WCET Analysis and Optimization for Multi-Core Real-Time Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Timon Kelter

Dortmund

2015

Tag der mündlichen Prüfung:	12. März 2015
Dekan / Dekanin:	Prof. Dr. Gernot Fink
Gutachter / Gutachterinnen:	Prof. Dr. Peter Marwedel
	Prof. Dr. Isabelle Puaut

Acknowledgments

First and foremost I want to thank my advisor Prof. Dr. Peter Marwedel for pointing my research into a rewarding direction from the start on and for providing me with the opportunity to work on this fascinating field of research in an inspiring, international team. Without his continued support this thesis would not exist. I would also like to thank Prof. Dr. Isabelle Puaut for her time and commitment to review this thesis.

The implementation of the WCET analyzer and the WCET optimizations which are presented in this thesis would not have been possible without the previous work of numerous colleagues at our chair over the course of more than one decade. I am especially grateful for the provision of the WCC framework, on which the majority of my practical work was built. In this context, I owe special thanks to Prof. Dr. Heiko Falk for being one of the most thorough reviewers and advisors I have ever met, to Dr. Paul Lokuciejewski for introducing me to this interesting field of research, to Dr. Sascha Plazar for being a fantastic office neighbor, to Jan Christopher Kleinsorge for our mutual motivation to finish the PhD project and to all of them for the enjoyable time in countless on- and off-topic disucssions.

For proof-reading my drafts and papers, for helpful discussions and for being really good colleagues I would also like to thank Björn Bönninghoff, Olaf Neugebauer, Pascal Libuschewski, Chen-Wei Huang, Dr. Michael Engel, Helena Kotthaus, Andreas Heinig, Florian Schmoll and Dr. Daniel Cordes. On the implementation side, my work was supported by Jan Körtner, Hendrik Borghorst, Tim Harde, Christian Günter and Henning Garus. Without their help the whole WCET analyzer implementation would be in a different shape now.

Furthermore, I am deeply thankful towards the former WCET analysis team at the National University of Singapore, most of all to Dr. Sudipta Chattopadhyay and Prof. Dr. Abhik Roychoudhury, for giving me the opportunity to work on the Chronos WCET analyzer. Without these first steps I possibly would have not found my way into the topic of WCET analysis.

What has kept me going in the last years was of course not only scientific progress but also the support that I received from friends and family, who kept me grounded when my thoughts were spinning around work issues. Many thanks to all of you – you know who you are. In particular I owe my father a big debt of gratitude for motivating me to pick up computer science as a profession and to finally strive for the PhD.

Abstract

During the design of safety-critical real-time systems, developers must be able to verify that a system shows a timely reaction to external events. To achieve this, the *Worst-Case Execution Time* (WCET) of each task in such a system must be determined. The WCET is used in the schedulability analysis in order to verify that all tasks will meet their deadlines and to verify the overall timing of the system. Unfortunately, the execution time of a task depends on the task's input values, the initial system state, the preemptions due to tasks executing on the same core and on the interference due to tasks executing in parallel on other cores. These dependencies render it close to impossible to cover every feasible timing behavior in measurements. It is preferable to create a static analysis which determines the WCET based on a safe mathematical model.

The static WCET analysis tools which are currently available are restricted to a single task running uninterruptedly on a single-core system. There are also extensions of these tools which can capture the effects of multi-tasking, i.e., preemptions by higher-priority tasks, on the WCET for certain well-defined scenarios. These tools are nowadays already used to verify industrial real-time software, e.g., in the automotive and avionics domain. Up to now, there are no mature tools which can handle the case of parallel tasks on a multi-core platform, where the tasks potentially interfere with each other.

This dissertation presents multiple approaches towards a WCET analysis for different types of multi-core systems. They are based upon previous work on the modeling of hardware and program behavior but extend it to the treatment of shared resources like shared caches and shared buses. We present multiple methods of integrating shared bus analysis into the classical WCET analysis framework and show that time-triggered bus arbitration policies can be efficiently analyzed with high precision. In order to get precise WCET estimations for the case of shared caches, we present an efficient analysis of interactions in parallel systems which utilizes timing information to cut down the search space. All of the analyses were implemented in a research C compiler. Extensive evaluations on real-time benchmarks show that they are up to 11.96 times more precise than previous approaches.

Finally, we present two compiler optimizations which are tailored towards the optimization of the WCET of tasks in multi-core systems, namely an evolutionary optimization of shared resource schedules and an instruction scheduling which uses WCET analysis results to optimally place shared resource requests of individual tasks. Experiments show that the two combined optimizations are able to achieve an average WCET reduction of 33%.

During the course of this thesis, a complete WCET analysis framework was developed which can be used for further work like the integration of multi-task and multi-core-aware techniques into a single analyzer.

Publications

Parts of this thesis have been published in journals and the proceedings of the following conferences and workshops (in chronological order):

Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. “Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds”. In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS).* Porto, Portugal, 07/2011, pp. 3–12.

Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Heiko Falk, and Peter Marwedel. “A Unified WCET Analysis Framework for Multi-Core Platforms”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* Beijing, China, 04/2012, pp. 99–108.

Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. “Evaluation of Resource Arbitration Methods for Multi-Core Real-Time Systems”. In: *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis (WCET).* Ed. by Claire Maiza. Paris, France, 07/2013.

Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. “Static Analysis of Multi-Core TDMA Resource Arbitration Delays”. English. In: *Real-Time Systems* 50.2 (03/2014), pp. 185–229. ISSN: 0922-6443. DOI: `10.1007/s11241-013-9189-x`. URL: `http://dx.doi.org/10.1007/s11241-013-9189-x`.

Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. “A Unified WCET Analysis Framework for Multicore Platforms”. In: *ACM Transactions on Embedded Computing Systems* 13.4s (04/2014), 124:1–124:29. ISSN: 1539-9087. DOI: `10.1145/2584654`. URL: `http://doi.acm.org/10.1145/2584654`.

Timon Kelter, Peter Marwedel, and Hendrik Borghorst. “WCET-aware Scheduling Optimizations for Multi-Core Real-Time Systems”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* Samos, Greece, 07/2014.

Chen-Wei Huang, Timon Kelter, Bjoern Boenninghoff, Jan Kleinsorge, Michael Engel, Peter Marwedel, and Shiao-Li Tsao. “Static WCET Analysis of the H.264/AVC Decoder Exploiting Coding Information”. In: *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* IEEE. Chongqing, China, 08/2014.

Timon Kelter and Peter Marwedel. “Parallelism Analysis: Precise WCET Values for Complex Multi-Core Systems”. In: *Third International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS).* Ed. by Cyrille Artho and Peter Ölveczky. Luxembourg: Springer, 11/2014.

Contents

Chapter 1

Introduction

This dissertation deals with an aspect of computer science that has been a second-class citizen since the emergence of the discipline: With *time*. Or, to put it more precisely, with *safe bounds on the timing behavior* of programs running on a computer system.

Since the days of mainframes, attempts have been made to increase the productivity of a programmer by supplying him with ever more powerful programming languages and compilers for the latter [Myc07], by teaching useful idioms and patterns [GHJ+95] and by making the single computers faster to allow the software complexity to increase steadily [Sch97]. The runtime behavior of algorithms has traditionally been classified asymptotically in big O notation which eases or even enables the reasoning about runtime behavior for complex algorithms [Weg03]. For many computer applications the asymptotical classification is sufficient, even though its limitations are already stressed by examples like the Simplex algorithm, which in spite of having exponential asymptotical runtime performs better than its polynomial-time counterparts on most real-world examples [Cor10, Choosing an optimizer for your LP problem].

The major shortcoming of this type of runtime classification is that it is not usable in the important area of *real-time systems*, i.e., computer systems in which the executed tasks *must* always fulfill their work in a bounded time interval or before a given deadline. An asymptotic modeling which ignores constant factors in the runtime formula is not applicable here, since taking twice the allowed time or only once is an important difference, possibly rendering the system dysfunctional in the former case. Most real-time systems are also *embedded systems*, i.e., “information processing systems embedded into enclosing products” [Mar11], which are integrated into many devices of daily life. Application domains for embedded systems are numerous and cover areas like automotive electronics, avionics, railways, telecommunication, the health sector, security, consumer-electronics, fabrication equipment, smart buildings, logistics, robotics, military applications, and many more [Mar11].

Real-time systems are generally classified as *soft* or *hard* real-time systems, where “soft” means that deadlines may be violated for *a few* executions of a task, but not regularly and “hard” means that not a single deadline must be violated. Multimedia applications like audio and video decoders are prime examples for soft real-time systems, whereas industrial *Electronic Control Units* (ECU) in robotics, power stations, cars and planes are typical hard-real time systems. Modern cars for example have more than 70 ECUs [ES08] for engine control, safety features like anti-lock braking

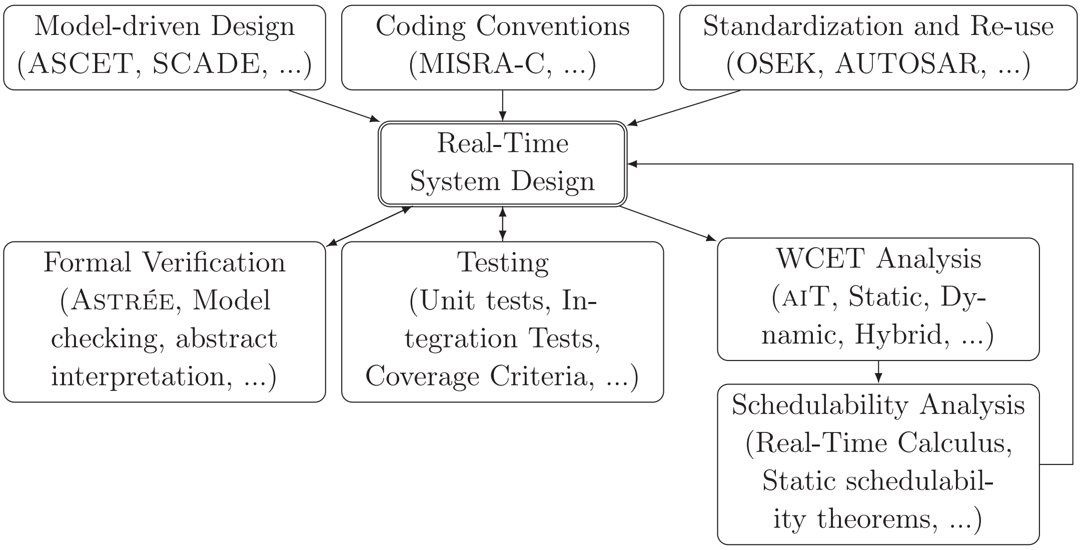

Model-driven Design (ASCET, SCADE, ...)
Coding Conventions (MISRA-C, ...)
Standardization and Re-use (OSEK, AUTOSAR, ...)
Real-Time System Design
Formal Verification (Astrée, Model checking, abstract interpretation, ...)
Testing (Unit tests, Integration Tests, Coverage Criteria, ...)
WCET Analysis (aiT, Static, Dynamic, Hybrid, ...)
Schedulability Analysis (Real-Time Calculus, Static schedulability theorems, ...)

Figure 1.1: Real-time system verification tools.

and electronic stability program and multi-media functions. At least for the first two categories, hard-real time implementations clearly must be provided to ensure a timely reaction of the system. According to a recent market study, about 60% of all embedded development projects [BW13] require real-time capabilities, though in some of these cases soft-real time may be sufficient. Still, the rising number of complications with real-world safety-critical embedded systems of everyday life, mostly cars[1], demonstrates that (semi-)automated safety certification of embedded systems is highly desirable. Here, WCET analysis comes into play, as a means to semi-automatically verify the timing behavior of the task set under analysis. Of course, this has to be complemented by other analyses as sketched in Figure 1.1.

The upper half of the figure shows methodologies, coding conventions and standard components which are used to avoid programming errors and to increase the productivity. They have a direct influence on the shape of the final real-time system code. As an example, code generation from models and conventions like MISRA-C [MIS13] can ease the WCET analysis by limiting the variability of the code and prohibiting hard-to-analyze software constructions.

The lower half of Figure 1.1 shows tools which are used to verify a system that has already been partly or fully designed. Formal verification is needed to detect run-time errors like, e.g., null-pointer, overflow, out-of-bounds bugs. One of the best-known tools in this area is Astrée[2] [Abs14b] which similar to WCET analysis relies on *abstract interpretation* to derive static information about the program. Model-checking has also proven to be useful especially when a system implementation is already generated from a high-level model. Model checkers focus on proving the

[1] As an example, there were 24 retractions of vehicle classes in the year 2011 [Ele12].

[2] Here and throughout the rest of this thesis, example tools are set in small caps.

absence of deadlocks, reachability conditions and program termination, rather than on run-time errors in real code. Testing is mandatory for any type of development, with a wealth of testing frameworks and methodologies being available and widely used.

WCET and schedulability analysis are then providing what formal error-checking and testing alone cannot offer, namely safe and precise bounds on the runtime of individual tasks (WCET analysis) and statements on whether the given task set will always meet its deadline on the given platform (Schedulability analysis). aiT [Abs14a] is the de-facto standard for industrially used WCET analysis. In addition to delivering highly precise WCET values for single tasks, it can also compute the worst-case memory consumption of tasks.

Therefore, WCET analysis is one of the key elements in timing verification. With imprecise, or worse with unsafe WCET values all statements derived in the schedulability analysis are overly pessimistic or even void.

1.1 Motivation

Unfortunately, a precise WCET analysis is undecidable in general. This is easy to see, since the WCET analysis problem is just an extension of the halting problem, where we not only ask *whether* a program will terminate, but also *when* it will terminate. Since the halting problem is undecidable [Weg99], so is the WCET analysis problem[3]. In addition, the timing behavior of modern hardware shows enough variance to render a simple enumeration of all possible execution paths infeasible.

Therefore, any practically feasible WCET analysis has to use *approximation* as a means to make the problem decidable at all. We will see in Chapter 2 that even with algorithmic approximation, WCET analysis potentially still requires some user interaction for complex tasks.

Since WCET analysis in general requires approximation, we distinguish between the WCET_{real}, which is the true worst-case runtime of the task under analysis, and estimations $\mathrm{WCET}_{est} \geq \mathrm{WCET}_{real}$. In Figure 1.2 an example distribution of runtimes is given, where the WCET_{real} is marked along with with one possible example of a WCET_{est} for this task. It is worth noticing that for any real task a runtime distribution as shown in Figure 1.2 can hardly be constructed, since it needs to cover all input combinations and all possible initial system states. Similar to the WCET we can also define the *Best-Case Execution Time* (BCET) of a task which is also indicated in Figure 1.2, where we again distinguish between the BCET_{real} and the $\mathrm{BCET}_{est} \leq \mathrm{BCET}_{real}$. The BCET is also important in the timing analysis, to, e.g., derive minimum task inter-arrival times in the schedulability analysis, but

[3] Even though Kirner, Zimmermann and Richter argue that the halting problem is in fact *not* undecidable for existing bounded-memory platforms [KZR09], it still is orders of magnitude too complex to decide in reasonable time.

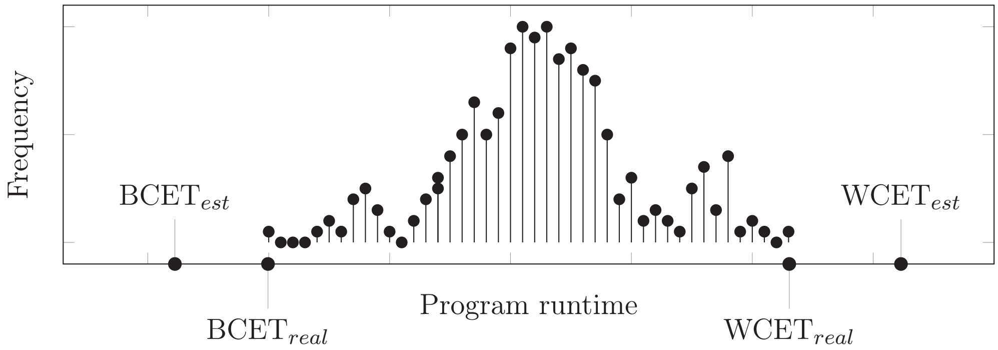


Figure 1.2: A sample distribution of runtimes of a program, along with with sample BCET and WCET estimates.

since most WCET analysis concepts are also directly applicable to BCET analysis, we focus on the WCET side in the following. Nevertheless, most analyses from Chapter 4 and Chapter 5 yield both BCET and WCET values. Since the WCET$_{real}$ is unknown in general, we use the term "WCET" as a synonym for "WCET$_{est}$" throughout this thesis (same for BCET).

