparation is the reconstructed control-flow graph (CFG) of the task's binary, which is obtained as the first step during WCET analysis (see Section 2.2).

Safe Configuration

Several methods to obtain a suitable safe configuration exist as detailed in the previous chapter. In Section 6.5 a greedy algorithm was presented, which is briefly summarized as follows (see Chapter 6 for details, especially supporting multiple hardware implementations per CI and the influence of reconfiguration delay on the selection result, which are ignored here for brevity). The algorithm repeatedly performs WCET estimation of the CFG and CI selection using the stalling model as shown in Fig. 7.4. The first WCET estimate will result in a worst-case path that does not utilize any hardware-accelerated CI: because the software implementation (x_{SW_i} in Fig. 7.3) has a longer latency than the CI, it is the worst-case choice. After each WCET estimate the CI that provides the maximum profit on the current worst-case path is selected and added to the temporary safe configuration, where the profit of a CI a is estimated as:

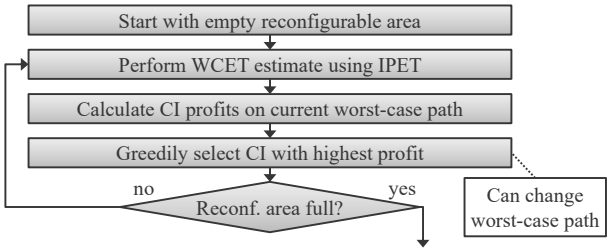


Figure 7.4: CIs are selected to obtain the safe configuration. Each time a CI is selected, the WCET needs to be estimated again

$$\mathrm{profit}_{\mathrm{WCET}}(a) = \sum_{x_i \text{ calls functionality of } a} x_i \cdot (\mathrm{latency}_{\mathrm{sw}} - \mathrm{latency}_{\mathrm{hw}})$$

I.e., the total profit of selecting a is estimated to be the latency difference between its software implementation and CI multiplied by the total amount of times a is executed in the *current* worst-case path. After annotating the selection of CI a' with maximum profit to the CFG such that all calls to a' utilize the hardware-accelerated CI (instead of functionality-equivalent software), another WCET estimate needs to be performed. The worst-case path might have changed, because the execution time of the previous worst-case path was reduced. WCET estimation and CI selection are repeated alternately until the whole reconfigurable area of the target platform is occupied.

Performance Configuration

The performance configuration is obtained with a similar greedy algorithm. We utilize the same latency estimates ($\mathrm{latency}_{\mathrm{sw}}$ and $\mathrm{latency}_{\mathrm{hw}}$) that were obtained during WCET estimation. Instead of worst-case path information, however, profiling information is used to determine the profit of a CI, i.e., the application is run for a typical use case to determine the number of calls n_a to each CI a. Thus, the profit on the ACET is estimated as:

$$\mathrm{profit}_{\mathrm{ACET}}(a) = n_a \cdot (\mathrm{latency}_{\mathrm{sw}} - \mathrm{latency}_{\mathrm{hw}})$$

Again, CIs are greedily added to the performance configuration until the whole reconfigurable area is occupied. In contrast to the worst-case path, the average-case calls to accelerators do not change when additional CIs are selected. Therefore, only a single profiling run is required.

WCET Bounds

The final offline preparation step is to determine WCET bounds for certain parts and configurations of the task that are used to perform decisions during online optimization. The task's guaranteed WCET bound is determined based on the safe configuration. Additionally, WCET bounds are determined for single iterations of each kernel.

```
1 wrslck 0                                              (reset runtime slack accumulator)
2 for (...){                                                        (kernel header)
3   mobeg WCET^safe_iter                                (set counter for current iteration)
4   rdslck slack                                        (read accumulated runtime slack)
5   conditionally reconfigure based on slack, see Fig. 7.5
6   ...                                                          (original kernel body)
7   moend}                                  (add counter value to accumulated slack)
```

Listing 7.1: Operations that manage slack monitoring are added to the kernels that should be optimized at runtime

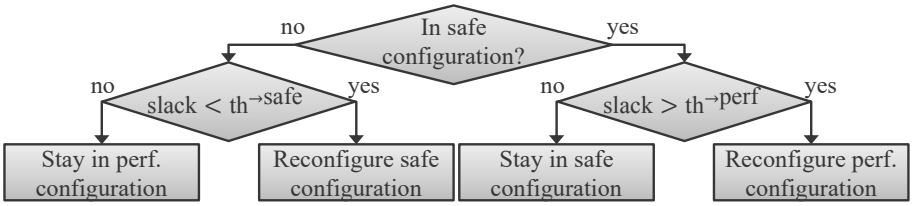


Figure 7.5: In each kernel iteration it is decided whether to reconfigure based on the current runtime slack and slack thresholds ($th^{\rightarrow perf}$ and $th^{\rightarrow safe}$)

As shown in Fig. 7.3, a single iteration starts with the kernel header and ends with a branch from the end of the kernel body back to the header. The bounds of an iteration are determined as $WCET^{safe}_{iter}$ and $WCET^{perf}_{iter}$ for the safe and performance configuration, respectively. This information is required for online slack monitoring and to decide when to switch the configuration as explained in the following section.

7.3.2 Online Optimization

Slack Monitoring

To enable online slack monitoring, we utilize a simple performance counter that counts CPU cycles similar to the cycle count register of the "Performance Monitoring Unit" in ARM Cortex-R cores used in the Xilinx Zynq UltraScale+ platform. Four operations are defined in the following that control counting and accumulation of CPU cycles: `mobeg`, `moend`, `rdslck` and `wrslck`. These operations are added to the kernels that should be optimized as shown in Listing 7.1. At the beginning of each iteration of a kernel, the counter is initialized with the kernel's per-iteration WCET bound $WCET^{safe}_{iter}$ using `mobeg`, which copies an unsigned integer value from a register argument into the counter. Then, during the kernel iteration, the counter is decremented in every cycle. At the end of the kernel iteration, `moend` stops the counter and adds the current value to the accumulator. Note that, because the counter was initialized with $WCET^{safe}_{iter}$, it is guaranteed to have a value ≥ 0 when the kernel is executed in the safe configuration. In the performance configuration, however, the counter could count down until $WCET^{safe}_{iter} - WCET^{perf}_{iter}$ (a negative number, because $WCET^{perf}_{iter} > WCET^{safe}_{iter}$). In this case, the accumulated slack is reduced. The currently accumulated slack is read using `rdslck`, which copies the accumulator value to a register argument. In case the accumulated slack is reduced below a certain threshold while in the performance configuration, the safe configuration needs to be configured again. How the thresholds are obtained and enforced using the conditional reconfiguration (Line 5) is described in the following.

Reconfiguration within WCET Bounds

While the stalling model (detailed in Chapter 5) enables reconfiguration only before entering a kernel, in the following conditional reconfiguration is introduced that decides whether to reconfigure or not in each iteration of the kernel based on the currently accumulated slack as shown in Fig. 7.5. As shown in Listing 7.1, conditional reconfiguration is executed before the kernel body in every kernel iteration (Line 5). It manages the state of the

reconfigurable area and triggers a reconfiguration in case the accumulated slack reached a certain threshold. To determine the minimum threshold of accumulated slack that enables to switch from safe to performance configuration (while maintaining WCET guarantees), the worst-case execution –immediately after reconfiguration of the performance configuration was triggered– needs to be considered. First, it takes $\text{rdelay}^{\text{perf}}$ cycles to configure the performance configuration. Then, the kernel iteration takes a maximum of $\text{WCET}^{\text{perf}}_{\text{iter}}$ cycles, which means that the accumulated slack is reduced by $|\text{WCET}^{\text{safe}}_{\text{iter}} - \text{WCET}^{\text{perf}}_{\text{iter}}|$. Finally, when the conditional reconfiguration block is executed again and we need to switch back to the safe configuration, it takes $\text{rdelay}^{\text{safe}}$ to perform the reconfiguration. In summary, the minimum accumulated slack to be able to switch from safe to performance configuration and remain within the WCET guarantee in the worst case is:

$$\text{th}^{\rightarrow\text{perf}} := \text{rdelay}^{\text{perf}} + |\text{WCET}^{\text{safe}}_{\text{iter}} - \text{WCET}^{\text{perf}}_{\text{iter}}| + \text{rdelay}^{\text{safe}} \tag{7.1}$$

As mentioned before, a kernel iteration reduces the accumulated slack by $|\text{WCET}^{\text{safe}}_{\text{iter}} - \text{WCET}^{\text{perf}}_{\text{iter}}|$ in the worst case and configuring the safe configuration takes $\text{rdelay}^{\text{safe}}$. Thus, to safely switch back from performance to safe configuration, the reconfiguration needs to be triggered once the accumulated slack is lower than:

$$\text{th}^{\rightarrow\text{safe}} := |\text{WCET}^{\text{safe}}_{\text{iter}} - \text{WCET}^{\text{perf}}_{\text{iter}}| + \text{rdelay}^{\text{safe}} \tag{7.2}$$

For any value $> \text{th}^{\rightarrow\text{safe}}$, there is still enough accumulated slack to safely execute one iteration of the kernel (even in worst case) and switch back to the safe configuration afterwards.
During the offline preparation phase (explained Section 7.3.1), the slack is annotated as constant 0 such that only the initial reconfiguration to the safe configuration is accounted for during WCET analysis (just like without applying the approach of this chapter).

7.4 Experimental Evaluation

The presented approach is suitable for any application that can benefit from hardware accelerators in different execution paths. It is evaluated on the runtime-reconfigurable processor *i*-Core that was introduced in Section 2.4 in the following. For this work, a simple performance counter (cycle counter plus accumulator) was added that implements the operations for slack monitoring explained in Section 7.3.2. The offline preparation algorithms were implemented (Section 7.3.1) by running Absint aiT [2] in batch mode to obtain WCET estimates and iteratively generate constraints in aiT's AIS2 constraint language to model selected CIs. As aiT is closed-source software, we could not directly integrate support for reconfigurable CIs. Instead, every call to a CI in the binary was substituted by an ADD opcode and a constraint that sets the delay for the new ADD instruction to the delay of the specific CI during WCET estimation (see Section 5.6 for details). Guaranteed reconfiguration delays are obtained using CoRQ (see Chapter 4) and are annotated using additional AIS2 constraints.
The approach is evaluated with the same H.264 encoder application used in Chapters 5 and 6 that uses 9 CIs, which cover the most compute-intensive kernels shown in Table 7.1. Each kernel reconfigures the full reconfigurable area (modeled to be of similar size like in the Xilinx Zynq XC7Z010 platform). The H.264 encoder covers most of the properties tested in the TACLeBench[2] WCET Benchmark, e.g., the H.264 decoder –that is part of TACLeBench– performs a subset of the computations performed in the H.264 encoder that we evaluate. We compiled the application using BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1[3] for a frame size of 396 macroblocks (i.e., CIF resolution). Note that higher resolutions would benefit our approach by reducing the relative overhead of reconfiguration delays. Measured execution times are obtained using our cycle-accurate SystemC-based simulator of the

[1] C Lines of Code that are replaced by utilizing a hardware CI (without comments or whitespace)
[2] `http://www.tacle.eu/index.php/activities/taclebench`
[3] At higher optimization levels, GCC emitted so-called *irreducible loops* that increase the WCET estimate compared to O1.

Table 7.1: CIs used in the H.264 Application

`Accelerated func.` and Description	MB type	Working Set	CLoC[1]
`MotionEstimation` Kernel			
`SATD`: Sum of Abs. Transf. Differences	P and I	16×16 px	123
`SAD`: Sum of Abs. Differences	P and I	16×16 px	24
`EncodeMacroBlock` Kernel			
`MC_Hz`: Motion Compens. Interpol. Horiz.	P	4 px	51
`IPred_HDC`: Intra Prediction Horiz.	I	16×16 px	35
`IPred_VDC`: Intra Prediction Vert.	I	16×16 px	19
`DCT`: Discrete Cosine Transf.	P and I	4×4 px	76
`HT2x2`: Hadamard Transform	P and I	2×2 px	12
`HT4x4`: Hadamard Transform	I	4×4 px	111
`LoopFilter` Kernel			
`LoopFilter`: In-Loop Deblock. Filter	P and I	4 px	82

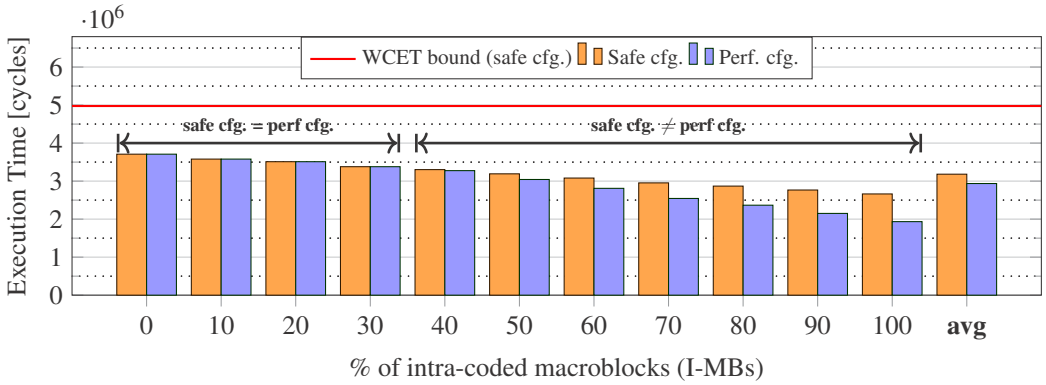


Figure 7.6: Execution time of `EncodeMacroBlock` for safe configuration and different performance configurations obtained for different execution profiles

reconfigurable processor. Before performing the evaluation, we calibrated aiT and our simulator by harmonizing hardware parameters and verifying the results of test-cases, e.g., load-store sequences. Hardware parameters like reconfigurable area constraints, reconfiguration bandwidth and partial bitstream sizes of CI implementations were obtained from our Xilinx Virtex-7-based hardware prototype. The CPU is modeled to be clocked at 400 MHz (at which the LEON3 operates when implemented as an ASIC), the reconfigurable area is clocked at 100 MHz (frequency of accelerators used by CIs when implemented on Virtex-7).

7.4.1 Results

Offline Preparation

The H.264 encoder has two main execution paths with different CI requirements, depending on whether a macroblock (MB) is encoded either by referencing an MB from a previous frame (P-MB) or using the current frame only (I-MB). Table 7.1 shows which CIs are used in the I-MB or P-MB path. First, we will focus on the `EncodeMacroBlock` kernel that performs the actual encoding. It is the most complex kernel and provides an opportunity to create distinct safe and performance configurations. Which of the total 396 MBs is encoded as either I-MBs or P-MBs at runtime is input-dependent: hectic video scenes have a high ratio of I-MBs (up to 100%, e.g., video from a camera pointing sideways out of a moving car), steady scenes have a low ratio of I-MBs. Therefore, the performance configuration differs for different execution profiles (synthesized input data that is encoded as I-MBs and P-MBs randomly distributed in the desired ratio). During WCET optimization, however, the current worst-case path encodes all MBs either as P-MBs or I-MBs (the worst-case path changes while preparing the safe

configuration, see Fig. 7.4). The final safe configuration (see Section 7.3.1 and Chapter 6) selects `MC_Hz`, `DCT`, `HT2x2` and `HT4x4`. In this configuration, the worst-case path does not encode any I-MBs but only P-MBs.
Figure 7.6 shows execution time results for different execution profiles (and different resulting performance configurations) of the `EncodeMacroBlock` kernel. The performance configuration differs from the safe configuration when 40% or more of the total MBs in a frame are I-MBs. In these cases, `IPred_HDC` and `IPred_VDC` are selected instead of `MC_Hz` and a performance increase of up to 27.4% is achieved at $x = 100$ (7.7% on average, without reconfiguration delay). Unaccelerated execution takes between $20.1 \cdot 10^6$ ($x = 0$) and $19.4 \cdot 10^6$ ($x = 100$) cycles, i.e., the measured speedup of the safe configuration is between $7.1\times$ and $5.4\times$.