Any WCET analysis is required to be *safe*, i.e., the relation WCET$_{est}$ ≥ WCET$_{real}$ must always hold. In addition, it is desirable for the analysis to be *precise*, which means that the difference WCET$_{est}$ − WCET$_{real}$ should be minimized. To the best of the author's knowledge, theoretical bounds on the precision of WCET analyses are not available, so the precision is usually determined empirically by comparing the WCET$_{est}$ with measured execution times.

For a task which runs uninterruptedly on a single core, effective WCET analysis methodologies were developed and compared throughout the last two decades, which deliver WCET estimates which are 3% to 25% above the WCET$_{real}$ [Abs14a]. One key problem for these analyses is that preemptions by other tasks are usually not accounted for in the WCET analysis itself, but only during schedulability analysis (cf. Figure 1.1). This decomposition of WCET and schedulability eases the analysis of both, but also promotes overestimation, since the communication between WCET and schedulability analysis is unidirectional and on a very coarse level. Most importantly, the schedulability analysis intentionally has no access to the detailed worst-case hardware states that are generated by the WCET analysis. Instead, the schedulability tests are solely based on numeric WCET values and platform assumptions to lower their algorithmic complexity.

WCET analysis faces an equally important problem due to the latest hardware development trends, namely the shift towards multi-core architectures. While until 2005 the performance improvement of new chips mainly was generated by frequency increases, we are now faced with stagnating frequencies and rising numbers of cores per chip [Sut12]. This can be seen as a general trend towards parallel computing which does not stop at homogeneous multi-cores, but continues with heterogeneous

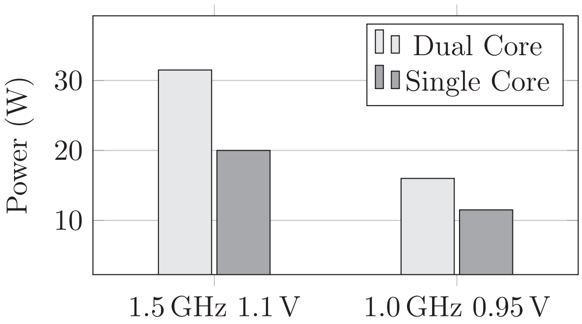


Figure 1.3: Multi-core implementation power consumption for the FreeScale MPC8641 [Fre09, Section 1.1], depending on the core frequency and voltage.

multi-cores (since 2009) and "elastic cloud compute cores" (since 2010), e.g., the outsourcing of computations to commercial or private clouds. While the latter is not expected to have a big impact on hard real-time systems, multi-cores have already arrived in the embedded market.

As Equation 1.1 shows, the power consumption of a core is directly proportional to the frequency, but according to [Fre09] increased voltages are needed to reach higher frequencies due to electrotechnical reasons. Therefore, by rule of thumb, a doubling of frequencies leads to a fourfold increase in power consumption [Fre09].

$$\text{Power} \propto \text{Capacitance} \times \text{Voltage}^2 \times \text{Frequency} \tag{1.1}$$

This power consumption increase in turn generates more heat which leads to decreased component lifetime and higher cooling needs. For battery-driven devices like many embedded systems all of these points are highly important and motivate the now widespread use of multi-cores also in the embedded domain. As an example, Figure 1.3 shows the power consumption of the FreeScale MPC8641 in single- and dual-core configurations as found in [Fre09]. It is visible that the dual-core configuration only needs about 30% more power than the single-core one, whereas a single-core implementation with doubled frequency would have caused an four-fold increase of the power demand as mentioned above. A more extreme example is the IBM SyNAPSE chip, according to IBM research the biggest chip that was ever built by IBM, which integrates 4096 cores on a single chip [Joh14]. Each core is running at 1 kHz only, leading to a marginal power consumption of 70 mW. Though this chip is not yet in volume production, it shows the capabilities of multi-cores. Finally, Figure 1.4 presents recent ARM architectures which can be found in abundance in modern smartphones and embedded systems. It is visible that high-performance (with the exception of the Cortex-A8) is only achieved with multi-core designs, especially since these chips are often used in passively cooled systems where the heating problem mentioned above is a severe issue. For the ARM designs and similar chips which are intended to offer more than some kilohertz of per-core frequency it is

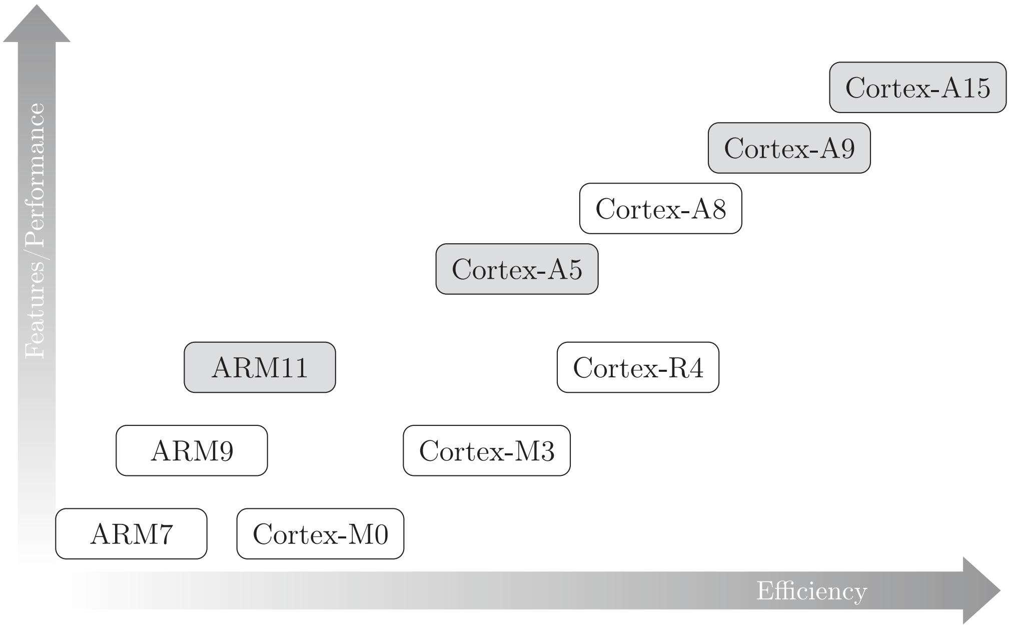


Figure 1.4: ARM Processor Families [ARM14b]. Multi-core architectures are marked in gray.

already predicted that the multi-core scaling will finally end due to thermal issues which arise when all cores on the same chip are powered on [EBS+11]. This effect is also called "dark silicon", since it implies that not all cores can be powered on at the same time without causing the system to overheat. Therefore, even though the integration density may still increase the number of concurrently usable cores does not, because some of them must be powered off, i.e., they are "dark silicon". Nevertheless, multi-cores will continue to scale for some time and will remain the predominant type of computing system for the foreseeable future.

From the WCET perspective, the problem with this inevitable trend is that the cores in a multi-core system usually share some hardware components for efficiency reasons. Typical examples of this are shared I/O-devices, shared main memory and shared cache levels. Since these resources can only be accessed by one task at a time, concurrent requests to the resource need to be sequentialized by some kind of arbiter. This *arbitration delay* now has to be bounded as precisely as possible by the static WCET analysis, which may be hard to achieve, depending on the employed arbitration strategy. In addition, some shared components have a *shared state* which determines the timing behavior of the component. A prime example for this is a shared cache where it makes a big difference in terms of timing whether the requested block was found in the cache or not. Therefore WCET analysis now faces the problem of finding static estimates of

1. the arbitration delay and
2. the shared state

of any shared resource, or to put it in other words, the *interference* on this resource. Both may depend on the *order of concurrent accesses* which are issued to the resource. As an example, a shared cache will have a different state if two conflicting cache blocks A and B are requested in the order A, B, A, B or B, B, A, A. Mature WCET analysis tools for multi-cores are not yet available, therefore the alarming reaction of industrial real-time system designers is to switch all but one single core off [WHK+13]. This removes all interference-related problems, but of course also effectively degrades the multi-core system to a single-core one. Since the most recent hardware generation is often only available as multi-core chips, this is sometimes still the only viable option.

1.2 Contributions of this Work

This thesis presents multiple approaches towards a precise WCET analysis for multi-cores, which may help to alleviate the aforementioned problems with the adoption of multi-core hardware in real-time system design. Also, the concrete implementation of these approaches inside the *WCET-aware C Compiler* (WCC) is demonstrated and used to evaluate the presented techniques. The WCC also provides unique opportunities to couple the analysis of a task's WCET with the optimization of the latter. Therefore, this thesis also presents two optimizations which utilize this unique capability to demonstrate that an optimization of task WCETs is feasible and useful in practice.

We build our approach on the branch of WCET analysis methods which has proven to be most efficient in the past, namely on a decomposition of the WCET analysis into microarchitectural analysis and path analysis. This approach is also employed in the de-facto standard WCET analyzer aiT [Abs14a]. It consists of an abstract interpretation-based microarchitectural analysis phase, during which the best- and worst-case runtime of each basic block in the task's control-flow graph is determined. With these values a successive path analysis can compute the shortest and longest path through the control-flow graph, whose length corresponds to the BCET and WCET, respectively.

For the single-core case, the WCET analysis methodology is reviewed. Focus is laid on how to achieve a value analysis with sufficient precision and on the modeling of microarchitectural features, since these form the basis for a precise WCET analysis of both single- and multi-core systems.

The extension of the microarchitectural modeling to include shared buses and shared caches is discussed, and the possibility to model the behavior of time-triggered arbiters by means of *TDMA offsets* is presented. These can be used to statically capture the duration of shared resource access requests. Their true potential only unfolds when they are not applied naively, but combined with intelligent loop unrolling techniques. Since the loop unrolling incurs a significant analysis time overhead, the concepts of *offset relocation* and *offset contexts* are established to allow for a fast but also precise WCET analysis.

The timing behavior of stateful resources and non-time-triggered arbiters relies on the *ordering* of accesses, therefore any analysis which tries to bound this timing must safely consider all possible interleavings of potentially parallel actions. This thesis for the first time presents a structured way to explore such interleavings on the single-machine-cycle level. To avoid some part of the combinatorial explosion that is inevitable in such analyses, a timing-based pruning criterion is developed which uses the generated timing information to rule out invalid parallel system states.

The presented analysis methods are compared with respect to the achievable precision and to the overhead which is incurred by platform configurations which are routinely advertised for being more predictable and thus easier to analyze.

Finally, two compiler optimizations are given, which can be used to decrease the WCET of tasks in a multi-core system. The first is a WCET-aware multi-criteria evolutionary optimization of the schedule of shared resources. By efficiently exploring different system configurations a suitable schedule can be found for a given task set. It is also shown that this schedule is highly dependent on the given tasks, i.e., an optimal default schedule cannot be specified in general, necessitating an optimization like the one presented here.

The second optimization is a WCET-driven instruction scheduling which utilizes the close coupling of analyses and optimizations inside the WCC by using microarchitectural analysis results to optimally place single instructions inside the tasks' source code. The placement is done in such a way that the single shared resource accesses incur minimum access overhead, which is only possible with the help of detailed microarchitectural information.

1.3 Organization of the Thesis

The structure of the remaining parts of this thesis is as follows:

- **Chapter 2** presents the existing concepts used in WCET analysis. It starts with a general review of abstract interpretation, the basic method onto which the presented WCET analysis is built. Subsequently, different approaches to WCET analysis are shown, and necessary definitions of WCET-related phenomena like timing compositionality and timing anomalies are given. The chapter closes with a presentation of preexisting approaches to WCET analysis in multi-task and multi-core systems.
- The WCC framework is introduced in **Chapter 3**. Starting with an overview of the compiler infrastructure and related work on similar projects, it covers the assumed system model and the compiler phases. Finally, extensions of the WCC framework to incorporate binary code into WCET analysis and optimization are presented.

- **Chapter 4** picks up the WCET analysis concepts presented in Chapter 2 and demonstrates how these are used in the context of the WCC framework to create a WCET (and BCET) analyzer for a single-core ARM platform.
- The main contributions of this thesis are found in **Chapter 5**, where the methods for WCET analysis in multi-core systems are discussed. It covers TDMA offset analysis for time-triggered arbiters as well as precise analysis for less predictable architectures and an extensive evaluation.
- **Chapter 6** contains the presentation of the aforementioned two compiler optimizations tailored towards WCET optimization. It starts with the evolutionary shared resource schedule optimizations and then proceeds with the WCET-aware multi-core instruction scheduler. For both optimizations, evaluation results on a large number of real-time benchmarks are given.
- Finally, **Chapter 7** closes the thesis with a summary and an outlook on future work.

1.4 Author's Contribution to this Dissertation

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation within the context of doctoral studies has to contain a separate list that highlights the author's contributions to research and results obtained in cooperation with other researchers. Therefore, the following overview lists the contribution of the author on the presented results for each chapter:

- **Chapter 2:** This chapter summarizes related work only, therefore there is no contribution to account for.
- **Chapter 3:** The WCC framework [FL10] was created by a multitude of people, among others Heiko Falk, Paul Lokuciejewski, Sascha Plazar and Jan Kleinsorge. The author has also worked on this framework to some extent by using it as a basis for the multi-core WCET analysis and optimizations. The extension of the WCC framework to multi-core architectures was done by the author only. An initial version of the virtual platform that was used to evaluate the proposed architecture was implemented by Tim Harde [Har13] and later largely re-structured by the author. The employed cycle-true virtual platform simulator CoMET was donated by Synopsys Inc. [Syn14].
  The concepts for the extension of the WCC towards the handling of binary input files were developed by the author and implemented by Christian Günter [Gün13].
- **Chapter 4:** The WCC value analysis was developed by the author in collaboration with Jan Körtner, who carried out the majority of the implementation work. The analysis and context graph handling as well as the microarchitectural modeling as presented in this chapter are the work of the author.

- **Chapter 5:** The WCET analysis approaches for shared buses were entirely designed and developed by the author. They are based on previous work from Chattopadhyay et al. [CRM10], which was later extended in cooperation with the same authors in the publications [KFM+11; KFM+14]. The co-authors of these publications assisted the author in technical discussions, proof-reading and structuring of the publications. Furthermore, the presentation of the analyses in this thesis contains a state which is far more advanced than the one in [KFM+11; KFM+14] and integrates better with the classical microarchitectural analysis. These advances also are original work of the author.

  The comparison of arbitration strategies as already published in [KHM+13] was developed by the author of this thesis, based on the platform implementation by Tim Harde.

  The WCET analysis for stateful resources considering all possible interleavings of task executions was designed and implemented exclusively by the author and published in [KM14].
- **Chapter 6:** The optimization opportunities and concepts were developed and formalized by the author. A first version of the implementation was done by Hendrik Borghorst [Bor13] which was later reworked by the author, leading to the publication [KMB14].

Chapter 2

Timing Analysis Concepts

Contents

In this chapter we will review the existing literature on WCET analysis and investigate which concepts have already been established in this domain. We start in Section 2.1 with a thorough treatment of *abstract interpretation*, since this is one very fundamental technique that forms the basis for WCET analysis. In Section 2.2 we examine how this technique and others can be applied in the context of classical WCET analysis. Section 2.3 proceeds with concepts for the timing analysis of multiple tasks on a single processor. Finally, we present existing approaches for the analysis of the WCET of tasks in parallel systems in Section 2.4.

2.1 Abstract Interpretation

Abstract Interpretation (AI) is one of the most well-developed theories for the approximation of states in discrete transition systems. The basic ideas date back to a 1973 publication from Kildall [Kil73] which were later formalized and generalized by Cousot and Cousot [CC77]. The presentation in this thesis is based on the more modern introduction in [ALS+07]. Though AI is applicable to various discrete transitions systems like, e.g., source-code programs, petri nets and Kahn process networks, we base our examples on the approximation of low-level computer system states since these will be the target of WCET analysis.

In general, we can associate a concrete semantics with any computer system. This semantics reflects the transformation of all memory cells in the system by operations carried out by the computational circuits. Since all currently relevant computer systems are working in clocked operation, we can define these transformations on discrete time steps. Therefore, if $\tilde{L}$ is the set of all possible memory cell assignments including all registers, the program counter, current instruction and otherwise stored values, a single cycle of a computer system's operation can be deterministically described by a concrete semantics function $[\![]\!]_{conc} : \tilde{L} \mapsto \tilde{L}$. Since all relevant computer systems are also working on a well-defined instruction set, concrete states are always generated by programs.