Online Optimization

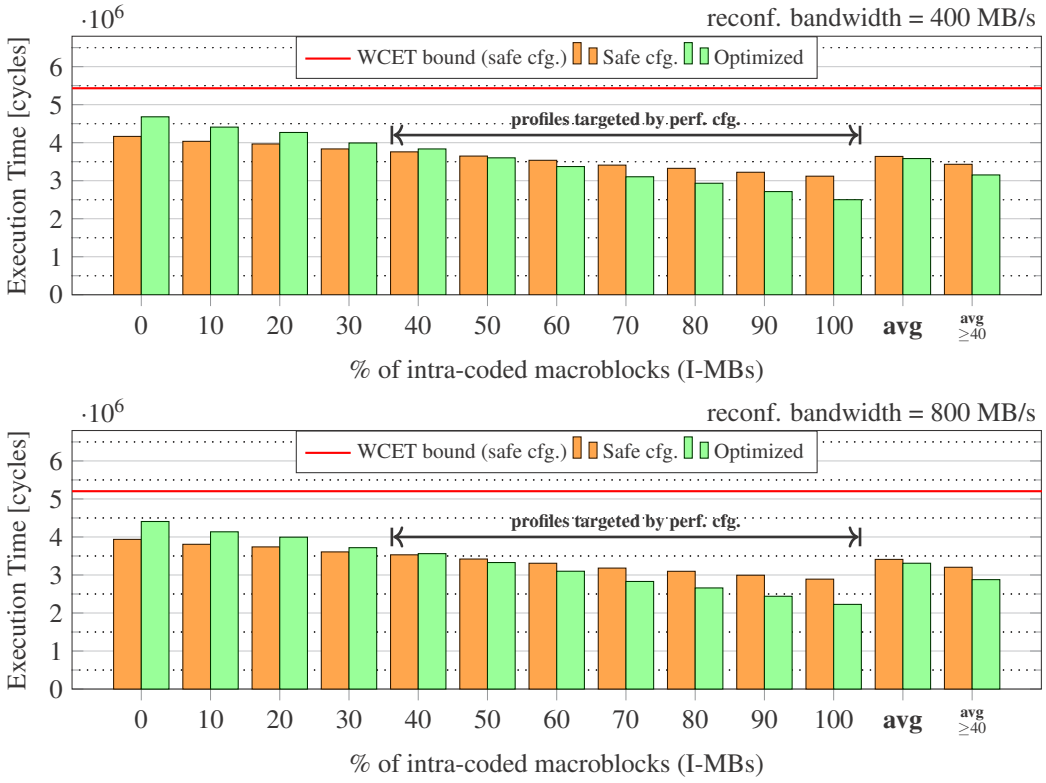


Figure 7.7: Execution time of `EncodeMacroBlock` for safe configuration and online optimization for different execution profiles and reconfiguration bandwidths (400 MB/s (top), 800 MB/s (bottom))

As shown in Fig. 7.6, the presented optimization approach is suitable to the H.264 encoder when video frames with $\geq 40\%$ I-MBs can be expected (i.e., moderate movement). We proceed using the performance configuration obtained in this case (`IPred_HDC`, `IPred_VDC`, `DCT`, `HT2x2`, `HT4x4`) in the following for all execution profiles. In this case, $|\text{WCET}_{\text{iter}}^{\text{safe}} - \text{WCET}_{\text{iter}}^{\text{perf}}| = |12630 - 16374| = 3744$, i.e., an iteration encoding P-MBs takes 3744 cycles longer for the performance configuration compared to safe configuration in the worst case. Figure 7.7 shows execution time results of the `EncodeMacroBlock` kernel when our approach is applied for different reconfiguration bandwidths and I-MB ratios. A reconfiguration bandwidth of 400 MB/s (Fig. 7.6 (top)) is supported by our Xilinx Virtex-7-based hardware prototype, while recent Xilinx FPGAs ("UltraScale+") support a reconfiguration bandwidth of 800 MB/s[4] (Fig. 7.6 (bottom)). In both cases, our approach is beneficial for frames that contain $\geq 50\%$ of I-MBs. Even when the performance configuration results in a lower execution time at 40% I-MBs (see Fig. 7.6), the speedup is voided by the additional reconfiguration delay (of switching from safe to performance

[4] See: "Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics" (DS923)

configuration after accumulating sufficient slack). The maximum execution time reduction is 19.8% and 23.0% for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. For frames that contain $< 50\%$ of I-MBs, the execution time can be slowed down (as expected) by up to 11% ($x = 0$ for both reconf. bandwidths). When only considering the execution profiles that the performance configuration was obtained for ($\geq 40\%$ of I-MBs), the average execution time reduction ($\overset{\text{avg}}{\geq 40}$) is 8.1% and 10.2% (total average 1.5% and 3.0%) for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. The statically guaranteed WCET bounds were hit in none of our experiments.
Finally, ca. 45% of the execution time of the H.264 encoder are spent in the `EncodeMacroBlock` kernel (between 38.2% and 51.7% in our measurements). Thus, our technique (when applied to the `EncodeMacroBlock` kernel) approximately reduces the total execution time on average (including atypical execution profiles like still images) by 0.7% and 1.4% for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. On average in the targeted execution profiles ($\overset{\text{avg}}{\geq 40}$), the total execution time is reduced by 3.6% and 4.6%; as well as up to 8.9% and 10.4% (again, for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively). Note that the execution time reductions were achieved by only adding slack monitoring using performance counters as described in Section 7.3.2 to the runtime-reconfigurable processor *i*-Core.

7.5 Conclusion

This chapter presented the first step towards optimized static WCET guarantees *and* runtime optimization of the ACET using runtime reconfiguration of hardware accelerators. It might very well enable the use of runtime reconfiguration for performance in safety-critical systems –where nowadays only static configurations are used– because it enables to utilize runtime slack for reconfiguration accelerators that benefit average-case execution while maintaining WCET guarantees. Execution time reductions by up to 8.9% and 10.4% (on average 3.6% and 4.6% for targeted execution profiles) at a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively, were demonstrated for the H.264 encoder. To achieve these benefits, only slack monitoring using a performance counter needed to be added to a runtime-reconfigurable processor. In future work, more sophisticated runtime slack prediction (instead of static thresholds) could benefit execution profiles that were not targeted by the "performance configuration", e.g., to stay in the time-wise safe configuration when the worst-case path is frequently executed even though sufficient slack was accumulated.
The following chapter concludes the contributions of this thesis.

8 Thesis Conclusion

Real-time systems have a rapidly increasing demand for performance that cannot be provided by high-performance architectures, which were designed for average-case performance. First, this thesis introduced novel co-scheduling approaches to distribute work among CPU and GPU in an extensive analysis of how (average-case) performance is achieved on *fused CPU-GPU architectures*, a main trend in current high-performance microarchitectures that combines a CPU and a GPU on a single chip. Being able to employ such architectures in real-time systems would be highly desirable, because they provide high performance within a limited area and power budget. During the analysis, however, a cache coherency bottleneck was uncovered that (i) complicated performance predictions and (ii) added a shared last level cache between CPU and GPU to the growing list of microarchitectural features that can benefit average-case performance, but render the analysis of WCET guarantees on high-performance architectures virtually infeasible. This motivated the need for novel microarchitectural features that provide *predictable* performance that are amenable to timing analysis. Thus, the main focus of this thesis was to establish worst-case execution time guarantees for runtime-reconfigurable systems as a novel means to achieve predictable performance. Towards this end, first a runtime reconfiguration controller called "Command-based Reconfiguration Queue" (CoRQ) was presented that provides guaranteed latencies for its operations and enables timing analysis of runtime-reconfigurable architectures for WCET guarantees. Based on the –now feasible– guaranteed reconfiguration delay of accelerators, a WCET analysis was introduced that enables tasks to reconfigure application-specific custom instructions (CIs, which invoke execution of one or more accelerators) during runtime. Different measures to deal with reconfiguration delays were compared as well as the *timing anomaly* of runtime reconfiguration identified and safely bounded: a case where executing iterations of a computational kernel faster than in WCET can prolong the total execution time of a task. Once tasks that perform runtime reconfiguration of CIs could be analyzed for WCET guarantees, the question of *which* CIs to configure on a constrained reconfigurable area to optimize the WCET was raised, when multiple CIs with different implementations each (allowing to trade-off latency and area requirements) can be selected. This so-called *WCET-optimizing instruction set selection problem* was modeled based on the Implicit Path Enumeration Technique. To our knowledge, this is the first approach that enables WCET optimization with support for global program flow information (and reconfiguration delay). An optimal algorithm (similar to Branch and Bound) and a fast greedy heuristic algorithm (that achieves the optimal solution in most cases) were presented. Finally, an approach was presented that for the first time combines optimized static WCET guarantees *and* runtime optimization of the average-case execution (maintaining WCET guarantees) using runtime reconfiguration of hardware accelerators. It comprised an analysis of runtime slack bounds that enable safe reconfiguration for average-case performance under WCET guarantees and presented a mechanism to monitor runtime slack using a simple performance counter that is commonly available in many microprocessors.
Ultimately, runtime reconfiguration of accelerators was shown as a key feature to achieve predictable performance.