Definition 1. *A* program $X = (i_0, i_1, \dots, i_n)$ *is a sequence of instructions* $i \in I$ *from a global instruction set* I *with start instruction* i_0 *and a set of terminal instructions* I_t*. An* execution *of* X *is a trace of concrete system states* $(\tilde{l}_0, \tilde{l}_1, \dots, \tilde{l}_m)$ *such that*

- $\tilde{l}_0$ *encodes* X *in its memory content and executes* i_0*, and*
- $\forall i \in \{1, m\} : \tilde{l}_i = [\![\tilde{l}_{i-1}]\!]_{conc}$*, and*
- $pc(\tilde{l}_m) \in I_t$.

where $pc : \tilde{L} \to I$ *extracts the value of the program counter from a concrete state.*

An ideal analysis would determine the execution trace for every possible initial system state $\tilde{l}_0$. The length of the longest of these traces would be the WCET of the program under analysis. Obviously, a naive attempt to collect all of these traces and return them as the analysis result will fail, since

a) there may be traces of infinite length corresponding to non-terminating programs and

b) modeling the *whole* state of the system under analysis and maintaining huge numbers of these states during the analysis is practically infeasible.

To be able to reason about program executions in a structured way, the notion of a control-flow graph is used.

Definition 2. *A* basic block $v = (i_0^v, \dots, i_k^v)$ *of a program* X *is a maximal subsequence of* X*, such that for all* $j \in \{1, \dots, k\}$ *and* $\tilde{l}, \tilde{l}' \in \tilde{L}$

$$\left(pc(\tilde{l}) = i_j^v \wedge [\![\tilde{l}']\!]_{conc} = \tilde{l}\right) \implies \left(pc(\tilde{l}') = i_j^v \vee pc(\tilde{l}') = i_{j-1}^v\right). \tag{2.1}$$

A Control Flow Graph *(CFG)* (V, E, v_0) *of a program* X*, with* V *being the set of basic blocks of* X *and* $E \subset V \times V$ *being a set of directed edges which model every possible transfer of control within the program. The entry point of the program is given by node* v_0.

A *path* P in a CFG from $v_s \in V$ to $v_e \in V$ is a non-empty sequence of nodes $P = (v_s, \dots, v_e)$ such that for any two adjacent v_i and v_{i+1} in P, $(v_i, v_{i+1}) \in E$ holds. If a path from $v_s \in V$ to $v_e \in V$ exists, we call v_e *reachable* from v_s which is expressed as $v_s \rightsquigarrow v_e$, else v_e is *unreachable* from v_s written as $v_s \not\rightsquigarrow v_e$. The set of all paths from v_s to v_e is called $P_{[v_s,v_e]}$. For any node $v \in V$, $\delta^+(v) = \{w \mid (v,w) \in E\}$ and $\delta^-(v) = \{w \mid (w,v) \in E\}$ are the successors and predecessors of v.

The connection between traces of concrete states and the CFG is easily made, since each $\tilde{l} \in \tilde{L}$ holds a concrete value of the *Program Counter* (PC) register of the underlying architecture, which points to an instruction $i_{\tilde{l}} \in I$. Thus, if each $\tilde{l}_j$ of a trace is mapped to the node $v \in V$ with $i_{\tilde{l}_j} \in v$ the trace is transformed to a CFG path.

A path is called *feasible*, if there is a concrete trace which is mapped to this path. Infeasible paths may exist in a CFG due to dependencies between control flow branches, which are not reflected in the graph structure. As an example, if two branches b_1 and b_2 have the same branch condition and the value of the condition remains unmodified between b_1 and b_2, then either both branches are taken or none. The CFG, in contrast, may also contain an infeasible path in which b_1 is taken whereas b_2 is not. A *cycle* at $v \in V$ is a path $(v, \dots, v)$. A path which contains no cycle is called *acyclic*. The set of all paths $P_{[v_s,v_e]}$ can be restricted to feasible ($P^{\text{feasible}}_{[v_s,v_e]}$) or acyclic paths ($P^{\text{acyclic}}_{[v_s,v_e]}$).

To deal with issue *a)* from above, the *collecting semantics* $[\![]\!]_{coll} : 2^{\tilde{L}} \times V \mapsto 2^{\tilde{L}}$ is defined as taking a set of concrete states, executing the instructions from the given CFG node in those states for all possible inputs and returning the resulting end states. We can trivially extend the collecting semantics to paths $p = (v_1, v_2, \dots v_n)$ by setting $[\![s]\!](p) = [\![\dots[\![[\![s]\!](v_1)]\!](v_2)\dots]\!](v_n)$. We are then searching for the state set $s^{coll}_v = \bigcup_{p \in P_{[v_0,v]}, \tilde{l}_0 \in \tilde{L}} [\![\tilde{l}_0]\!](p)$ for every node $v \in V$. If $\tilde{L}$ has infinite cardinality, this result may still be infinitely big. To overcome this and the issue *b)* mentioned above, AI further changes from the collecting semantics $[\![]\!]_{coll}$ to an *abstract semantics* $[\![]\!]_{abs} : \mathbb{L} \times V \mapsto \mathbb{L}$ which is defined on abstract system states $\mathbb{L}$. To be useful, the abstract semantics must *overapproximate* the collecting one, which is expressed by the notion of a Galois connection.

Definition 3. *A* Galois connection (α, γ) *is a pair of functions* $\alpha : P \mapsto Q$ *and* $\gamma : Q \mapsto P$ *for two sets with partial orders* $(P, \leq)$ *and* $(Q, \sqsubseteq)$ *such that*

$$\forall x \in P, y \in Q : \alpha(x) \sqsubseteq y \Leftrightarrow x \leq \gamma(y) \tag{2.2}$$

For our concrete example, this means $(P, \leq) = (2^{\tilde{L}}, \subseteq)$ and $(Q, \sqsubseteq) = (\mathbb{L}, \sqsubseteq)$, where $x \sqsubseteq y$ if and only if $\gamma(x) \subseteq \gamma(y)$. I.e., an abstract state x is "bigger" than y if and

$$\begin{tikzcd}
2^{\tilde{L}} \arrow[r, "\llbracket\rrbracket_{coll}"] \arrow[d, "\alpha"'] & 2^{\tilde{L}} \\
\mathbb{L} \arrow[r, "\llbracket\rrbracket_{abs}"'] & \mathbb{L} \arrow[u, "\gamma"']
\end{tikzcd}$$

Figure 2.1: The Galois connection between concrete and abstract semantics.

only if it "contains" a superset of the concrete states which are contained in y. This is a general necessity in a Galois connection, where we also have the property, that $x \leq \gamma \circ \alpha(x)$, e.g., the re-concretization of an abstraction of x always contains x. The general idea is visualized in Figure 2.1. With the abstraction function α, we can map the initial system states into the abstract domain $\mathbb{L}$, where we conduct our analyses to compute abstract results l_v^{out}. After the analysis is finished the Galois condition ensures that $\gamma\left(l_v^{out}\right) \supseteq s_v^{coll}$, i.e., every reachable concrete state is covered in the result. Since we are computing an overapproximation, potentially also other states which are not reachable in any concrete execution are covered.

For the analysis to be efficiently possible, the abstract domain $\mathbb{L}$ must be a semi-lattice.

Definition 4. *A* semi-lattice $(\mathbb{L}, \sqcup)$ *is a set* $\mathbb{L}$ *with a* meet operator $\sqcup : \mathbb{L} \mapsto \mathbb{L}$*, which is required to be*

- *idempotent:* $\forall l \in \mathbb{L} : l \sqcup l = l$*,*
- *commutative:* $\forall l, m \in \mathbb{L} : l \sqcup m = m \sqcup l$ *and*
- *associative:* $\forall l, m, n \in \mathbb{L} : l \sqcup (m \sqcup n) = (l \sqcup m) \sqcup n$*.*

The meet operator induces a partial order $\sqsubseteq$ *on* $\mathbb{L}$*, which is defined by*

$$\forall l, m \in \mathbb{L} : l \sqsubseteq m \Leftrightarrow l \sqcup m = m \tag{2.3}$$

In a semi-lattice $(\mathbb{L}, \sqcup)$*, also a biggest element* $\top = \sqcup \mathbb{L}$ *exists, with* $\forall l \in \mathbb{L} : l \sqcup \top = \top$ *or equivalently* $\forall l \in \mathbb{L} : l \sqsubseteq \top$*. This element is usually named "top". Additionally, a smallest element* $\bot \in \mathbb{L}$ *("bottom") may exist, with* $\forall m \in \mathbb{L} : \bot \sqcup l = l$*.*

The height *of a lattice is one less than the length of the longest sequence* $(l_1, l_2, \ldots, l_n)$ *of* $l_i \in \mathbb{L}$ *such that* $\forall i : l_i \sqsubseteq l_{i+1}$*.*

Definition 5. *A monotonic* Data-Flow Analysis *(DFA)* framework $((\mathbb{L}, \sqcup), F)$ *consists of a*

- *semi-lattice* $(\mathbb{L}, \sqcup)$ *and*
- *a set* F *containing functions* $f : \mathbb{L} \to \mathbb{L}$*, where*
    1. *every* f *is monotonic in* $\sqsubseteq$*, i.e.,* $\forall l, m \in \mathbb{L} : l \sqsubseteq m \implies f(l) \sqsubseteq f(m)$*,*
    2. *the identity function id with* $\forall l \in \mathbb{L} : id(l) = l$ *is contained in* F *and*

3. F is closed under function composition, i.e., $\forall f,g \in F : f \circ g \in F$.

A DFA framework is called distributive *iff*

$$\forall f \in F, l, m \in \mathbb{L} : f(l \sqcup m) = f(l) \sqcup f(m) \tag{2.4}$$

The weaker form of this condition $f(l \sqcup m) \sqsubseteq f(l) \sqcup f(m)$ is already true also for non-distributive DFA frameworks.

Definition 6. *An* instance *of a DFA framework $((\mathbb{L}, \sqcup), F)$ is a tuple $((V,E), v_0, l_0)$, where (V,E) is a control-flow graph of the program to analyze, v_0 is the node where the control flow enters the program and $l_0 \in \mathbb{L}$ is the initial data-flow information for the start node and every node $v \in V$ has an associated* transfer function *$f_v \in F$. A* solution *for an instance is a set of data-flow items l_v^{out} for all $v \in V$ such that the reachable concrete states s_v^{coll} are covered, i.e., $\gamma\left(l_v^{out}\right) \sqsupseteq s_v^{coll}$.*

The transfer functions are just invocations of the abstract semantics, i.e. $f_v = [\![]\!]_{abs}(v)$ and similar to it, we can extend the transfer function to CFG paths. By following all feasible paths through the program's control flow graph given by a DFA framework instance, we can define an *ideal* solution

$$l_v^{out,\text{IDEAL}} = \bigsqcup_{p \in P_{[v_0,v]}^{\text{feasible}}} f_p(l_0) \tag{2.5}$$

Since the identification of feasible paths in impossible in general, we can relax this to the *Meet Over All Paths* (MOP) solution, which considers all paths in the CFG

$$l_v^{out,\text{MOP}} = \bigsqcup_{p \in P_{[v_0,v]}} f_p(l_0) \tag{2.6}$$

The MOP solution is still not computable, since there may be infinitely many paths due to the existence of loops in the CFG as already discussed. Therefore as the last coarsening step, we revert to the *Minimum Fixed Point* (MFP) solution which is the fixed point of Equation 2.7.

$$l_v^{out,\text{MFP}} = \begin{cases} f_v(l_0) & \text{if } v = v_0 \\ f_v\left(\bigsqcup_{(u,v)\in E} l_u^{out,\text{MFP}}\right) & \text{else} \end{cases} \tag{2.7}$$

This solution avoids the computation of all paths by only propagating the states via the edges of the CFG until a fixed point is reached. During this procedure all finite paths are visited implicitly. From the fixed point definition we also know, that for all infinite paths to v the generated data-flow information is covered by $l_v^{out,\text{MFP}}$. Further details on why the precision relation $l_v^{out,\text{MFP}} \sqsupseteq l_v^{out,\text{MOP}} \sqsupseteq l_v^{out,\text{IDEAL}}$ is true for all $v \in V$ can be found in [ALS+07, Chapter 9.3].

For monotonic frameworks, an MFP solution always exists according to the Knaster-Tarski-Fixed-Point Theorem [Tar55]. In general, the MOP solution is more precise than the MFP one, since in the MFP case we apply the meet operator

Algorithm 1 The generic data-flow analysis work-list algorithm.

```
1:  function WorkListAlgorithm($((\mathbb{L}, \sqsubseteq), F)$, $((V, E), v_0, l_0)$)
2:      worklist $\leftarrow v_0$
3:      for $v \in V$ do                                ▷ Initialization of the $l_v^{in}/l_v^{out}$ ...
4:          if $v = v_0$ then
5:              $l_v^{in} \leftarrow l_0, l_v^{out} \leftarrow \bot$                    ▷ ... for the start node ...
6:          else
7:              $l_v^{in} \leftarrow \bot, l_v^{out} \leftarrow \bot$                  ▷ ... and for all other nodes.
8:      while worklist $\neq \varnothing$ do                 ▷ Loop until a fixed point was found
9:          $v \leftarrow$ pop(worklist)
10:         $l_v^{in} \leftarrow \bigsqcup_{(u,v) \in E} l_u^{out}$
11:         $l_v^{tmp} \leftarrow f_v(l_v^{in})$                       ▷ Apply transfer function of node $v$
12:         if $l_v^{out} \neq l_v^{tmp}$ then                    ▷ If there were changes at node $v$ ...
13:             $l_v^{out} \leftarrow l_v^{tmp}$
14:             for $(v, w) \in E$ do
15:                 push(worklist, $w$)             ▷ ... propagate them to all successors
16:     return $\{v \rightarrow l_v^{in} \mid v \in V\}$
```

earlier than in the MOP case, namely after each node instead of only after each path. For distributive frameworks this does not make a difference (see Equation 2.4), therefore the MFP solution is equal to the MOP solution for distributive frameworks ("Interprocedural Coincidence Theorem" [KS92]). For computing the MFP solution, the *work-list algorithm* is a standard approach as shown in Algorithm 1.

The data-flow information is initialized in lines 3-7 and then propagated through the graph in lines 8-15. The nodes at which the data-flow information has not yet converged are kept in a work-list which is processed until all state information has converged. As mentioned, this convergence relies on the monotonicity of the height of the underlying lattice. For finite-height lattices, the convergence is guaranteed, since any data-flow item either reaches a fixed point other than $\top$, or reaches $\top$ in a finite number of steps which is a forced fixed point due to the monotonicity of the transfer functions. To speed up the convergence of the data-flow items we can sort the work list in topological order, ignoring back-edges in the CFG.

As an example for a lattice, consider the problem of determining whether a single register has a constant value at the individual CFG nodes. The Hasse diagram of the lattice for this problem is shown in Figure 2.2. It has a finite height of 2, though it has an infinite number of elements. The bottom element $\bot$ denotes the state "register was not initialized", the top state $\top$ means "register does not hold a constant value" and the middle row of elements represents actual constants. Whenever a block v loads a constant value into the register, l_v^{out} is set to this value. If multiple constants are merged in the meet operation, l_v^{in} is set to $\top$ in Algorithm 1. Further examples and an overview of possible lattices for program analysis can be found in [Cou01].

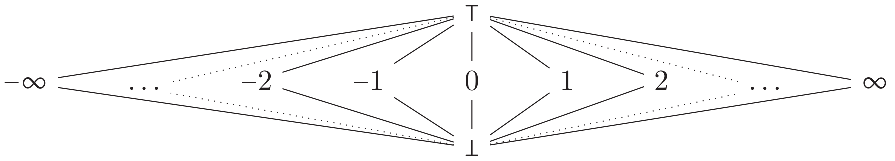


Figure 2.2: The lattice of integer constants.

Since the convergence of Algorithm 1 is not guaranteed for lattices with infinite height, such as intervals on integer numbers, *widening* is used to enforce the termination of the abstract interpretation in such lattices.

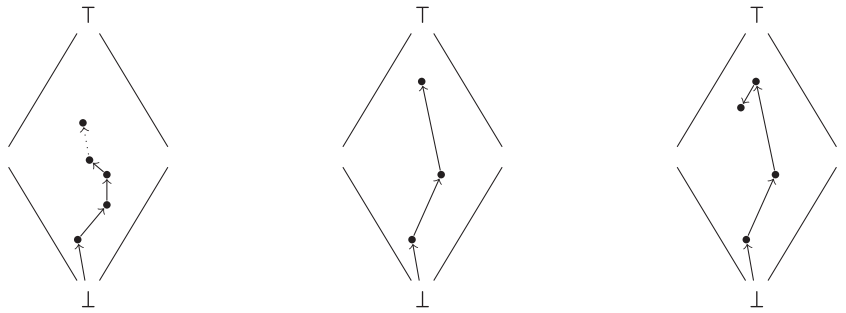
Definition 7. *A* widening operator Δ *is a unary function on* $\mathbb{L}$*, such that* $\forall l \in \mathbb{L} : l \sqsubseteq \Delta(l)$ *and for any sequence* $l_1, l_2, \ldots$ *with* $\forall i > 1 : l_i \sqsupset \Delta(l_{i-1})$ *the top element is reached in a finite number of steps, i.e.,* $l_n = \top$ *for some* $n \in \mathbb{N}$*.*

With a widening installed, line 11 in Algorithm 1 becomes

$$l_v^{tmp} \leftarrow \Delta\left(f_v\left(l_v^{in}\right)\right)$$

After the fixed point was found, it is possible to refine the results again by letting the main loop of Algorithm 1 iterate again for a user-defined number of times with the original definition of l_v^{tmp}, i.e., without the widening. This process may remove some of the overestimation induced by the widening and is called *narrowing.* In Figure 2.3 an example is given, how widening and narrowing affects the generated results. The figures show the Hasse diagram of the lattice and the evolution of a single data-flow value (one l_v^{out} for a fixed node v) during the runtime of the fixed point determination. Every arc corresponds to one iteration of the main loop from Algorithm 1 for v. The widening skips over many intermediate values by artificially coarsening the results of the individual steps. This speeds up the convergence (reduced number of arcs) but also leads a more imprecise result (higher in the Hasse diagram). The narrowing is able to re-gain some precision by applying the unmodified transfer functions to the fixed point values. It can lead to results which are at best as good as in the case without widening, but usually they are worse.

To further refine the results of the data-flow analysis, *path-awareness* can be introduced as shown in [HT98], which inflates the analysis lattice by multiplying it with a lattice that represents some of the edges that were visited during the generation of a data-flow information item. If that is done, data-flow items with different path expressions are not merged, which potentially increases the precision if the original lattice was not distributive. In addition, path-awareness can be used to rule out infeasible paths [NKJ10] if the edges in the path have contradictory guard conditions, thus moving the fixed-point result a step closer to the IDEAL solution sketched above. All of this comes at the price of increased analysis duration, since the size of the underlying lattice is multiplied by the number of possible edge



(a) Without widening **(b)** With widening **(c)** With widening and narrowing

Figure 2.3: Convergence behavior in a lattice. Solid and dotted arcs represent one or indefinitely many applications of the transfer function, respectively.

strings. Therefore, [NKJ10] try to limit the edge strings to *relevant edges* only, heuristically. Another approach to control the complexity of the path-awareness is to gradually increase the allowed length of the path expressions during the analysis as shown in [DDY06], where the authors use this method to analyze large real-world benchmarks. As a last option, [BSI+08] separate the detection of infeasible paths from the data-flow analysis itself and re-structure the program such that infeasible paths are avoided also in a non-path-aware analysis.

With the systematical description of a DFA framework coined above, it should not be surprising that DFA implementations for concrete problems can easily be generated from compact lattice and transfer function descriptions [SSB09], even for path-aware DFAs [HMM12].

2.2 WCET Analysis for Uninterrupted Single Tasks

In this section we will review the existing approaches to the determination of WCET estimates for a single program running without interruption on a single-core computer with no other active hardware components which could interfere with the program under analysis. This is the most basic of all WCET analysis scenarios, but it already offers a plethora of pitfalls and problems to solve [WEE+08]. In general, the challenges that WCET analysis is facing in this scenario fall into one of the following categories:

- **Unpredictable hardware components:** The WCET analysis has to follow all execution paths of the hardware to find the path with maximum execution time. Abstract interpretation can relieve this problem to some limited extent, but strong abstractions on the hardware state also lose much precision. Therefore, hardware features which increase the state space of the pipeline or memory hierarchy such as caches, superscalar and out-of-order execution,

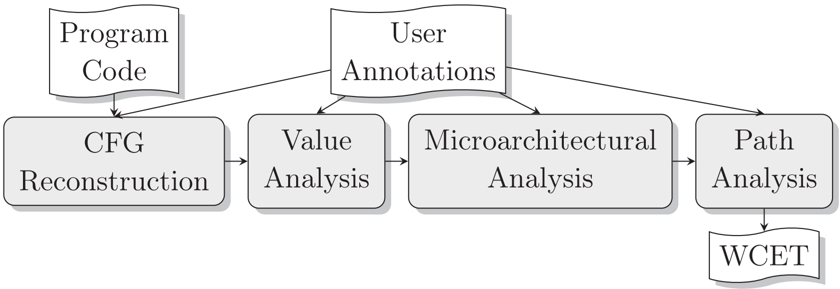


Figure 2.4: Structure of most static WCET analyzers.

fetch and commit queues and speculative execution complicate the WCET analysis [WGR+09].

- **Unpredictable software structure:** Usage of runtime-dynamic software constructs like function pointers, virtual inheritance and even dynamic memory allocation is hard to analyze statically. These can be circumvented to some extent by coding conventions such as MISRA-C [MIS13] which prohibit the usage of these features. Nevertheless even predictably-designed software exhibits an iteration structure of loops and recursions which may be not analyzable statically. Here, the user has to intervene in the worst case and needs to give hints to the WCET analyzer [KKP+11].

The following subsections will introduce the important branches of single-task WCET analysis, review real-life examples and case studies of their application and finally finish with an introduction to timing anomalies and compositionality, two concepts which are central to WCET analysis and frequently used in the following sections.

2.2.1 Static WCET Analysis

Static WCET analysis analyzes a program without executing any part of it, solely based on abstract models of the hardware and program semantics. Therefore, it provides stronger safety and coverage guarantees than measurement-based methods. The usual approach is to separate the analysis into multiple steps as shown in Figure 2.4.

A program, which, depending on the analyzer, may be given in binary or source-code form, is first fed into the *control-flow graph reconstruction* which extracts an interprocedural control-flow graph that reflects all possible transitions of control flow among instructions from the given program. In case of non-analyzable control-flow transitions, such as function pointers, the user might be required to specify their targets manually. After this step, the *value analysis* computes safe approximations of the register or variable values by means of AI on the generated CFG. The user can manually refine these results if he is not satisfied with their precision. The *microarchitectural analysis* uses the CFG and the value results to provide overestimates of the possible hardware states at each CFG node, again usually by means of AI. The user might have to specify machine details like the used clock frequency and the memory hierarchy specification for this step. Finally, the *path analysis* computes the WCET result. To achieve this, it must know upper bounds on the execution frequency of all cyclic paths, i.e., loops and recursions. For simple cases, the value analysis will deliver these bounds, whereas for complex loops the user has to provide a bound.

The prime example of an implementation of this scheme is the aiT WCET analyzer from AbsInt GmbH [FH04], which is the most well-known and industrially-used WCET analyzer available. The analysis that was implemented for this thesis inside the WCC [FL10] also follows this line of research, as well as other industrial and academic WCET projects like Bound-T [Tid04], the Open Timing Analysis Platform (OTAP) [HPP11], the OTAWA framework [Tea14], the CalcWCET167 tool [Kir12], SWEET [MRT14], Chronos [LLM+07] and TuBound [PSK08]. Apart from aiT, all of these tools were written mainly for research purposes and have experimental or simplified versions of individual analysis stages. Most importantly, only aiT includes an abstract processor model also for non-trivial target processors in the microarchitectural analysis. In Chapter 4 we will look at the individual analysis stages in more detail, where we also show their realization inside the WCC.

The analysis structure sketched in Figure 2.4 can be regarded as the most promising approach to WCET analysis. Nevertheless, it is not the only one. Model-checking has proven to be effective in the analysis of distributed real-time systems [AD90; BY04; FKP+07]. There also exist proposals to determine the WCET using model-checkers, which requires the whole architecture and the program under analysis to be modeled as a transition system amenable to classical model-checking theory. The main drawback of early approaches was the fact, that a dedicated model has to be created which reflects the timing of the program under analysis [CCM97]. Any error done in the manual modeling invalidates the results. More recent approaches incorporate program and hardware state into the model, e.g., using a generic model-checker like UPPAAL to solve the WCET problem [BC11]. The usage of a generic model-checker opens up the possibility to model arbitrarily complex systems. Unfortunately, due to the same reason, these approaches also lack the capabilities to form controlled abstractions of the resulting state space which leads to poor analysis times when compared to abstract interpretation-based designs [Wil04]. In experimental results from Gustavsson [GEL+10] runtimes of more than 30 hours are reported even for drastically simplified architectures. Similarly, the path analysis speed is far inferior with model checking than with classical methods based on *Integer Linear Program*s (ILPs) [HS09].

A third, but now outdated approach is the integration of microarchitectural analysis and path analysis into one single ILP. This was only attempted for simplified architectures, where only a cache was modeled, and already there major problems with the scalability of this method were found [LMW95; LMW96].

2.2.2 Parametric WCET analysis

One problem of the "classical" approach to WCET analysis as shown in the last subsection is, that the result is a single numeric value. For software which shows significant variation in its runtime, it would be better if the WCET analysis yielded a *formula* which describes the WCET depending on the program's input parameter values. This concept is called *parametric WCET analysis*. If the variables in the WCET formula are input values, we specifically call it *input-parametric*. According to an industrial case study from Gustafsson and Ermedahl [GE07] the lack of parametric WCET analysis tools is one major shortcoming of present WCET analyzers.

The main problem for input-parametric WCET analysis is the path analysis stage, which now must produce a WCET formula describing the longest path through the CFG depending on input values which may trigger different behaviors of loop and condition structures in the program under analysis. Methods for constructing such formulas are stepwise CFG reductions [AAN11], algebraic simplification of weighted path expressions [HPP12], solving symbolic ILP problems [AHL+08; BL08], custom graph-flow algorithms [BEL11] and symbolic evaluation [Bli02].

A less complex but also more restricted approach is the identification of *execution scenarios* prior to WCET analysis. These are often present in the software structure if the code is able to run in different input-defined *modes*. For each identified scenario a separate WCET analysis can be conducted with user annotations that force the analyzer to limit the examined program paths to those of the particular scenario [LPW09; MA11; HKB+14]. The generated WCET formula thus only assigns a single numeric WCET value to each scenario, which is identified by the inputs that trigger this scenario.

Apart from input-parametric WCET analysis, there have recently also been efforts to establish *architecture-parametric* WCET analysis [RD14]. In the latter case, the WCET formula is a function of architectural properties of the system under analysis. This mainly affects the microarchitectural analysis, which must now generate results for all possible target hardware configurations. Although architecture-parametric WCETs would be useful, input-parametric WCETs are a more urgent problem in practice [GE07].

2.2.3 Hybrid WCET analysis

As a counterpart to static WCET analysis, *dynamic WCET analysis* refers to end-to-end measurements of the program runtime on the actual hardware. These will not be safe unless all input parameter and initial system configuration combinations were exercised, which is infeasible for any real-world system.

Thus, to re-gain some confidence in the results supplied by dynamic WCET analysis, *hybrid WCET analysis* was proposed. It replaces the value and microarchitectural analysis from Figure 2.4 with a measurement of the runtime of each CFG node and then continues with the known path analysis methods from static WCET analysis to derive the WCET estimate. Since the measured CFG node runtimes are not guaranteed to be safe, this method is only suited for soft real-time tasks, where precision of the estimate is the main concern. An industrial analyzer which uses this technique is the *RapiTime* tool from Rapita Ltd. [Ltd14]. To increase the confidence in this method, test input generation heuristics like the *Balanced Path Generation* [BZT+11] have been proposed.

2.2.4 Early-Stage WCET analysis

Typically, WCET analysis can only be applied after the complete binary code of the program to be analyzed is available and the hardware platform on which it should run is known. This is needed for the WCET analysis to be safe, since all program execution paths on the platform must be considered hence both the program and the platform must be known. *Early-stage WCET analysis* abandons safeness in favor of getting early results, even for models of the systems which are not yet available in binary form. For these models, a safe estimation is hardly possible, instead an efficient, early feedback on the achievable WCET is requested. For this purpose, mostly regression with linear models is used [GAE+09], which are trained with WCET values of high-level programs [AEL+11]. The results are not safe, nor is there any known bound on the maximum deviation from the true WCET, but empirical results indicate that early-stage WCETs have a deviation of less than 20% compared to the final, safe WCET values.

2.2.5 Statistical WCET analysis

Since any computing hardware has some defined *mean time before failure*, *statistical WCET analysis* is an attempt to compute a probability distribution of execution times of a task on a given platform. If the probability for a violation of the task's deadline is at least as small as the probability of a hardware failure, the remaining failure probability can be acceptable [Höf12; KVA+13]. Still, many assumptions have to be verified before applying this concept, like the statistical independence of the probability distributions of software and hardware events [SLL+11]. In practice, proving this independence will be challenging and possibly lead to unsafe results if invalid assumptions are made.

A relatively new approach to this problem is *Probabilistic Timing Analysis (PTA)*, which tries to justify these independence claims by requiring a hardware where everything is randomized as far as possible, from the memory layout over microarchitectural aspects to the cache placement and replacement policies [CQV+13]. However, a recent publication [Rei14] shows that the handling of randomized caches in current PTA approaches is unsafe, and that deterministic replacement policies like LRU perform consistently better, even in the context of PTA. It is also shown that statistical independence of microarchitectural event distributions, which is the basis of PTA, was often wrongly assumed [Rei14]. Therefore, this analysis branch can only be considered highly experimental in its current state.

2.2.6 WCET-friendly Hardware Design

Even mainstream processor development has acknowledged the need for more predictable hardware, though only to a limited extent. As an example, the ARM Cortex-R series of processors [ARM14a] features local scratchpads for each core and a time-predictable interrupt handling. Nevertheless, it still contains a deep superscalar pipeline with speculative execution and instruction pre-fetching. Though these are not necessarily features which hamper the "observed predictability", i.e., the timing variation between multiple independent measurement runs, they still complicate a static WCET analysis by increasing the hardware state space considerably. The Infineon AURIX series [Inf14] shows similar properties from the analysis point of view, though with a shallower pipeline and a higher focus on safety and security. Lastly, the popular Kalray MPPA 256 manycore [KAL14] is an example for how power-awareness can also increase the predictability. To achieve power-efficiency while maintaining high performance it implements a VLIW architecture which is not only more power-efficient than dynamic out-of-order processors but also fundamentally easier to analyze.

In contrast to that, there exist multiple academic proposals for more predictable hardware. The most extreme of these are proposals to make the hardware capable of running in a *worst-case mode* [ORM+09] or to design hardware with constant-execution-time instructions such as the Java-Optimized Processor [SPP+10] or the PRET machine [LRB+12].

Other ideas include the instantiation of a dedicated DRAM-refresh software task to replace the unpredictable hardware DRAM refresh [BM11] and the adaption of the cache hierarchy [HPS12] or the load-store-buffer [MR12] to the needs of WCET analysis.

2.2.7 Experiences with Practical Application of WCET Analysis

Multiple case studies have been conducted on the usability of static WCET analysis in practice. These studies indicate that the tools are usable by non-experts [TSH+03] and that a maximum overestimation between 4% and 33% is achievable [GE07; CSB+10]. Compared to "traditional" methods which measure the runtime and add a safety margin, improvements of approximately 10% were observed [TSH+03]. Since static analyses must be able to safely bound all program loops, the user may have to provide such bounds if they cannot be found automatically. This is an issue especially for loops in operating system kernels, which are often data-dependent. The manual bounding of such loops is time-consuming and often requires assistance from the original developers of the respective binary code [GE07].

For tools which operate on the source-level, like SWEET, also the development toolchain poses problems since they need to integrate with it to find all source files which might be written in differing, non-compatible source-language dialects [LES+13]. Even worse, the source code might be missing altogether because a

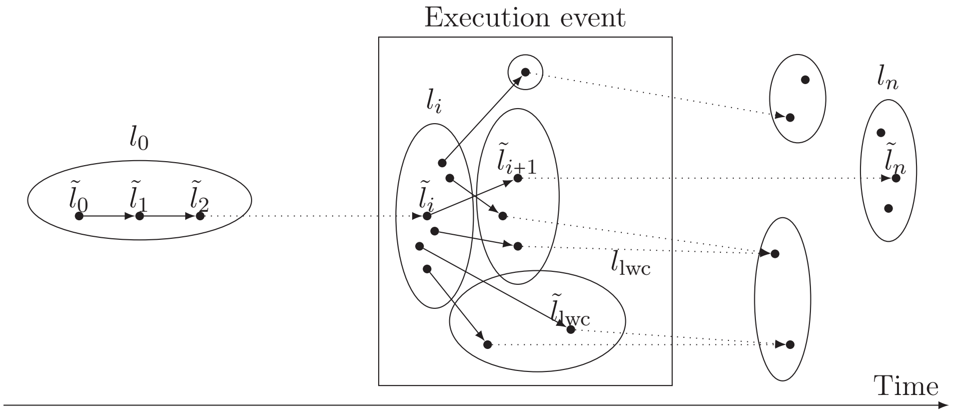


Figure 2.5: The execution paths of a program on a timing-anomalous system depicted by transitions between hardware states.

particular component is supplied by a subcontractor [LES+13]. This provides strong arguments in favor of binary-level analysis, where these problems do not exist.