8.1 Future Work

This thesis opens several directions for future research. Two main directions are highlighted in the following.

8.1.1 WCET Guarantees and Mixed-Criticality for Loosely-Coupled Reconfigurable Architectures

The WCET analysis and optimization of Chapters 5 to 7 focus on reconfigurable processors, i.e., architectures where the reconfigurable fabric is integrated into the pipeline of a processors (a so-called *tightly-coupled architecture*). A tight coupling between processor and FPGA reduces communication latencies and enables readily-configured accelerators to be analyzed just like existing multi-cycle instructions during WCET analysis (see Chapter 5). However, the design of a tightly-coupled architecture requires considerable effort to integrate processor pipeline and reconfigurable fabric. Thus, many commercially-available architectures are *loosely-coupled*, i.e., processor(s) and reconfigurable fabric are separate processing devices on the same chip that are connected via a common system bus. Commercial loosely-coupled architectures are, e.g., the Xilinx Zynq or Intel (formerly Altera) SoC FPGA platforms. To achieve WCET guarantees on such architectures, communication protocols between processor and reconfigurable accelerators as well as worst-case analyses need to be designed that consider the common system bus, but still allow execution time guarantees. This would enable application of the approaches presented in Chapters 5 to 7 to loosely-couples architectures and provide the basis for further research in the direction of *mixed-criticality* [19], where only a subset of tasks needs to fulfill execution time guarantees while the other tasks are executed with *best effort*. E.g., the Xilinx Zynq UltraScale+ couples an ARM Cortex-A53 high-performance processor and an ARM Cortex-R5 real-time processor to a reconfigurable fabric. When only real-time tasks executing on the Cortex-R5 utilize reconfigurable accelerators, it can be expected that –as a result of requiring schedulability guarantees– the overall utilization of the reconfigurable fabric will be quite low, as recent works on real-time scheduling on a loosely-coupled system have shown [16]. Allowing best-effort tasks that execute on the Cortex-A53 to also utilize reconfigurable accelerators on the shared fabric can help to raise its utilization (resulting in less wasted resources of the whole system), while guarantees are maintained for real-time tasks[1].

8.1.2 Probabilistic WCET Guarantees

This thesis focused on analysis of *deterministic* WCET guarantees, i.e., through static analysis an execution time bound is obtained that is guaranteed to never be exceeded. As motivated in Chapters 1 and 3, the problem with this approach is that current high-performance architectures can virtually not be analyzed, because they introduce average-case performance enhancing features that lead to an explosion of possible microarchitectural states. More precisely, for many of these features, e.g., out-of-order execution [63, 90], the effects on the execution time are actually understood in principle, but modeling them analytically is only possible with simplifications that introduce so much pessimism that the results cannot be used in practice [14]. Therefore, an emerging trend in real-time systems research is the analysis of *probabilistic* WCET (pWCET) guarantees. Instead of obtaining an absolute WCET guarantee, the aim of pWCET analysis is to obtain a probability density function that, for a given execution time, determines the probability that the execution time is exceeded. When this probability is sufficiently low (often lower than 10^{-15} is targeted), the execution time bound that achieves this probability is considered safe, because, e.g., the probability of mechanical failures in the real-time system is several magnitudes higher. pWCET analysis exists in static and measurement-based variants. Measurement-based pWCET analysis can derive execution time guarantees from measured execution times of a task (“end-to-end” measurements), when the measurements fulfill certain statistical properties (e.g., they need to be independent and identically distributed (*i.i.d.*)) that allow the application of Extreme Value Theory [41]. Extreme Value Theory can derive probability density functions that precisely model the extremes from statistical data (like maximum execution time from execution time measurements). This makes measurement-based pWCET an especially promising approach, because the real-time system can be treated as a *gray box* (only partial knowledge of its behavior is required) such that it can potentially overcome the

[1] Early results of our research in this direction can be found in Rapp, Martin. “A Mixed Criticality Architecture with Reconfigurable Accelerators”, *Master Thesis*, 2016.

limitation of static WCET analysis, which requires the whole system behavior to be modeled in detail. However, obtaining i.i.d. measurements that allow measurement-based pWCET requires some form of randomization and it is still unclear how pWCET results can be safely used in schedulability analyses [33]. Furthermore, in systems that are analyzable for deterministic WCET guarantees, pWCET analysis does not necessarily produce better results [1]. In our future research, we will investigate the applicability of pWCET analysis on fused CPU-GPU architectures (see Chapter 3) as well as reconfigurable architectures (basing on approaches presented in the previous chapters). Our aim is to enable pWCET guarantees for systems that can so far not be analyzed for execution time guarantees at all as well as to evaluate what kind of architectures are more suitable to either deterministic WCET or pWCET guarantees.

A Appendix

A.1 Demonstration Prototypes

In the context of this thesis, mainly within the DFG-funded project Invasic Computing (Section 2.5.1), demonstration setups were created that showed the practicality of the presented research.

A.1.1 Concurrent Reconfigurable Fabric Utilization

Figure A.1 shows a demonstrator, which bases on previous *i*-Core prototypes to implement an extension to the *i*-Core concept (see Section 2.4) called *Concurrent Reconfigurable Fabric Utilization* (COREFAB) [45] on a Xilinx Virtex-7 VC707 evaluation board. COREFAB allows general-purpose processors (GPPs) within the same system on chip to utilize reconfigurable fabric of *i*-Core to execute CIs. In the following, CIs issued by the *i*-Core are named 'primary CIs' and those from the GPPs are named 'remote CIs'. From the application developer's view, primary CIs and remote CIs appear identical, but the latency to issue a remote CI is higher and primary CIs are preferred over remote CIs. Primary and remote CIs are analyzed in hardware during execution. As long as no resource conflict appears, both CIs execute in parallel using accelerators that are configured on the reconfigurable fabric. In case a conflict appears, the remote CI is stalled and the primary CI proceeds. Thus, CI execution on *i*-Core is not impaired by remote CIs, which is an opportunity for future work to investigate mixed-criticality execution on the *i*-Core's reconfigurable fabric (guaranteeing WCET for *i*-Core only, but achieving a high utilization of accelerators using remote CIs).
The demonstrator shows the H.264 encoder that was evaluated in the previous chapters, running on a GPP that benefits from acceleration via remote CIs using COREFAB. The last encoded frame and an on-screen display showing status information are output to a screen using HDMI. *i*-Core provokes conflicts in this setup by running a loop that only executes primary CIs (`SAD` and `DCT`, see Table 7.1). This demonstrator shows that despite these conflicts, the H.264 encoder executes 2× to 3× faster (encoded frames per second) on the GPP using COREFAB than without.

A.1.2 Accelerating a Finite Volume Tsunami Model using Reconfigurable Hardware in Invasive Computing

Within the Invasive Computing project, pipelined floating-point accelerators were created for *i*-Core, to enable efficient implementation of floating-point CIs [9]. We designed floating-point CIs to accelerate the Shallow Water Equations application in X10 (SWE-X10) [80]. SWE-X10 is a proxy application for the computation of shallow water waves, it implements a model that can be used to predict the propagation of a tsunami wave [79].
In a collaboration between virtually all subprojects of Invasive Computing, the *i*-Core-accelerated SWE-X10 application was leveraged to create a demonstration setup of the full Invasive Computing stack as shown in Figs. A.2 and A.3. For this demonstrator, the *InvasIC* manycore architecture [50] was implemented on a proDesign proFPGA[1] consisting of four Xilinx XC7V2000T FPGAs. In total, the prototype consists of 80 processor cores (16 *tiles* that were connected via a network on chip and contain 5 cores each). Four of these cores are *i*-Cores. On top of the InvasIC architecture, the parallel operating system OctoPOS [74] provides hardware abstractions and resource management, following the resource-aware invasive programming model (see Section 2.5.1). Invasive X10 (X10i)

[1] `https://www.profpga.com`

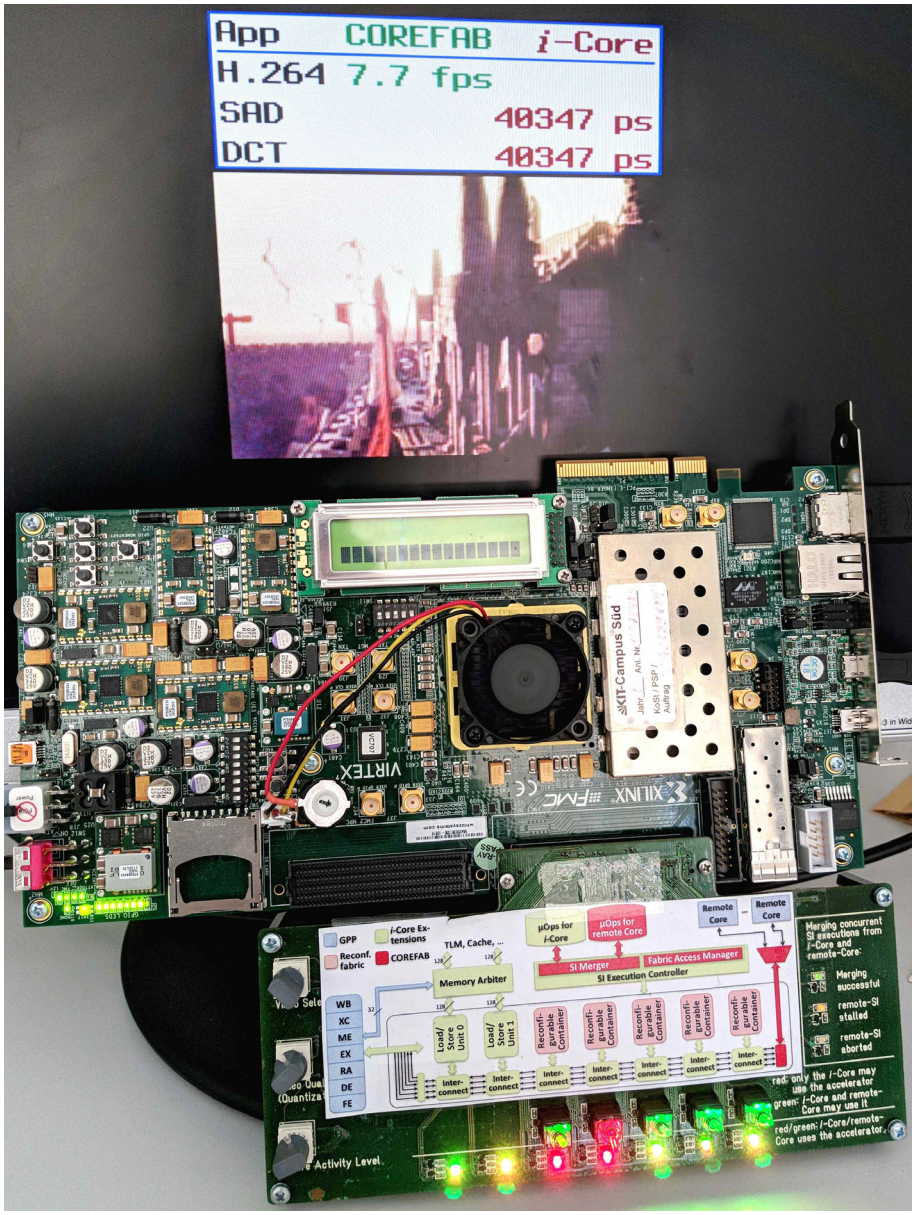


Figure A.1: Prototype demonstrating that accelerators can be efficiently shared between different cores within the same system on chip