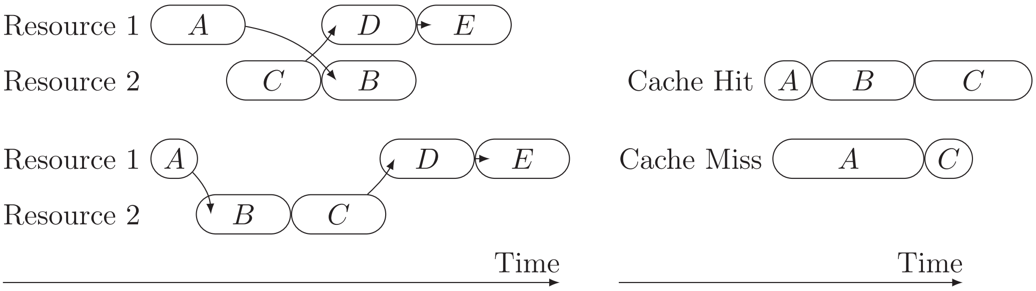
Finally, the analyzed software often operates in modes, therefore a WCET result which is parametric on the mode and input would be highly desirable [GE07]. This is supported by the finding, that the separation of multiple scenarios which use a common code base can reduce the resulting WCETs by up to 70% [HKB+14].

The *WCET tool challenge* was established to compare the results of different analysis tools [HGB+08]. Since the market is still very fragmented and it is already hard to find multiple tools which target the same hardware configuration, there are still no quantitative results comparing different analyzers.

The author is not aware of industrial case studies on hybrid, parametric, early-stage or statistical WCET analysis. The lack of such studies may also reflect the fact that static WCET analysis is the oldest variant and the tools produced in this area are more mature and ready for industrial use.

2.2.8 Timing Anomalies

Timing anomalies are a phenomenon which is observed on some architectures and which complicates static WCET analysis. During WCET analysis, we want to determine all possible *concrete hardware states (CHS)* for every node in the CFG to find the worst-case path through the program. An example for a worst-case execution sequence is given in Figure 2.5, where each dot is a CHS. The figure shows the worst-case execution of the program under analysis in the CHS sequence $(\tilde{l}_0, \tilde{l}_1, \tilde{l}_2, \ldots, \tilde{l}_i, \tilde{l}_{i+1}, \ldots, \tilde{l}_n)$. Of course, the machine works deterministically, therefore each CHS has only a single successor which can be determined using the $[\![\,]\!]_{conc}$ cycle-step semantics from Section 2.1. As we have already seen there, we need to introduce abstract semantics $[\![\,]\!]_{abs}$ which work on the *abstract hardware states (AHS)* to make the analysis decidable. Each AHS contains one or more CHS, or to be more



(a) Scheduling anomaly **(b)** Speculation anomaly

Figure 2.6: Examples of timing anomalies [RWT+06].

precise, the AHS and the CHS form a Galois connection as shown in Figure 2.1. In Figure 2.5, the ellipses represent AHS, thus the initial abstract state l_0 contains at least the concrete states $\tilde{l}_0$, $\tilde{l}_1$ and $\tilde{l}_2$. Since every AHS contains multiple CHS, the cycle step for an AHS is no longer deterministic. Depending on the contained concrete states we might end in one of multiple successor AHS as visible at state l_i. Concrete examples for sources of this kind of non-determinism are unknown addresses of memory accesses (all possible memory access targets must be considered in separate successor AHS) or unknown cache access behavior (cache hit and cache miss must be tracked independently).

At this point, it would make WCET analysis easier if is was guaranteed that only the local worst-case (the cache miss) may lead to the WCET of the program. If this were true, we could restrict our search to the local worst-case successor AHS and ignore all other possible successor states (e.g. the case of a cache hit). A *Timing Anomaly* (TA) occurs when this assumption is violated, e.g. when a successor state that does not represent a local worst-case behavior nevertheless exclusively leads to the global worst-case behavior. To define the notion "local worst-case", we need to define a local scope, i.e., the processing of some microarchitectural event such as a single cache miss or maybe a single instruction. In Figure 2.5, the rectangle shows such an event. For simplicity assume that a cache access is modeled. This scope contains all CHS which contain the processing of this cache access. The local worst-case is given by the state $\tilde{l}_{\text{lwc}}$ (AHS l_{lwc}), but it does not lead to the global WCET. Therefore, in such a case, the WCET analysis cannot assume that the only relevant successor of l_i is l_{lwc}, but instead it had to search every possible successor.

This definition was first given by Reineke et al. [RWT+06]. The authors also give concrete examples of timing anomalies, the two most well-known ones are depicted in Figure 2.6. In Figure 2.6a two possible execution paths are shown assuming an out-of-order processor with two execution units (resources), which are utilized by instructions A, B, C, D, and E. The arcs visualize the data or control-flow dependencies between the instructions. The execution time of instruction A is assumed to be unknown and variable, all other execution times are fixed. Contrary to naive assumption, a *short* runtime of A leads to the WCET for this sample program (as shown in the lower half of Figure 2.6a), not a longer one (as shown in the upper half). This is caused by the dynamic scheduling of the resources in the out-of-order core, i.e., since B becomes ready for execution earlier than C in the lower scenario, the execution order of B and C is reversed. Other known sources of timing anomalies are found in caches when they are combined with speculative execution and prefetching, a combination which is also often present in modern processors. Figure 2.6b shows an example for such a timing anomaly, where node A performs a cache access which might be a hit or a miss. In the case of a hit, speculative prefetching for block B begins and pollutes the cache until the hardware recognizes that the speculation was wrong and execution of B is aborted. Due to the cache pollution, the runtime of C is increased, such that this case is the global worst-case even though the cache hit was the local best-case. In the event of a cache-miss, the miss takes so long that the branch condition is fully evaluated once the miss was processed, therefore no speculation and cache pollution takes place here.

Timing anomalies can be further subdivided into *constant-bounded anomalies* and *domino effects* [RS09]. A system suffers constant-bounded anomalies if for any two concrete hardware states $l_i, l_j \in L$

$$t_{\text{term}}\left(l_i\right) - t_{\text{term}}\left(l_j\right) \leq \Delta\left(l_i, l_j\right) \tag{2.8}$$

holds, where $\Delta : L \times L \rightarrow \mathbb{N}$ and $t_{\text{term}} : L \mapsto \mathbb{N}$ is a function which returns the number of cycles until the executed program is completed, counted from the given input state on. A system suffers from domino effects if such a constant bound cannot be found, i.e., if there are instruction sequences for which the runtime starting in state l_i is a multiple of the runtime starting in state l_j. In theory, both concepts can be used to prune parts of the analysis search space even in systems with timing anomalies [RS09], but this is only possible when microarchitectural analysis and path analysis are combined which is intentionally not the case in current static analyzers to limit the analysis complexity (compare Figure 2.4).

Up to now, there has been only one publication on an approach to prove the *absence* of TAs in a given architecture specification [EPB+06]. Unfortunately, this approach has never been tested for real architectures. In contrast, there exist a number of *existence proofs* by example. The following combinations of hardware features are known to exhibit timing anomalies:

- out-of-order, superscalar processors with execution units with non-overlapping functionality [RWT+06],
- systems with caches, prefetching and speculative execution [RWT+06],
- pseudo-LRU caches [RWT+06],
- multi-cores with a round-robin arbitrated bus and private L1 caches [SHK14],
- MRU caches [Geb10] and
- partial filling of cache lines (cache streaming) [Geb10].

Related to timing anomalies are *parallel timing anomalies* which are defined in [KKP09] as analysis errors that are introduced by a decomposition of the microarchitectural analysis into multiple phases which analyze subsets of the hardware in separation. In practice, this is less relevant since the state of hardware components is highly interdependent. A separation of cache, bus and pipeline analysis is usually only possible at the expense of a decreased analysis precision.

2.2.9 Compositionality in WCET Analysis

To account for "extra delays" due to interference by other tasks, analysis approaches often compute a single-task WCET and then add the delay that occurs due to phenomena which were not considered in the single-task WCET computation. Many analyses in Section 2.3 and Section 2.4 will follow this scheme. All of these approaches require *timing compositionality*, i.e., the notion that the WCET of a task running on a system can be safely derived from separate contributions of individual hardware components to the WCET.

In systems with timing anomalies, we have already seen that the occurrence of a timing anomaly depends on an interaction between multiple hardware components. If each component is analyzed in separation to achieve a timing-compositional WCET, the occurrence of TAs can never be excluded and thus must be assumed wherever possible. This will make the timing-compositional WCET for TA-prone systems highly pessimistic. Due to their more limited impact on the WCET, the case of constant-bounded TAs according to Equation 2.8 introduces less pessimism than the case of domino effects. On the other hand, WCETs for timing-anomaly-free systems are always timing-compositional. Here we know that when we externally increase the duration of a hardware event e by a duration d, e either already was the local worst-case and thus the WCET increases by d, or e has not been the local worst-case and the WCET increases by at most d, depending on whether e now becomes local worst-case or not. This classification has first been introduced by Wilhelm et al. in [WGR+09] and has led to the categories

- *fully timing-compositional* (no timing anomalies),
- *constant-bounded timing-compositional* (only constant-bounded anomalies) and
- *not timing-compositional* (domino effects).

Unfortunately, this well-established nomenclature is a bit misleading, since also for domino-effect-systems, timing-compositionality is achievable at the expense of increased overestimation as mentioned above. Therefore, calling these systems "not timing-compositional" is not fully appropriate. Recently, Hahn, Reineke and Wilhelm further elaborated the definition to the following form [HRW13]:

Definition 8. *Let C be a system and $(C_i)_{i\in\{1,\dots,n\}}$ its components with associated state spaces S and $(S_i)_{i\in\{1,\dots,n\}}$. Furthermore, let the timing contributions $(tc_i : S_i \mapsto \mathbb{N}_0)_{i\in\{1,\dots,n\}}$ together with state abstraction functions $(a_i : S \mapsto S_i)_{i\in\{1,\dots,n\}}$ and combination operator $\oplus : {\mathbb{N}_0}^n \mapsto \mathbb{N}$ be a decomposition of the system's timing*

$t_{term} : S \mapsto \mathbb{N}_0$. *We call the decomposition* (μ, α)*-timing compositional where* $\mu \in \mathbb{R}_{\geq 1}$, $\alpha \in \mathbb{R}_0^+$ *if and only if*

$$\forall s \in S : t_{term}(s) \leq \bigoplus_{i=0}^{n} tc_i\left(a_i\left(s\right)\right) \leq \mu \cdot t_{term}\left(s\right) + \alpha \tag{2.9}$$

This definition is more akin to the original, intuitive understanding of timing compositionality than the definitions from [WGR+09]. The leftmost inequality in Equation 2.9 states that the timing decomposition must be *safe*, i.e., greater than the most precise timing value $t_{\text{term}}(s)$ whereas the rightmost inequality imposes a bound on the precision loss due to the decomposition. The combination function $\oplus$ will be the addition operator in almost all cases.

From the point of view of an analysis which requires timing-compositional WCET results, "constant-bounded timing-compositional" now corresponds to a $(1, \alpha)$-decomposition and "fully timing-compositional" denotes a $(1, 0)$-decomposition. Still, an implicit connection to the definitions from [WGR+09] exists, since a $(1, 0)$-decomposition will be easy to find for a timing-anomaly-free architecture, whereas for architectures with TAs, a $(1, 0)$-decomposition for more than one component $(n > 1)$ will only be achievable at the cost of degraded analysis precision. To the best of the author's knowledge, no publications exist which actually quantify the achievable precision of timing-compositional WCETs for systems with and without TAs.

In the following, we will not use the nomenclature from [WGR+09], but explicitly state to which TAs a system is susceptible. When analyses require timing compositionality in the (intuitive) sense of [HRW13], we will use (μ, α)-timing compositionality to denote this.

A different problem which is also termed "timing-compositionality" in some publications is the construction of software from multiple, independent and possibly precompiled software components. For WCET analysis this is a problem, since the intention of component-based design is, that the developer can use the components as black-boxes fulfilling a particular task. However, during WCET analysis the microarchitectural behavior and the worst-case iteration count of loops in the component must be known. This leads to the problem that the person conducting the WCET analysis has to manually bound the loops in the code of the component as already mentioned in Section 2.2.7, which is often highly time-consuming. To resolve this, a standardized summary format for binaries was proposed which sums up the effects that the binary code has for given data-flow analyses [BCM09]. This would enable the inclusion of the semantics of black-box components into a bigger data-flow analysis, e.g. the microarchitectural analysis from Figure 2.4. Still, the approach requires a major standardization effort, since every data-flow analysis uses different domains and formats. For the path analysis (cf. Figure 2.4), a constraint-logic-programming-based approach was presented in [Mar10] which requires parametric WCETs for the components.

2.3 Timing Analysis of Sequential Multi-Task Systems

When multiple real-time tasks are executed on one core, usually a *Real-Time Operating System* (RTOS) is employed to schedule and decouple the tasks. For WCET analysis this poses the problems that

- single-task execution may be *preempted* by interrupts and successively running higher-priority tasks and
- tasks will be using *system calls* to communicate with the RTOS, thus the timing of these system calls must be analyzable.

Traditional system calls in addition are a separate kind of transfer of control that is beyond the capabilities of conventional WCET analyzers as introduced in Section 2.2. This problem is alleviated by the fact that many RTOSes give up the traditional strong boundary between the OS and user code for performance reasons. In these cases, the RTOS and the user code are compiled into a single binary and system calls collapse to normal function calls which are again amenable to standard WCET analysis. Nevertheless, the two main issues as listed above remain. In the following subsections we will review the state of research concerning these points.

2.3.1 Accounting for the Timing Behavior of System Calls

One of the most important standards for RTOS design in the automotive domain is AUTOSAR [AUT09]. Even though this standard is already tailored towards real-time automotive systems, it is still not trivial to achieve a time-predictable and efficient implementation of inter-task communication system calls as shown in [FDG+09]. The authors of this work do not consider static bounds on the duration of communication calls, but instead they examine the *time predictability* of the calls from a measurement point of view. In fact, a time-predictable implementation of system calls not only helps the WCET analysis, but also in additional measurements of the timing on an example system.

For a real-time operating system which is tailored towards WCET analysis, it is often possible to determine static bounds on the runtime of the individual system calls as shown in the MERASA project [WGK+10]. Nevertheless, for existing OSes, which were not designed with analyzability in mind, this is a considerable effort whose success has proven to be limited [LGZ+09]. The main problems which are discussed by Lv et al. are lack of

- parametric WCET tools since system calls show highly variable behavior,
- integration between the WCET analysis of the RTOS and the analysis of the user tasks to achieve better precision,
- integration of schedulability analysis and WCET analysis to reach more precise results and
- automation of the WCET analysis, since many user interactions are needed for complex code.

In contrast to timing compositionality which denotes that a WCET value can be broken down into multiple contributions from different hardware components (cf. Section 2.2.9), *timing composability* describes that the timing behavior of a piece of software is independent of other software running in parallel or in sequential alternation on the same core. Timing composability is a highly desirable feature for RTOS implementations, since only then different tasks can be analyzed in separation from each other before they are integrated into the system. A possible implementation of a timing-composable operating system based on the ARINC standard is given by Baldovin, Mezetti and Vardagena in [BMV12], where also neccessary criteria to achieve timing composability are defined.

Nevertheless, in summary there are few approaches which construct dedicated, WCET-analyzable operating systems but for standard operating systems, even for standard RTOSes, WCET analysis is highly complicated due to the points identified by Lv et al. [LGZ+09] as mentioned above.