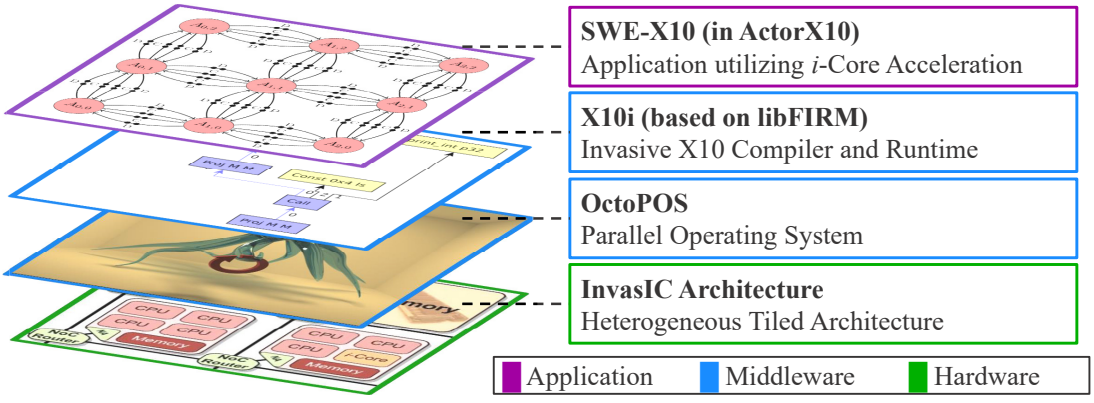


Figure A.2: High-level overview of the Invasive Computing stack

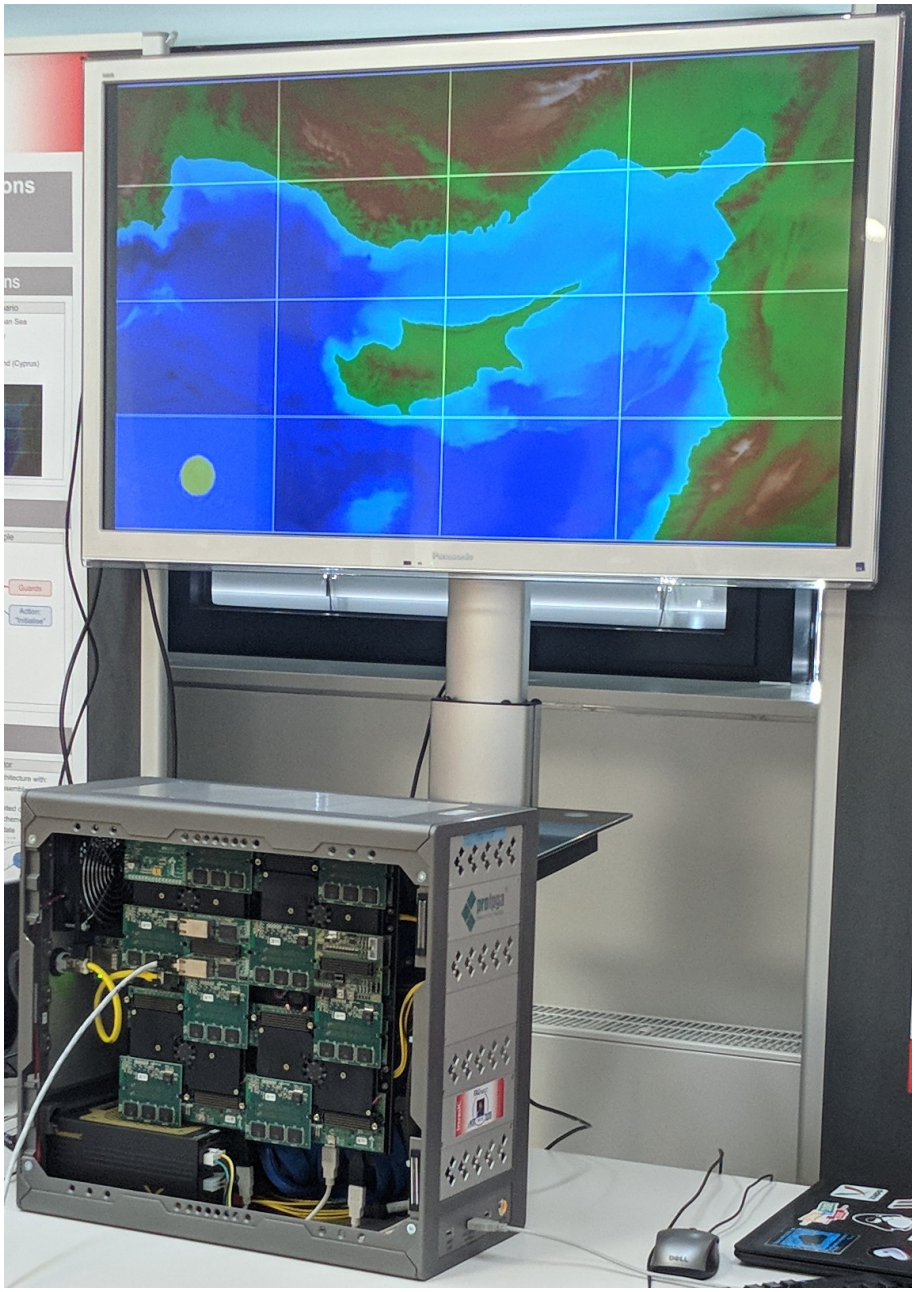


Figure A.3: Demonstrator that executes *i*-Core-accelerated SWE-X10 on the full Invasive Computing stack

[18] is a compiler and runtime system that enables high-level programming of invasive applications based on the programming language X10 [21]. Finally, the SWE-X10 application is executed on top, demonstrating the full invasive stack. For demonstration purposes, a live visualization was implemented. It is executed on a separate computer and receives live simulation results from the InvasIC prototype over Ethernet. Figure A.3 shows the demonstrated scenario, where a tsunami wave propagates from the south-west of Cyprus.

List of Figures

List of Tables

Bibliography

[1] Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J Cazorla. “On the comparison of deterministic and probabilistic WCET estimation techniques”. In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE. 2014, pp. 266–275.

[2] AbsInt. *aiT Worst-Case Execution Time Analyzers*. Website: `http://www.absint.com/ait/`. [Online; accessed 31-Aug-2018]. 2018.

[3] Andreas Agne, Markus Happe, Andreas Keller, Enno Lubbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. “ReconOS: An operating system approach for reconfigurable computing”. In: *IEEE Micro* 34.1 (2014), pp. 60–71.

[4] Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, and Christine Rochange. “WCET and Mixed-Criticality: What does Confidence in WCET Estimations Depend Upon?” In: *OASIcs-OpenAccess Series in Informatics*. Vol. 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[5] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank Mueller. “Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems”. In: *SIGARCH Comput. Archit. News* 31.2 (May 2003), pp. 350–361. ISSN: 0163-5964. DOI: 10.1145/871656.859659.

[6] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. “Building timing predictable embedded systems”. In: *ACM Trans. on Embed. Comput. Syst.* 13.4 (2014), 82:1–82:37.

[7] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. “OTAWA: An open toolbox for adaptive WCET analysis”. In: *SEUS*. Springer, 2010, pp. 35–46.

[8] Rajkishore Barik, Naila Farooqui, Brian T Lewis, Chunling Hu, and Tatiana Shpeisman. “A black-box approach to energy-aware scheduling on integrated CPU-GPU systems”. In: *IEEE/ACM Int. Symp. on Code Gen. and Opt.* IEEE. 2016, pp. 70–81.

[9] Lars Bauer, Artjom Grudnitsky, Marvin Damschen, Srinivas Rao Kerekare, and Jörg Henkel. “Floating point acceleration for stream processing applications in dynamically reconfigurable processors”. In: *IEEE Symp. on Embed. Syst. For Real-time Multimedia (ESTIMedia), Amsterdam, The Netherlands, October 8-9, 2015*. 2015, pp. 1–2. DOI: 10.1109/ESTIMedia.2015.7351762.

[10] Lars Bauer and Jörg Henkel. *Run-time adaptation for reconfigurable embedded processors*. Springer Science & Business Media, 2010.

[11] Lars Bauer, Muhammad Shafique, and Jörg Henkel. “A computation-and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor”. In: *Int. Conf. on Field Programmable Logic and Applications*. IEEE. 2008, pp. 203–208.

[12] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. “RISPP: rotating instruction set processing platform”. In: *Proc. of Design Automat. Conf.* ACM. 2007, pp. 791–796.

[13] Khaled Benkrid, Ying Liu, and AbdSamad Benkrid. “A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.4 (2009), pp. 561–570.

[14] Guillem Bernat, Antoine Colin, and Stefan M Petters. “WCET analysis of probabilistic hard real-time systems”. In: *IEEE Real-Time Syst. Symp.* IEEE. 2002, pp. 279–288.

[15] Margrit Betke, Esin Haritaoglu, and Larry S Davis. “Real-time multiple vehicle detection and tracking from a moving vehicle”. In: *Machine vision and applications* 12.2 (2000), pp. 69–83.

[16] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. “A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs”. In: *Proc. of Real-Time Syst. Symp.* IEEE. 2016, pp. 1–12.

[17] Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. “FFX: A portable WCET annotation language”. In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM. 2012, pp. 91–100.

[18] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *Dynamic X10: Resource-Aware Programming for Higher Efficiency*. Tech. rep. 8. X10 ’14. Karlsruhe Institute of Technology, 2014.

[19] Alan Burns and Rob Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013).

[20] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Acm Sigplan Notices*. Vol. 40. 10. ACM. 2005, pp. 519–538.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. “Rodinia: A benchmark suite for heterogeneous computing”. In: *IEEE Int. Symp. on Workl. Charact.* Ieee. 2009, pp. 44–54.

[23] Juan Antonio Clemente, Javier Resano, and Daniel Mozos. “An Approach to Manage Reconfigurations and Reduce Area Cost in Hard Real-time Reconfigurable Systems”. In: *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014), 90:1–90:24. ISSN: 1539-9087. DOI: 10.1145/2560037.

[24] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. “Composable Accelerator-rich Microprocessor Enhanced for Adaptivity and Longevity”. In: *Proc. of the Int. Symp. on Low Power Electronics and Design*. ISLPED ’13. Beijing, China: IEEE Press, 2013, pp. 305–310. ISBN: 978-1-4799-1235-3.

[25] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proc. of the Symp. on Principles of programming languages*. ACM. 1977, pp. 238–252.

[26] Michael Dales. “Managing a reconfigurable processor in a general purpose workstation environment”. In: *Proc. of the Conf. on Design, Automation and Test in Europe*. IEEE Computer Society. Mar. 2003, p. 10980.

[27] Marvin Damschen, Lars Bauer, and Jörg Henkel. “Extending the WCET Problem to Optimize for Runtime-Reconfigurable Processors”. In: *ACM Trans. on Archit. and Code Optim. (TACO)* 13.4 (2016), 45:1–45:24. DOI: 10.1145/3014059.

[28] Marvin Damschen, Lars Bauer, and Jörg Henkel. “CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems”. In: *IEEE Embedded Systems Letters (ESL)* 9.3 (2017), pp. 77–80. DOI: 10.1109/LES.2017.2714844.

[29] Marvin Damschen, Lars Bauer, and Jörg Henkel. “Timing Analysis of Tasks on Runtime Reconfigurable Processors”. In: *IEEE Trans. on Very Large Scale Integration Syst. (TVLSI)* 25.1 (2017), pp. 294–307. DOI: 10.1109/TVLSI.2016.2572304.

[30] Marvin Damschen, Frank Mueller, and Jörg Henkel. “Co-Scheduling on Fused CPU-GPU Architectures with Shared Last Level Caches”. In: *IEEE Trans. on Comput.-Aided Design of Integrated Circuits and Syst. (TCAD)* (2018). ESWEEK Special Issue, to appear. DOI: 10.1109/TCAD.2018.2857042.

[31] Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel. “*i*-Core: A runtime-reconfigurable processor platform for cyber-physical systems”. In: *Embedded, Cyber-Physical, and IoT Systems: Smart Cameras, Hardware/Software Co-Design, and Multimedia — Essays Dedicated to Marilyn Wolf on the Occasion of Her 60th Birthday*. Ed. by S. S. Bhattacharyya, M. Potkonjak, and S. Velipasalar. to appear. Springer International Publishing, 2019.