2.3.2 Accounting for Task Interaction Impacts on the WCET

If the scheduler can decide to preempt a real-time task, its WCET alone is no longer sufficient when reasoning about whether it will meet its deadline or not. In this case, the *Worst-Case Response Time* (WCRT) must be computed, which is the WCET plus the delay introduced by preemptions of all other tasks. For the case of fixed-priority scheduling it has been shown that WCRT values can be computed by solving a set of recurrence equations [JP86]. The original formulation assumed static WCET values, which is not true for modern hardware, due to the fact that pipelines and caches use part of the execution history to speed up future calculations. To be precise, a WCET analysis has to consider these history effects in the microarchitectural analysis. Each preemption invalidates the results of a single-task WCET analysis, since when the preempted task (the *preemptee*) is resumed

- the pipeline must be reloaded with instructions from this task, and
- the caches and all cache-like structures as, e.g., branch prediction buffers have been potentially modified by the *preempter*.

The former point is a small, local effect which can be bounded by a constant number of cycles, whereas the latter is a major issue since a modified cache state can lead to additional delays throughout all following program parts. This non-local effect is called the *Cache-Related Preemption Delay* (CRPD). Since the CRPD is dependent on the number of preemptions, which in turn is dependent on the WCRT, there is a cyclic dependency between CRPD and WCRT determination.

Therefore, Lee et al. combined the WCRT computation from [JP86] with an integrated determination of the CRPD [LHS+98]. They introduced the notion of *Useful Cache Blocks* (UCB) which are those memory blocks of the preemptee that may be re-used. Only if one of these blocks is evicted from the cache by a preempter, the timing of the preemptee will be influenced. By providing an estimate of the number of UCBs, they are able to bound the CRPD and therefore the WCRT of the tasks. To make this estimate more precise, the cache behavior of the preempter can also be taken into account in the form of an estimation on its number of *Evicting Cache Blocks* (ECB) [TD00], i.e. those blocks that are touched by the preempter. The intersection of UCBs and ECBs provides a more precise bound on the number of cache misses due to a preemption than obtainable with the UCBs alone.

Whereas Lee et al. [LHS+98] use the UCBs as a single numeric value specifying the number of such blocks, Negi, Mitra and Roychoudhury propose to work on the level of abstract cache states of preempter and preemptee directly, instead of abstracting both to a simplified numeric value [NMR03]. This allows for a higher precision but also incurs an exponentially increased analysis effort.

Altmeyer and Burguière [AB09] develop the UCB concept further to also account for *Definitely Cached Useful Cache Blocks* (DC-UCB). The DC-UCBs are a subset of the UCB which contains all blocks that *must* be cached. With this additional concept they can avoid some overestimation in the CRPD analysis by identifying where DC-UCBs can be used to eliminate the double accounting of a single block eviction.

Altmeyer, Maiza and Reineke further refined the CRPD for set-associative caches by noting that even if some UCBs and ECBs collide, the UCBs are only evicted if the cache set in which they are placed cannot hold all ECBs. Based on the abstract cache block age information, they determine a *resilience value* [AMR10] per cache block at each program point which determines how many ECBs this block can take without being evicted. Finally, the same authors show that FIFO and Pseudo-LRU (PLRU) replacement strategies cannot be analyzed with the known UCB/ECB procedure, but they establish the concept of *relative competitiveness* [BRA09] which denotes that a CRPD and WCET estimate for FIFO and PLRU can be obtained by analyzing an appropriately shrinked LRU cache. It turns out that the shrinking needs to be more aggressive for FIFO to make this work, than for PLRU.

Kleinsorge, Falk and Marwedel [KFM11] proposed a method to compute UCB, ECB and resilience information in a single analysis pass with a full cache state representation, similar to [NMR03] which further increases the precision while removing the analysis overhead for resilience computation.

Apart from implicit influence via the CRPD, tasks also have explicit influence on each others' timing via the invocation of potentially blocking communication system calls. The bounding of end-to-end message delivery delays has not been solved in general, but for the special case of tasks, which only send messages when they *terminate*, solutions exist [TBW95].

Finally, interactions on global variables may also have timing effects. Research on this topic is still very limited, but data-flow analysis on multi-task systems could potentially be a solution. A recent publication from Mittermayr and Blieberger [MB11] demonstrates how the potential interleavings of the threads' instructions can be efficiently explored when exploiting the synchronization structure of programs. As a side effect, their framework can also be used to bound the synchronization delay in a multi-task system with lock-protected shared resources.

In summary, the determination of implicit interactions like the CRPD is well-understood for at least the case of fixed-priority scheduling and LRU caches. The overestimation for these effects has been considerably lowered by the refinements mentioned above. Still, it is worthwhile to mention that *all* of the above approaches require (μ, α)-timing-compositional architectures to work, since an extra delay is added to the single-task WCET. This means that potentially they are only applicable to timing-anomaly-free architectures without loss of precision. Bounds on the timing behavior of explicit interactions between the tasks are still an active field of research, where no standard solution has been established.

2.3.3 Schedulability of Multi-Task Systems with Given WCETs

Schedulability is a synonym for the existence of a schedule under which each task from a given task set meets its deadline. If the WCRTs of the tasks have been computed for some type of schedule, testing for schedulability is trivial, since only the WCRTs and the deadlines need to be compared.

If this is not possible, e.g., because no WCRT algorithm is known for the considered schedule, the *Real-Time Calculus* (RTC) offers possibilities to model a system as a network of communicating components whose interaction is described by *event arrival curves* and *service curves*. Finally, these can be used to analyze the schedulability of the described task set [TCN00; PC07]. This line of research has led to a number of software tools which support the analysis of a real-time system with the help of the RTC, as, e.g., MAST [HGG+01] and SymTA/S [HHJ+05]. Similar to the WCRT/CRPD analysis mentioned above, the RTC-based modeling methods require (μ, α)-timing-compositionality with the mentioned drawbacks.

As an alternative to RTC-based methods, real-time schedulability theory can be used which provides necessary and sufficient conditions which can show that a schedule must exist or may not exist [Mar11, Chapter 6.2]. Unfortunately, these static conditions almost always assume a fixed WCET per task, which makes it impossible to precisely account for the CRPD.

2.4 Timing Analysis of Parallel Multi-Task Systems

Since approximately 2005 the hardware market has inevitably shifted towards multi-core systems [Sut12], also in the embedded domain [Fre09]. Therefore, interactions of tasks which execute in sequential alternation, possibly preempting each other and possibly in interaction with an RTOS are no longer the only complication for WCET analysis. On modern hardware, it also has to account for possible timing effects of tasks that execute concurrently.

Since this is the main topic of this thesis, we will only provide some general ideas on how parallel timing interaction can be handled. In Chapter 5 we will then examine the specific problems of WCET analysis for parallel tasks in more detail. We will only analyze the behavior of a given parallel system – the creation of such a system, whether by explicitly parallel programming, by generic parallelization techniques [Mid12] or by multi-criteria-aware embedded parallelization [CEN+13] is a topic of its own.

In the following discussion, we distinguish between closely-coupled and loosely-coupled parallel systems, e.g., multi-cores and distributed systems, because the granularity of the timing effects which are to be analyzed differs in these two cases.

2.4.1 Multi-Core Systems

An example for the acuteness of the shift to multi-cores in industrial practice is a growing number of publications on how to transfer the existing AUTOSAR standard for automotive software to multi-cores [AUT09]. New problems that arise in this field are

- how to map tasks to cores while controlling the communication overhead and fulfilling scheduling constraints [GHK+11],
- how to extend the *priority ceiling protocol* [Mar11], which guarantees the absence of priority inversion, to multi-cores [KYM+09], building on previous work on multi-processor priority ceiling [RSL88; CT94],
- how to support migration of legacy code to multi-cores [JMR10].

Schneider, Bohn and Rößger [JMR10] classify dependencies between tasks into the categories “none”, “mutual exclusion”, “precedence” and “temporal distance”, where the latter denotes a precedence constraint with the additional requirement that a minimum amount of time passes between the termination of the first task and the starting of the second. They emphasize the importance of tools to visualize execution and scheduling scenarios by which the developer can verify that these dependencies are met. WCET analysis can be one of those tools, since it also delivers a value analysis which [JMR10] names as an important factor to determine (possibly implicit) dependencies between legacy AUTOSAR tasks. It is also shown that real-time software realizes the named dependencies not only via classical synchronization management but also via interrupt enabling/disabling and task priority levels. From the perspective of WCET analysis this is favorable, since the timing of explicit synchronization primitives is often hard if not impossible to determine (cf. Section 2.3.2).

Since the timing of the components in a multi-core system is tightly coupled, these systems are more amenable to an integrated analysis, which tries to analyze the state of all hardware components in an integrated way. Such approaches have been made, based on the classical static WCET analysis framework from Figure 2.4 [CKR+12; KFM+14], based on model-checking [GEL+10] and on a combination of both [LYG+10]. We will discuss these approaches in more detail in Chapter 5.

Although real-time calculus-based methods have also been proposed for the analysis of multi-core architectural effects [SE09], these methods are often too coarse-grained for such an application, since they require (μ, α)-timing-compositional timing analyses for the pipeline and the caches. In a recent publication, Shah, Huang and Knoll show that such analyses will be highly pessimistic for multi-cores, since timing anomalies already occur in multi-cores with L1 caches and a round-robin bus [SHK14].

Difficulties exist not only for WCET analysis but also for schedulability tests. Extensions of classical schedulability theory towards multi-cores was attempted by Li et al. who developed a schedulability test for *global earliest deadline first (GEDF)* scheduling on periodic tasks [LAL+13]. Nemati and Nolte derived a schedulability test for clustered fixed-priority-scheduled parallel task sets [NN13] with sporadic tasks. A constructive schedulability proof for periodic tasks was given by Moir and Ramamurthy [MR99], who show that a valid schedule can be found in polynomial time iff such a schedule exists.

Some scheduling approaches also try to explicitly take hardware capabilities into account as, e.g., the conflict behavior of the tasks in the shared L2-cache [ACD06] or the distribution of memory accesses among local and remote memories [CCK+13]. These approaches are limited by the fact that they consider complex task activation schemes where a precise tracking of hardware states as done in a WCET analysis is infeasible due to the number of possible execution paths and task preemptions. Therefore, they require a rather simple parametric WCET model, in which the influence of the scheduling decisions on the WCET can be quickly determined.

2.4.2 Distributed Systems

In contrast to multi-cores, a network of processing nodes, also called a *distributed system*, is much more loosely coupled. Events at single nodes may trigger messages to other nodes which may start some kind of processing chain. Timing analysis of distributed systems is mostly concerned with bounding the frequency and duration of the messages and events in the system. Since the frequencies are generally much lower than, e.g., the frequency of shared memory accesses in a multi-core, a higher overestimation is acceptable. In this context, the Real-Time Calculus [Thi05] is usually used to analyze the system behavior. An advantage of the high level of abstraction that it provides is, that complex task activation schemes can also be modeled easily [HT07]. In addition, there exist attempts to couple the RTC with timed automata [LPT09] to reduce the overestimation in the analysis.

A seminal example of how to make the behavior of individual network components predictable enough such that methods like the RTC can reasonably well approximate them is the time-triggered architecture developed over multiple decades by Kopetz et al. [KB03]. It is based on the idea that time-triggered operation of systems provides the most predictable timing behavior and shows constructively that such an architecture is feasible in practice. We will build upon this work in Section 5.4 when building a WCET analysis for time-triggered multi-core resources.

Apart from the RTC-based methods, there are also analyses which integrate the message runtime determination with classical WCRT computation for the case of CAN [DBB+07] and FlexRay buses [PPE+08]. These analyses were tested in practice to determine, e.g., the end-to-end delay of messages in an AUTOSAR environment [LBR10].

Chapter 3

WCC Framework

Contents

The *WCET-aware C Compiler* (WCC) is the basis for the implementations of the WCET analyses developed in this thesis. The rationale behind its design [FLT06; FL10] is the integration of a WCET analyzer with a C compiler to give source and machine code optimizations the opportunity to evaluate the effect of their transformations on the WCET. Also, the WCET is broken down into the contributions of the single basic blocks of the program to make it possible to focus specifically on the optimization of the *Worst-Case Execution Path* (WCEP).

It has been shown that applying an optimization aggressively wherever possible may as well be *adverse* to the WCET of the program [ZCS03]. Similarly, WCET, *Average-Case Execution Time* (ACET) and code size can be conflicting goals [LPF+10; PKF+11]. Therefore, a prediction of the optimization effect in terms of the considered target function has to be achieved to guide the compiler optimization.

Since the WCC was originally intended as a drop-in replacement for a classical compiler, there was little to no multi-task and multi-core support. This support is needed at least in the WCET analysis to be able to account for interference by other tasks and cores. Originally, the WCC exclusively used the analyzer aiT [Abs14a], since this is a stable and industrially-proven WCET analysis solution. Unfortunately, it is only applicable to single-task WCET analysis, and since it is a commercial product, modifications are also not possible. Therefore, to have a flexible experimentation platform and since many components required for a WCET analysis were already present, a WCC-internal WCET analysis was devised in the course of this thesis. Chapters 4 and 5 will introduce this analyzer and its multi-core-specific extensions. Finally, the WCC was extended to also handle binary input files, which allows for a limited degree of modularity in the WCET analysis.

3.1 Related Work

Apart from WCC there are several other projects which have made attempts to integrate a WCET analyzer with a compiler. The first of these approaches dates back to 1996 and consisted of a combination of the commercial IAR compiler with multiple, now outdated WCET computation methods [Bor96]. The emphasis was put on the representation and management of flow facts and the project was not pursued long enough to achieve a working implementation.

Kirner and Puschner have first established the idea that *flow facts* which are needed for WCET analysis should automatically be updated when an optimizing compiler transforms the control-flow of the program [KP01]. They built a prototype based on the well-known GCC, which was later incorporated into the TuBound [PSK08] WCET analysis framework. For the actual WCET analysis, TuBound uses a previously developed WCET analyzer for the Infineon C167 processor (CalcWCET167 [Kir12]). Still, the generated WCET information is not made available to optimizations here, the purpose of the modified compiler is to transform the user-specified high-level flow facts to valid machine-level flow facts.

The VISTA framework [ZKW+04] was designed for the optimization of WCETs. Unlike WCC, however, it is targeted at *interactive* optimization guided by the user and its timing model is restricted to (μ, α)-timing-compositional architectures.

A recent project that bears many similarities with WCC is the *Open Timing Analysis Platform (OTAP)* [HPP11], which also tries to integrate aiT with a compiler, in this case the LLVM compiler framework, for the purpose of WCET-guided optimization and general experiments with WCET analysis. The project also uses the SWEET [MRT14] tool to obtain highly precise flow facts. Lately, the OTAP compiler has seemingly merged into the T-CREST project infrastructure [PKH+12; PHP14] which studies predictable multi-core hardware. T-CREST in addition pursues the idea of *single-path* code transformations which generate a branch-free (not loop-free) sequence of instructions from arbitrary code with the help of predicated execution.

The fact that all of the mentioned projects including the WCC are focusing on C as the prime source language, can be explained with C's popularity among real-time system developers. C++ in contrast holds more sources of implicit unpredictability like virtual methods, virtual inheritance and dynamic casts. Though there exist methods like, e.g., a real-time capable dynamic cast implementation [DMS08], these are rather limited and make the code generation and WCET analysis even more complex. Therefore, the restriction to C encourages the development of predictable code to some extent.

Finally, another big concern in the development of safety-critical real-time systems is that tools which are used for the development must usually undergo some certification process or must even better be formally verified to work correctly. However, formal verification of a compiler even for subsets of the C language consumes

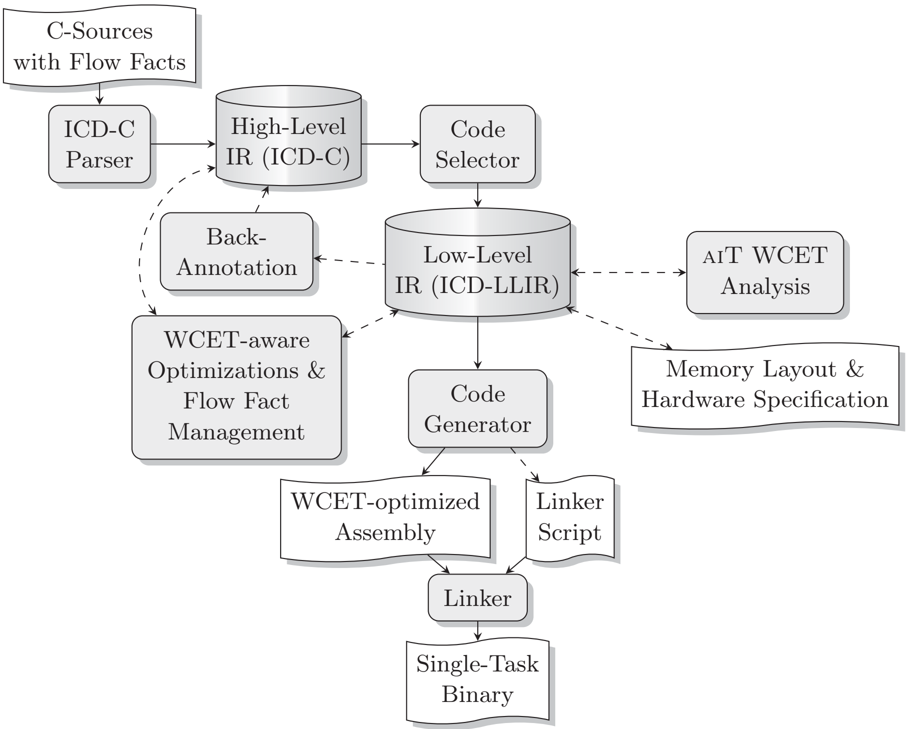


Figure 3.1: Previous structure of the WCC compiler [FL10].

multiple man-years of work as shown by the work of Leroy [Ler09]. Therefore, this certification effort is simply infeasible for WCC at its current stage of development.

3.2 Compiler Phases

The original structure of the WCC is shown in Figure 3.1. It follows the phase model of a general-purpose compiler [ALS+07], which are shown by the solid edges in Figure 3.1. The dashed edges connect components which are only needed in the context of the analysis and optimization of the WCET. First, the C source files which are annotated with flow fact pragmas are read and transformed to a *High-Level Intermediate Representation* (HLIR) called *ICD-C*. A *code selector* then lowers the HLIR to the *Low-Level Intermediate Representation* (LLIR). The LLIR is a direct representation of assembly code or relocatable object code. Together with the memory layout and hardware specifications it can be used by aiT to determine the WCET of the single task, and the WCETs and *Worst-Case Execution Counts* (WCECs) of the individual basic blocks and of functions inside the tasks. Through the usage of a *back-annotation* [FL10] which transforms the low-level basic block WCET contributions back to the high-level basic blocks, WCET-aware optimizations are enabled on both the HLIR and the LLIR. An overview of the developed WCET-aware single-task optimizations can be found in [LM10]. Unfortunately, the mapping of low-level basic blocks to high-level ones is an $n : m$-mapping in general. The back-annotation is still possible, because each component which modifies the basic block structure of either representation as, e.g., the code selector and many optimizations announce their changes to the back-annotation which then updates its internal data structures accordingly. A similar problem is the required updating of flow facts after optimizations [KPP10], which is also handled in WCC through the communication of all relevant changes to the *flow fact management module*, which then updates the affected flow facts.

The original compiler was designed to translate the source files of a single task into a separate binary which can be loaded by a surrounding operating system. However, in the case of real-time operating systems, the OS binary is often linked together with the tasks into a single binary program, or even compiled together with the task code and based on parameters depending on the tasks. Most OSEK-compliant RTOSes like, e.g., ERIKA Enterprise [Evi14] and FreeOSEK [Ope14] follow this approach. Therefore, the WCC was extended to support the analysis of multiple tasks inside the same binary (so-called “entrypoints”, see Section 3.3) and to the compilation of multiple binaries in parallel, where each binary is loaded onto a particular core of a multi-core system.

The resulting structure is shown in Figure 3.2. The vertical flow corresponds to the original single-task phases, where all core-binaries are finally packed together with a boot loader into a multi-core system ROM. Also, since parts of the task code may only be given in assembly or precompiled form, parsers for these two types of input files were developed. The result is fed directly into the LLIR of the task. Most importantly, a new WCET analysis was developed with can exploit the knowledge about all tasks and cores in the system to achieve precise multi-core WCET estimations. The elements in Figure 3.2 which were developed in the course of this thesis are drawn with black font, whereas those which preexisted are drawn in gray. Since a system description now comprises multiple source files per core and possibly the specification of the task entrypoints in these sources, a *task system description file* can be used as an input for the compiler. This task system description file simply contains references to those input files drawn with dotted border in Figure 3.2 and in addition it may specify new entry points and/or properties of entry points.

WCC is mainly written in C++ and comprises approximately $340,000$ *Lines Of Code* (LOC), counted without comments and blanks plus some fraction of assembly startup code files and bash scripts. The analyses presented in Chapter 4 and Chapter 5 account for $32,000$ LOC, whereas the optimizations from Chapter 6 are formulated in $10,000$ LOC. Finally, $10,000$ LOC were needed to implement the parsing of binary inputs (Section 3.5) and $39,000$ LOC to implement the configurable simulator platform (Section 3.4). In total, the extensions described in this thesis contribute approximately 27% of WCC’s current code size.

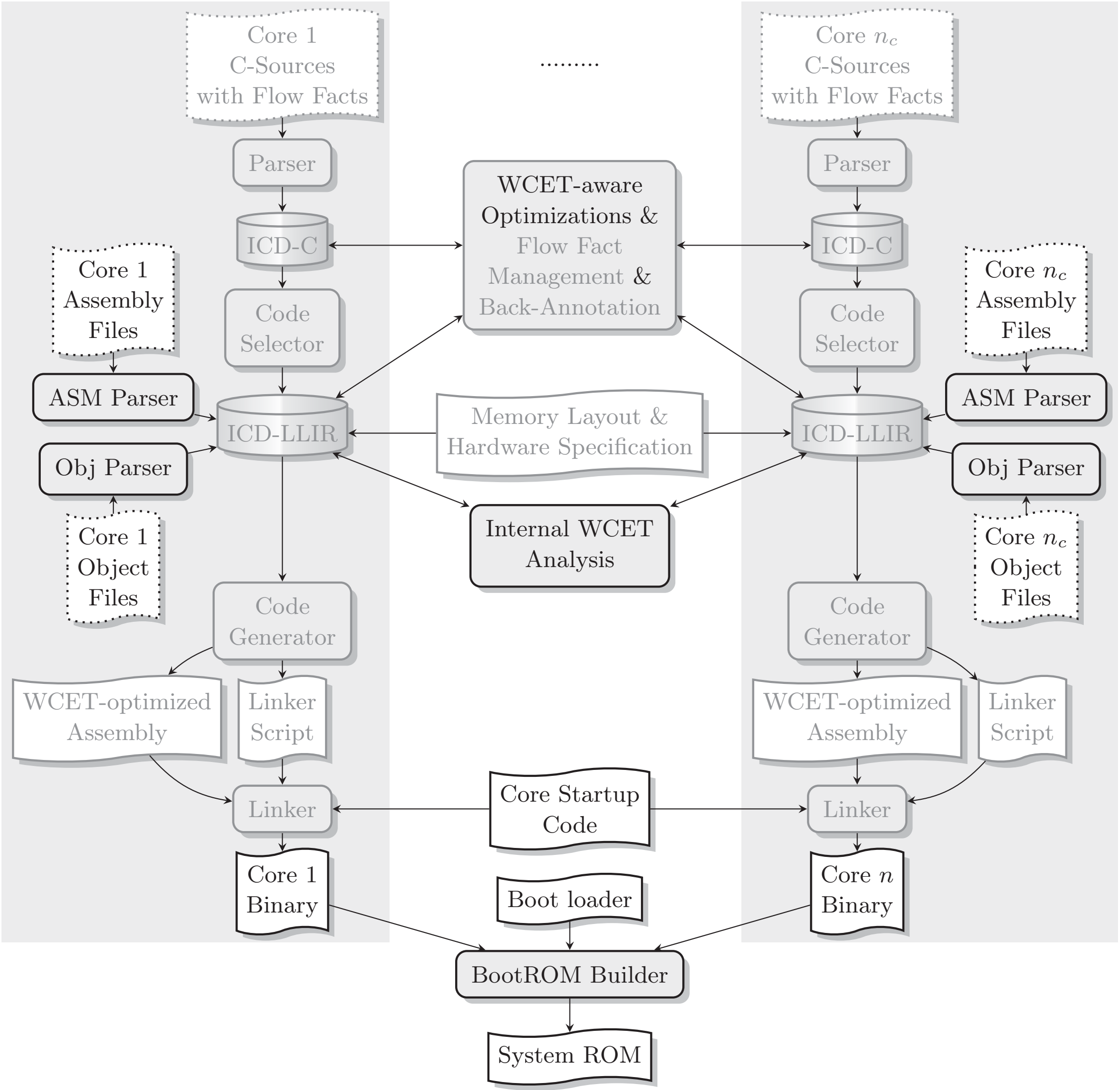


Figure 3.2: Structure of the WCC for multi-core compilation and analysis.

The next three sections will shortly introduce the handling of flow facts inside the WCC, the multi-core target platform and the extension to assembly and binary input files as shown in Figure 3.2.

3.3 Flow Fact Management

As already mentioned in Section 2.2.1, static WCET analysis as done inside of the WCC requires user inputs for loops which cannot be bounded by the internal analyzers. Both the WCC [LCF+09] and aiT [Abs14a] feature a loop analysis which detects constant loop bounds and bounds with simple, mostly affine behavior. For more complex loop structures and recursions the user has to provide *flow facts* to make a WCET analysis possible [KKP+11]. This can be done at the source code level in the WCC, as indicated in the following code fragments. These flow facts are gathered in the HLIR and LLIR (see Figure 3.2) and are automatically kept consistent by the flow fact management. The simplest and most widely used type of annotation is a loop bound as shown in Code Example 1, which specifies how often the loop body may execute once the loop head is entered from the outside of the loop. For a loop l, this is always specified through a *minium* and *maximum iteration count* B^l_{min} and B^l_{max}, i.e., in Code Example 1 we have $B^l_{\text{min}} = 8$ and $B^l_{\text{max}} = 12$.

```
_Pragma( "loopbound min 8 max 12" );
while ( input > 0 ) {
  input = read( p, q, r );
}
```

Code Example 1: A simple loop bound for an input-dependent loop.

For non-reducible loops (compare Section 4.1.2) loop bounds are not applicable, and for loops with varying iteration count like shown in Code Example 2, they may be imprecise. Therefore is is also possible to specify the loop behavior with a *flow restriction* which is a linear relation between execution frequencies of program points. As an example, the flow restriction from Code Example 2 states, that for every visit of the “outer” point, the “inner” one may be executed 55 times. In the same manner, we can also bound recursions from their initial call site as shown in Code Example 3.

```
_Pragma( "marker outer" );
for ( int i = 0; i < 10; ++i ) {
  for ( int j = i; j < 10; ++j ) {
    _Pragma( "marker inner" );
    act();
  }
}
_Pragma( "flowrestriction 1*inner <= 55*outer" );
```

Code Example 2: A precise bound for a triangular loop.

```
_Pragma( "marker call" );
recurse( 10 );
_Pragma( "flowrestriction 1*recurse <= 10*call" );
...
int recurse( int p ) {
  if ( p > 0 )
    return recurse( p - 1 );
  else
    return 0;
}
```

Code Example 3: A recursion bound.

Finally, also the entrypoints at which a single task may start can be specified in the same way as presented in Code Example 4. A number of properties like the execution period or task priority can be specified for each entry point. In addition, the entry points can also be set in the task system description file, in the case that the same code shall be used with different entry point configurations.

```
_Pragma( "entrypoint period=20ms" );
void task1() {
  ...
}
```

Code Example 4: An entry point specification.

3.4 System Model

Usually, a compiler or static analyzer only needs to know about the external view of the target architecture and the specification of the semantics of the high-level language. The internal structure and behavior of the architecture are not relevant as long as they correctly implement the specified *Instruction Set Architecture* (ISA). However, ISAs usually do not contain any information about the time which will be needed to execute a given instruction. Therefore, a static WCET analyzer also needs precise information about the behavior and structure of the hardware implementation as least as far as it affects the timing of an instruction execution. Considering the classical Gajski-Kuhn-Y-Chart for levels of hardware design as shown in Figure 3.3, WCET analysis will require knowledge of at least some aspects of the *Register-Transfer Level* to be able to reason about what may happen in a single hardware cycle. Since cycles are the lowest granularity of time that is considered in WCET analysis, the *Logic* and *Circuit* levels do not need to be known.

The initial design of the WCC included back-ends for the ISAs Infineon TriCore V1.3 and V1.3.1 [Inf08] and for the ARM v4T [ARM05]. The target platforms were

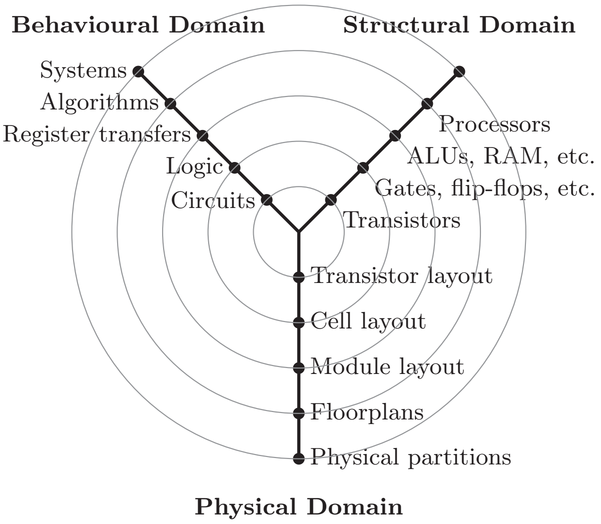


Figure 3.3: Levels of hardware design in a Gajski-Kuhn Y-chart [Gro08].

the TriCore TC1796/TC1797 [Inf09] and a generic ARM7TDMI platform [ARM04]. For all of these platforms also the virtual prototyping IDE CoMET [Syn14], which was donated to the chair by Synopsys, contains cycle-true simulation models. At the time of the first studies for this thesis, multiple multi-core architectures were considered. Desired properties included

- a high degree of configurability to be able to examine multiple hardware options,
- as close to a real-world architecture as possible to achieve realistic results,
- cores which operate as time-predictable as possible to ease the pipeline analysis,
- the architecture should be included in CoMET or at least be easily modeled with it, to ease ACET measurements.

Finally, the decision was made in favor of an ARM7TDMI-based multi-core, since this provided the best compromise among the mentioned points. The ARM7TDMI is a widely-used real-world core with highly predictable execution behavior. It is currently being replaced [Ele12] by its successor, the ARM Cortex M0, but both are almost identical when it comes to their functionality. They share a 3-stage in-order pipeline, the absence of on-chip caches and a low-power, highly predictable operation. Therefore, results gathered for the ARM7TDMI will also be valid for the Cortex M0. CoMET already includes a model of the ARM7TDMI and configurable platforms can easily be built. In addition, the WCC already contains a suitable back-end, which eases the implementation work.

The resulting architecture is shown in Figure 3.4. Each ARM7TDMI core is connected to a local core bus to which instruction and data scratchpads and caches are connected. The separation of data and instruction caches is a general recom-

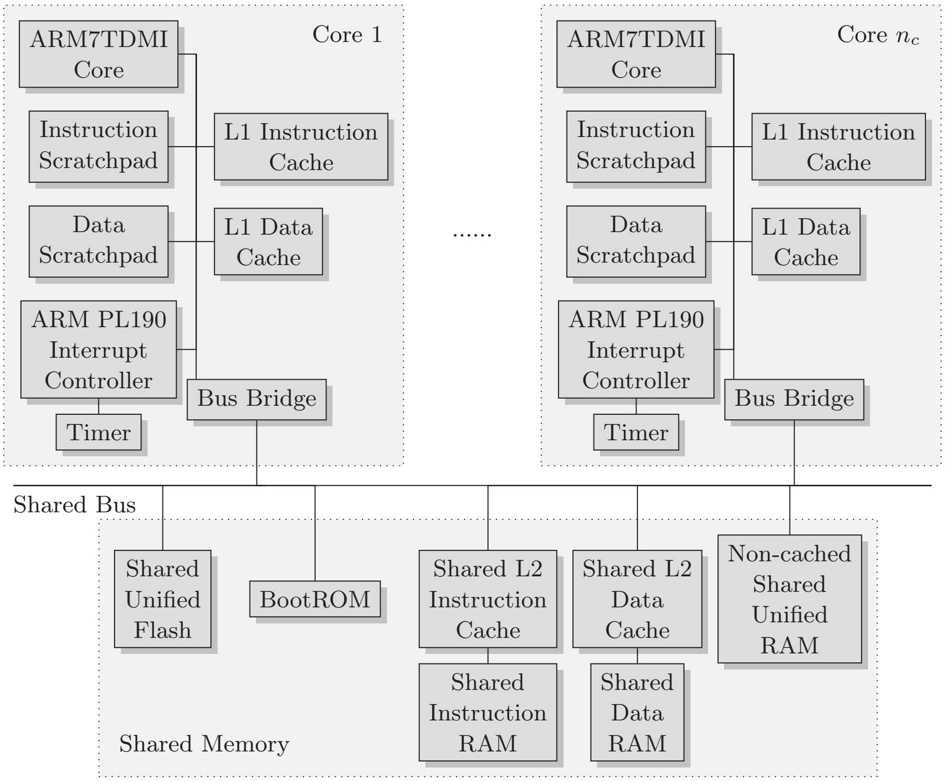


Figure 3.4: The multi-core system model.

mendation to increase the predictability [WGR+09]. Each core also has a timer and an ARM PL190 interrupt controller to be able to start time-triggered tasks, and a bus bridge to access the shared memory. The bridge is connected to the shared bus whose arbiter can be configured freely among multiple variants mentioned below. Shared, cached and uncached RAM memory is attached to the shared bus, again split between instruction and data in the cached case. In addition, a slower shared flash memory is assumed and a BootROM, which contains the system ROM generated by WCC (cf. Figure 3.2). All RAM memory is modeled as SRAM, since the analysis of standard DRAM refresh cycles makes a precise WCET analysis impossible [BM11]. This organization of memory is modeled in accordance with real-world memory hierarchies such as the Infineon TriCore TC1797 [Inf09]. Especially for the multi-core case such a distributed memory, partitioned into core-local, private modules, and shared modules is realistic [Fre09]. Also, with the choice of rather simple cores and a multi-level memory hierarchy, this design follows the PROMPT principles [CFG+10] for predictable multi-core design.