[32] Abhishek Das, David Nguyen, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. “An FPGA-based network intrusion detection architecture”. In: *IEEE Transactions on Information Forensics and Security* 3.1 (2008), pp. 118–132.

[33] Robert I Davis, Alan Burns, and David Griffin. “On the Meaning of pWCET Distributions and their use in Schedulability Analysis”. In: *Proceedings 2017 Real-Time Scheduling Open Problems Seminar (RTSOPS)*. 2017.

[34] Christian De Schryver, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. “An energy efficient FPGA accelerator for monte carlo option pricing with the heston model”. In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE. 2011, pp. 468–474.

[35] Christopher Dennl, Daniel Ziener, and Jurgen Teich. “On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library”. In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 45–52.

[36] Florian Dittmann and Stefan Frank. “Hard real-time reconfiguration port scheduling”. In: *Proc. of the Conf. on Design, Automation and Test in Europe*. IEEE. 2007, pp. 123–128.

[37] Stephen A Edwards and Edward A Lee. “The case for the precision timed (PRET) machine”. In: *Proc. of Design Automat. Conf.* ACM. 2007, pp. 264–265.

[38] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. “Clustered worst-case execution-time calculation”. In: *IEEE Trans. on Computers* 54.9 (2005), pp. 1104–1122.

[39] Heiko Falk and Jan C Kleinsorge. “Optimal static WCET-aware scratchpad allocation of program code”. In: *Proc. of Design Automat. Conf.* ACM. 2009, pp. 732–737.

[40] Heiko Falk, Sascha Plazar, and Henrik Theiling. “Compile-time decided instruction cache locking using worst-case execution paths”. In: *Proc. of Int. Conf. on Hardware/software codesign and syst. synthesis*. ACM. 2007, pp. 143–148.

[41] Willliam Feller. *An introduction to probability theory and its applications*. Vol. 2. John Wiley & Sons, 2008.

[42] Carlo Galuzzi and Koen Bertels. “The instruction-set extension problem: A survey”. In: *ACM Trans. on Reconfig. Technol. and Syst.* 4.2 (2011), p. 18.

[43] Robert S Garfinkel, George L Nemhauser, et al. *Integer programming*. Vol. 4. Wiley New York, 1972.

[44] Khronos OpenCL Working Group et al. “The OpenCL specification version 2.0”. In: *https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf* (2015).

[45] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. “COREFAB: concurrent reconfigurable fabric utilization in heterogeneous multi-core systems”. In: *Proc. of Int. Conf. on Compilers, Architecture and Synthesis for Embed. Syst.* ACM. 2014, p. 5.

[46] Tanja Harbaum, Christoph Schade, Marvin Damschen, Carsten Tradowsky, Lars Bauer, Jörg Henkel, and Jürgen Becker. “Auto-SI: An adaptive reconfigurable processor with run-time loop detection and acceleration”. In: *IEEE Intl. System-on-Chip Conf., (SOCC), Munich, Germany, September 5-8, 2017*. 2017, pp. 153–158. DOI: 10.1109/SOCC.2017.8226027.

[47] Scott Hauck, Thomas W Fry, Matthew M Hosler, and Jeffrey P Kao. “The Chimaera reconfigurable functional unit”. In: *IEEE Trans. on Very Large Scale Integration Syst.* 12.2 (2004), pp. 206–217.

[48] John R Hauser and John Wawrzynek. “Garp: A MIPS processor with a reconfigurable coprocessor”. In: *Proc. of Symp. on Field-Programm. Custom Comput. Machines*. IEEE. 1997, pp. 12–21.

[49] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, et al. “Design and architectures for dependable embedded systems”. In: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2011, pp. 69–78.

[50] Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, et al. “Invasive Manycore Architectures”. In: *17th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2012, pp. 193–200.

[51] Silvia Heubach and Toufik Mansour. “Compositions of n with parts in a set”. In: *Congressus Numerantium* 168 (2004), p. 127.

[52] Trang Hoang et al. “An efficient FPGA implementation of the Advanced Encryption Standard algorithm”. In: *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on*. IEEE. 2012, pp. 1–4.

[53] Dominik Honegger, Helen Oleynikova, and Marc Pollefeys. “Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU”. In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE. 2014, pp. 4930–4935.

[54] Huynh Phung Huynh and Tulika Mitra. “Runtime reconfiguration of custom instructions for real-time embedded systems”. In: *Proc. of the Conf. on Design, Automation and Test in Europe*. IEEE. 2009, pp. 1536–1541.

[55] Robert Ioffe, Sonal Sharma, and Michael Stoner. “Achieving performance with OpenCL 2.0 on Intel®processor graphics”. In: *Proc. of Int. Works. on OpenCL*. ACM. 2015, p. 3.

[56] Stephen Junkins. “The Compute Architecture of Intel® Processor Graphics Gen9”. In: *Intel whitepaper v1* (2015).

[57] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann, 2015.

[58] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T Lewis, Chunling Hu, and Keshav Pingali. “Adaptive heterogeneous scheduling for integrated GPUs”. In: *Proc. of the Int. Conf. on Par. Arch. and Comp.* ACM. 2014, pp. 151–162.

[59] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. “Beyond loop bounds: comparing annotation languages for worst-case execution time analysis”. In: *Software & Systems Modeling* 10.3 (2011), pp. 411–437.

[60] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for Software Programmers*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319264060, 9783319264066.

[61] Chris Lattner. “LLVM and Clang: Advancing Compiler Technology”. In: *Proc. of FOSDEM* (2011).

[62] Young Choon Lee and Albert Y Zomaya. “On Effective Slack Reclamation in Task Scheduling for Energy Reduction.” In: *JIPS* 5.4 (2009), pp. 175–186.

[63] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. “Modeling out-of-order processors for WCET analysis”. In: *Real-Time Systems* 34.3 (2006), pp. 195–227.

[64] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. “Efficient microarchitecture modeling and path analysis for real-time software”. In: *Proc. of Real-Time Syst. Symp.* IEEE. 1995, pp. 298–307.

[65] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. “Hybrid CPU-GPU scheduling and execution of tree traversals”. In: *Proc. of the Int. Conf. on Supercomp.* ACM. 2016, p. 2.

[66] Tiantian Liu, Minming Li, and Chun Jason Xue. “Minimizing WCET for real-time embedded systems via static instruction cache locking”. In: *Real-Time and Embed. Technol. and Applications Symp.* IEEE. 2009, pp. 35–44.

[67] Daniel Lo, Mohamed Ismail, Tao Chen, and G Edward Suh. “Slack-Aware Opportunistic Monitoring for Real-Time Systems”. In: *Real Time and Embed. Technol. and Applications Symp.* IEEE. 2014, pp. 203–214.

[68] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. “NetFPGA–an open platform for gigabit-rate network switching and routing”. In: *Microelectronic Systems Education, 2007. MSE’07. IEEE International Conference on*. IEEE. 2007, pp. 160–161.

[69] Thomas Lundqvist and Per Stenstrom. “Timing anomalies in dynamically scheduled microprocessors”. In: *Proc. of Real-time Syst. Symp.* IEEE. 1999, pp. 12–21.

[70] Arno Luppold, Benjamin Menhorn, Heiko Falk, and Frank Slomka. “A new concept for system-level design of runtime reconfigurable real-time systems”. In: *ACM SIGBED Rev.* 10.4 (2013), pp. 57–60.

[71] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. “Analysis of loops”. In: *Compiler Construction*. Springer. 1998, pp. 80–94.

[72] Tulika Mitra, Jürgen Teich, and Lothar Thiele. “Time-Critical Systems Design: A Survey”. In: *IEEE Design & Test* 35.2 (2018), pp. 8–26.

[73] Saoni Mukherjee, Yifan Sun, Paul Blinzer, Amir Kavyan Ziabari, and David Kaeli. “A comprehensive performance analysis of HSA and OpenCL 2.0”. In: *IEEE Int. Symp. on Perf. Analy. of Syst. and Soft.* IEEE. 2016, pp. 183–193.

[74] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “OctoPOS: A parallel operating system for invasive computing”. In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*. 2011, pp. 9–14.

[75] Prasanna Pandit and R Govindarajan. “Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices”. In: *IEEE/ACM Int. Symp. on Code Gen. and Opt.* ACM. 2014, p. 273.

[76] Chang Yun Park and Alan C Shaw. “Experiments with a program timing tool based on source-level timing schema”. In: *Proc. of Real-time Syst. Symp.* IEEE. 1990, pp. 72–81.

[77] Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø. “A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems”. In: *Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*. IEEE. 2017, pp. 92–100.

[78] Sascha Plazar, Jan C. Kleinsorge, Peter Marwedel, and Heiko Falk. “WCET-aware Static Locking of Instruction Caches”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM. 2012, pp. 44–52.

[79] Alexander Pöppl, Michael Bader, Tobias Schwarzer, and Michael Glaß. “SWE-X10: Simulating shallow water waves with lazy activation of patches using ActorX10”. In: *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*. IEEE Press. 2016, pp. 32–39.

[80] Alexander Pöppl, Marvin Damschen, Florian Schmaus, Andreas Fried, Manuel Mohr, Matthias Blankertz, Lars Bauer, Jörg Henkel, Wolfgang Schröder-Preikschat, and Michael Bader. “Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model Using Reconfigurable Hardware in Invasive Computing”. In: *Workshop on UnConventional High Performance Computing (UCHPC), Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*. 2017, pp. 676–687. DOI: 10.1007/978-3-319-75178-8_54.

[81] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. “Heterogeneous system coherence for integrated CPU-GPU systems”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2013, pp. 457–467.

[82] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. “gem5-gpu: A Heterogeneous CPU-GPU Simulator”. In: *IEEE Comp. Arch. Letters* 14.1 (2015), pp. 34–36.

[83] Peter Puschner and Ch Koza. “Calculating the maximum execution time of real-time programs”. In: *Real-Time Syst.* 1.2 (1989), pp. 159–176.

[84] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.

[85] Gurulingesh Raravi. “The Journey Towards Reconciling Performance and Predictability”. In: *CODES+ISSS: Special Session - Future Automotive Systems Design: Research Challenges and Opportunities*. 2018.

[86] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. “A Definition and Classification of Timing Anomalies”. In: *WCET* 4 (2006).

[87] Christine Rochange and Pascal Sainrat. “A context-parameterized model for static analysis of execution times”. In: *Trans. on High-Performance Embed. Architect. and Compilers*. Springer, 2009, pp. 222–241.

[88] Enrico Rossi, Marvin Damschen, Lars Bauer, Giorgio Buttazzo, and Jörg Henkel. “Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing with FPGAs”. In: *ACM Trans. on Reconfig. Technol. and Syst. (TRETS)* 11.2 (2018). to appear. DOI: 10.1145/3182183.

[89] Sangeet Saha, Arnab Sarkar, and Amlan Chakrabarti. “Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms”. In: *IEEE Embedded Systems Letters* 7.1 (2015), pp. 23–26.

[90] Martin Schoeberl. “Time-predictable computer architecture”. In: *EURASIP Journal on Embed. Syst.* 2009 (2009), p. 2.

[91] Kiran Seth, Aravindh Anantaraman, Frank Mueller, and Eric Rotenberg. “FAST: Frequency-aware Static Timing Analysis”. In: *ACM Trans. Embed. Comput. Syst.* 5.1 (Feb. 2006), pp. 200–224. ISSN: 1539-9087. DOI: 10.1145/1132357.1132364.

[92] Christoph Steiger, Herbert Walder, and Marco Platzner. “Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks”. In: *IEEE Trans. on Computers* 53.11 (2004), pp. 1393–1407.

[93] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. “Online scheduling and placement of real-time tasks to partially reconfigurable devices”. In: *Proc. of Real-Time Syst. Symp.* IEEE. 2003, pp. 224–235.

[94] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. “WCET Centric Data Allocation to Scratchpad Memory”. In: *Proceedings of the 26th IEEE International Real-Time Syst. Symposium*. RTSS ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 223–232. ISBN: 0-7695-2490-7. DOI: 10.1109/RTSS.2005.45.

[95] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. “Invasive computing: An overview”. In: *Multiprocessor System-on-Chip*. Springer, 2011, pp. 241–268.

[96] Russell Tessier and Wayne Burleson. “Reconfigurable computing for digital signal processing: A survey”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 28.1-2 (2001), pp. 7–27.

[97] Russell Tessier, Kenneth Pocek, and Andre DeHon. “Reconfigurable Computing Architectures”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 332–354.

[98] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. “Fast and precise WCET prediction by separated cache and path analyses”. In: *Real-Time Syst.* 18.2-3 (2000), pp. 157–179.

[99] Lothar Thiele and Reinhard Wilhelm. “Design for timing predictability”. In: *Real-Time Syst.* 28.2-3 (2004), pp. 157–177.

[100] Sascha Uhrig, Stefan Maier, Georgi Kuzmanov, and Theo Ungerer. “Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling”. In: *Proc. of Int. Symp. Parallel and Distributed Processing*. IEEE. 2006, 4–pp.

[101] Sascha Uhrig, Stefan Maier, and Theo Ungerer. “Toward a processor core for real-time capable autonomic systems”. In: *Proc. of Int. Symp. Signal Processing and Information Technology*. IEEE. 2005, pp. 19–22.

[102] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. “The MOLEN Polymorphic Processor”. In: *IEEE Trans. Comput.* 53.11 (Nov. 2004), pp. 1363–1375. ISSN: 0018-9340. DOI: 10.1109/TC.2004.104.

[103] Li Wang, Ren-Wei Tsai, Shao-Chung Wang, Kun-Chih Chen, Po-Han Wang, Hsiang-Yun Cheng, Yi-Chung Lee, Sheng-Jie Shu, Chun-Chieh Yang, Min-Yih Hsu, et al. “Analyzing OpenCL 2.0 workloads using a heterogeneous CPU-GPU simulator”. In: *IEEE Int. Symp. on Perf. Analy. of Syst. and Soft.* IEEE. 2017, pp. 127–128.

[104] Jack Whitham and Neil Audsley. “MCGREP–A Predictable Architecture for Embedded Real-Time Systems”. In: *Proc. of Real-Time Syst. Symp.* IEEE. 2006, pp. 13–24.

[105] Stefan Wildermann, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jürgen Teich, Andreas Weichslgartner, and Andreas Zwinkau. “Invasive computing for timing-predictable stream processing on MPSoCs”. In: *it - Information Technology* 58.6 (2016), pp. 267–280. DOI: 10.1515/itit-2016-0021.

[106] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389.

[107] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems". In: *Trans. on Comput.-Aided Design of Integrated Circuits and Syst.* 28.7 (2009), pp. 966–978.

[108] Ralph D Wittig and Paul Chow. "OneChip: An FPGA processor with reconfigurable logic". In: *Proc. of Int. Symp. FPGAs for Custom Comput. Machines*. IEEE. 1996, pp. 126–135.

[109] Ming Yang, Ying Wu, James Crenshaw, Bruce Augustine, and Russell Mareachen. "Face detection for automatic exposure control in handheld camera". In: *Fourth IEEE International Conference on Computer Vision Systems (ICVS'06)*. IEEE. 2006, pp. 17–17.

[110] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. "CPU-assisted GPGPU on fused CPU-GPU architectures". In: *Int. Symp. on High Perf. Comp. Arch.* IEEE. 2012, pp. 1–12.

[111] Pan Yu and Tulika Mitra. "Scalable custom instructions identification for instruction-set extensible processors". In: *Proc. of Int. Conf. on Compilers, Architecture and Synthesis for Embed. Syst.* ACM. 2004, pp. 69–78.

[112] Pan Yu and Tulika Mitra. "Satisfying real-time constraints with custom instructions". In: *Int. Conf. on Hardware/Software Codesign and Syst. Synthesis*. IEEE. 2005, pp. 166–171.

[113] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. "FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures". In: *IEEE/ACM Int. Symp. on Code Gen. and Opt.* IEEE. 2017, pp. 27–38.

[114] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. "Understanding co-running behaviors on integrated CPU/GPU architectures". In: *IEEE Trans. on Par. and Dist. Syst.* 28.3 (2017), pp. 905–918.