The cache and shared bus simulation modules were developed in the course of this thesis, since the CoMET-supplied cache is restricted to random replacement which is inherently not analyzable, even not with probabilistic techniques [Rei14].

Component	Property	Default Value	
Cores	Scheduler	Time-Triggered Non-Preemptive Dispatcher	
	Clock rate	200 MHz	
Caches		L1	L2
	Hit Delay	1 cycle	1 cycle
	Miss Delay[1]	1 cycle	2 cycles
	Size	8 kB	32 kB
	Associativity	2	4
	Line size	32 B	64 B
	Replacement strategy	LRU	LRU
	Write through	true	true
	Write allocate	true	true
Buses		Core Bus	Shared Bus
	Width	4 B	4 B
	Clock rate	200 MHz	200 MHz
	Arbitration strategy	FAIR (1 master)	FAIR (n masters)
	Arbiter delay	1 cycle	1 cycle
Memories		Size	Access Delay
	Boot ROM	4 MB	3 cycles
	Instruction Scratchpad (per core)	32 kB	1 cycle
	Data Scratchpad (per core)	32 kB	1 cycle
	Shared I-RAM	512 kB	3 cycles
	Shared D-RAM	512 kB	3 cycles
	Uncached Shared RAM	1 MB	3 cycles
	Shared Flash	8 MB	5 cycles
	Memory-mapped device registers		1 cycle

Table 3.1: Default system parameters.

Similarly, the CoMET-supplied bus does not offer time-triggered arbitration strategies, whose timing predictability was shown to be superior [PS10].

Since the experiments are done on a simulator, system parameters can be varied, depending on which scenarios should be modeled. The parameters and their default values are given in Table 3.1. The memory parameters are modeled in accordance with the real-world parameters of the Infineon TriCore TC1797 [Inf09]. The scheduler for the cores is generated by WCC on demand and simply invokes the

[1] The miss also triggers the reloading of the cache line from the next higher hierarchy level which dominates the total miss penalty and leads to a total miss delay far higher than 1 or 2.

annotated periodic tasks (cf. Code Example 4) according to their period. However, this is completely optional and can be exchanged for a user-defined RTOS. Finally, the default memory layout is such that instructions (`.text` section) and the stack are placed in the scratchpads, whereas global data (`.data` and `.bss` sections) is placed in the non-cached shared RAM.

Before the simulation starts, the hardware parameters are set and the boot ROM memory is filled with the contents of a WCC-compiled system ROM. The boot loader (cf. Figure 3.2) which runs on core 1 then unpacks this system ROM, which contains an ELF binary for each core. After the unpacking, each core decodes its ELF file and places the content according to the memory layout specification described above. Then the execution begins and either runs to completion (in case of non-periodic core tasks) or runs for a specified number of hyperperiods (in case of periodic core tasks) where the hyperperiod is the smallest common multiple of all tasks' periods. Probes in the CoMET simulation record the runtime of each task non-intrusively as well as numerous other microarchitectural events, such as cache hits and misses and bus arbitration events.

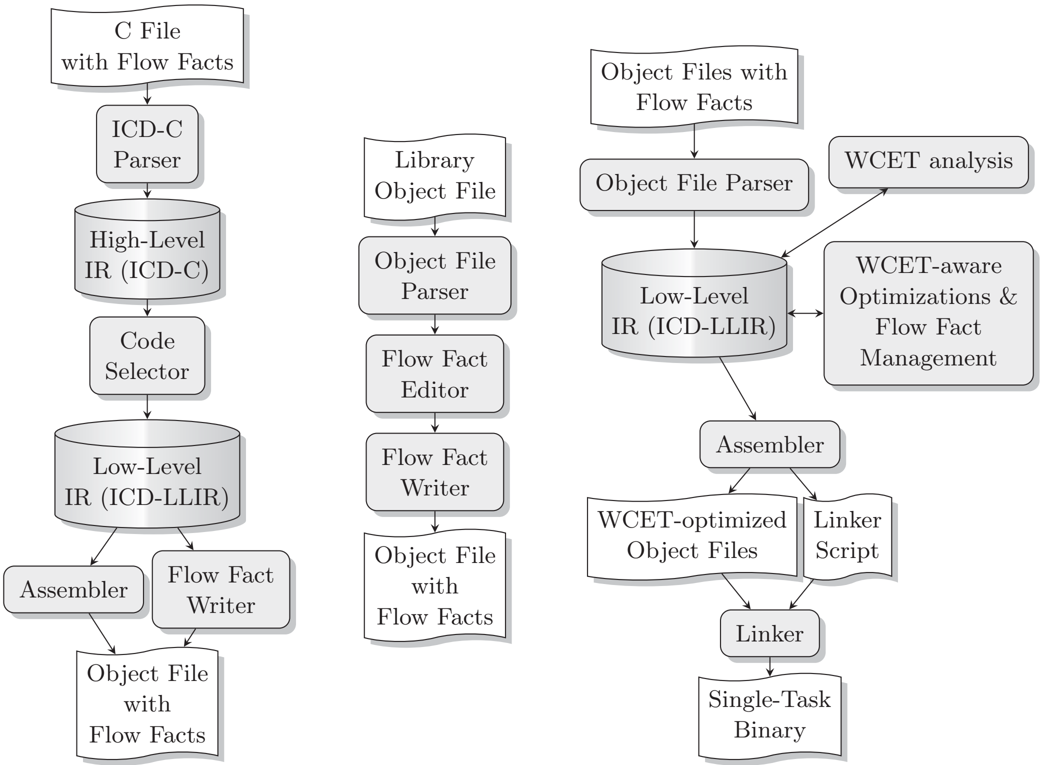
3.5 Extensions for Binary Input Files

An inherent problem with the workflow of the WCC has been the handling of binary input files. The original structure of the WCC as shown in Figure 3.1 only had access to the compiled translation units at the LLIR level. Unfortunately, often external libraries are linked into the program, which also contribute to the WCET, since they are called from within the compiled code. Even worse, programs may also use compiler-internal libraries implicitly since floating point arithmetic and integer division is not implemented in hardware on platforms like the ARMv4T. Therefore, the compiler will insert calls to precompiled software routines where these operations are used in the source code.

To be able to provide the internal WCET analyzer but also the optimizations with a complete view of the analyzed binary, WCC modules were developed to

- *read in library files*, e.g. relocatable object files,
- *reconstruct the control flow graphs* for the functions in these files
- provide a possibility to *read and store flow facts* inside the binary files.

This additional module also provides the WCC with the capability to compile each C file in separation, producing separate object files which contain the compiled code as well as the flow facts needed to analyze this code as shown in Figure 3.5a. Another WCC run can then read in all of these object files, perform the WCET analysis and possibly the low-level optimization as shown in Figure 3.5c. Of course, high-level WCET optimizations are impossible in this case. Flow facts can also be added to external libraries which were not compiled with WCC with the help of a command-line flow fact editor (cf. Figure 3.5b). The combination of C input files and binary input files is also feasible like shown in the multi-core compilation structure from



(a) Compilation of a single translation unit

(b) Annotation of a library file

(c) Compilation with binary input files

Figure 3.5: Interaction of binary input file modules with the rest of WCC.

Figure 3.2. In the following, we will shortly summarize how different challenges in the design of these modules were solved. A more detailed description can be found in [Gün13].

Container and instruction parser The first step during the read process is the reading of the object file format. Since multiple container formats like ELF and COFF have been established, we use the existing GNU `libbfd` to parse the container format.

The object file is partitioned into *sections* in which code and data are stored. Initialized data can be taken over into the LLIR in the form of a byte sequence, and instructions are parsed by using a *linear-sweep* over the machine instructions. Each instruction is read according to the ARM binary format definitions [ARM05]. During this process, the symbol table, relocation table and branch instructions of the program are used to add *labels* to the code and data. This is needed, since the LLIR is an assembler-level representation which requires labels for all data accesses and branch instructions. Another complication arises from the fact, that in the ARM case, data objects are sometimes embedded into the code. These are identified via debug information or via the load instructions that try to access the data.

CFG reconstruction The inserted labels partition the instruction sequence into disjoint basic blocks. To re-build the CFG the edges of the graph now need to be added. For *branch instructions with a fixed target*, usually a program-counter-relative byte offset, the control flow is easy to reconstruct. In these cases a CFG edge towards the target is inserted. In the case that the branch is executed conditionally, a second edge is inserted, which points towards the *fall-through* successor, i.e. the basic block following the current one in the instruction sequence.

For *dynamic branch instructions*, like a branch to a target given by the contents of a register, the situation is more complex. In these cases, one can either rely on pattern matching [Tid10] to detect patterns that are used to return from a function or to jump into a switch statement, or data-flow analysis has to be carried out concurrently to the CFG reconstruction to bound the possible branch targets [BHV11; KV08; KZV09; The00]. Both methods may fail and require the user to explicitly specify the targets of dynamic branches. To limit the implementation effort, the implemented modules use the pattern-matching approach, which works well for all compiler-generated code but is more likely to fail for hand-written assembly code.

Flow fact specification Flow facts as presented in Section 3.3 are either loop bounds or flow restrictions, which are needed to compute the WCET. In C files they are represented as source code pragmas which is obviously not possible for binary input files. Instead, the capability of the object file format to host an undefined number of sections is exploited. A new section is inserted which stores

- Loop bounds,
- Flow restrictions and
- Branch targets

in ASCII text form. Since the annotations are stored as text, the basic blocks are referenced using unique names that are generated by the flow fact reader. The implementation ensures that the generated names are the same for every invocation of the flow fact reader, since otherwise the reference points for the flow facts would be lost.

To fully exploit the possibilities of modular development and WCET analysis, parametric flow facts would be needed for an increased precision. This in turn would require a parametric WCET analysis as discussed in Section 2.2.2, which is not available in aiT and WCC. Therefore, parametric flow facts are not yet supported.

Chapter 4

Single-Core WCET-Analysis

Contents

The single-core analysis implementation is the basis for the extension to the multi-core case. We will therefore review its structure in this chapter and point out different challenges that arise during the analysis of predicated instructions in the ARM instruction set [ARM05]. Since we assume a non-preemptive scheduler, this analysis is both single-core and single-task, i.e., the code for each core and each task within this code (identified by the entry points, cf. Section 3.3) is analyzed in separation.

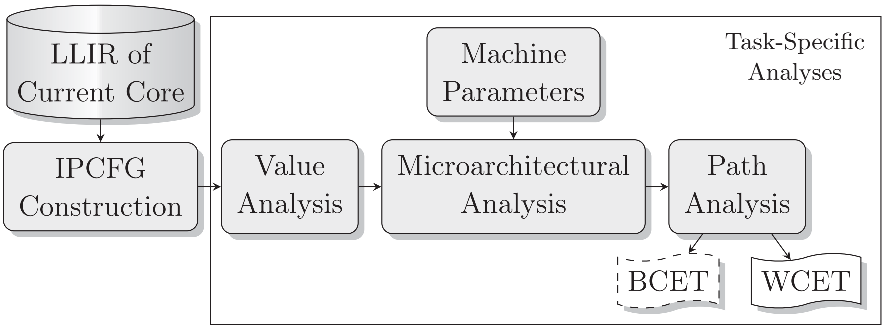


Figure 4.1: Structure of the WCC-internal, single-core, single-task WCET analysis.

The different phases of the WCC-internal analyzer are shown in Figure 4.1. Compared to the generic static WCET analysis structure from Figure 2.4, we have the noticeable difference that the input is an annotated LLIR code stemming either from the code selector (cf. Section 3.2) or the binary parser (cf. Section 3.5). The LLIR code contains both the flow facts and the intra-function CFGs which removes the necessity of a full CFG reconstruction, but still we will create an *Interprocedural Control Flow Graph* (IPCFG) for each task in the system. The IPCFG contains all flow fact information as later needed by the path analysis. Therefore, there is no further flow of user annotations into the path analysis. Based on the IPCFG, the values of the CPU registers are statically approximated for each program point by means of abstract interpretation. These values are then used in the microarchitectural analysis, again an abstract interpretation on a specialized domain, to determine memory access targets and machine instruction operand values. Since our architecture is configurable (cf. Section 3.4), the machine parameters must be specified at this point. Finally, a path analysis determines the WCET. The BCET is also determined if

- we are not exploiting the fact that the architecture is timing-anomaly-free (cf. Section 2.2.8), i.e., we are not tracking the local worst-case state only and
- the specified flow fact set includes *minimum* iteration bounds for all loops, too, as opposed to the *maximum* bounds required for WCET analysis.

During all of the following analyses we have the problem, that we will be doing computations on a machine with limited-precision integral types, i.e., overflows and underflows may occur. We handle this by performing all computations on the abstract wrapper type `SafeInt`[1], which makes integer overflows in the analysis explicit by throwing C++ exceptions. If a register content in the value analysis is subject to an overflow, these exceptions are handled by setting the register content to the maximum possible interval (the "top"-value of the lattice). If other fields like the execution time of a basic block overflow, the analysis is aborted. In our experiments the latter case never occurred, since the respective variables are appropriately sized and their variance is more limited.

4.1 IPCFG Construction

All of the following analysis stages require an IPCFG to work on, we therefore shortly introduce this graph and how it is constructed in the following. The IPCFG construction from Figure 4.1 can be partitioned into multiple sub-phases, which finally produce a *context graph* on which all of the following stages will work. Figure 4.2 shows the preceding construction steps which we will explain in the following.

For every core c and any task τ in the set T_c of tasks mapped to core c, the LLIR already contains the CFGs $G_c^f = (V_c^f, E_c^f)$ of every function f in the LLIR of c. In

[1] `https://safeint.codeplex.com`

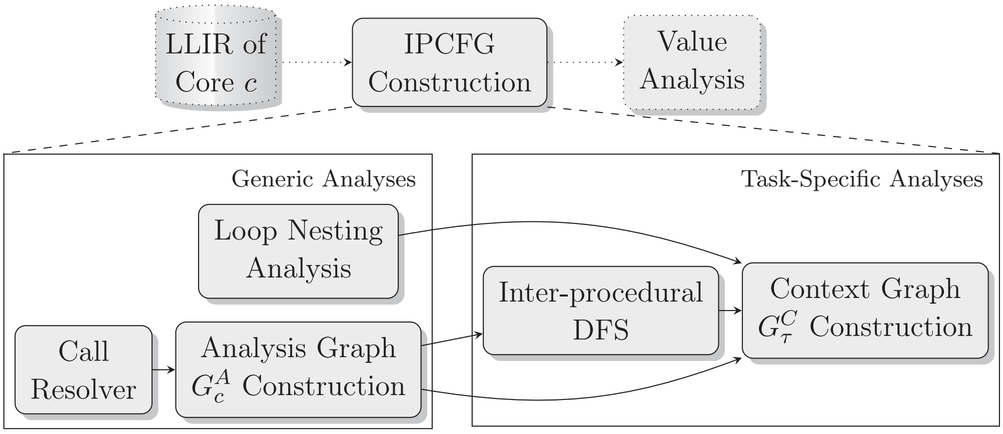
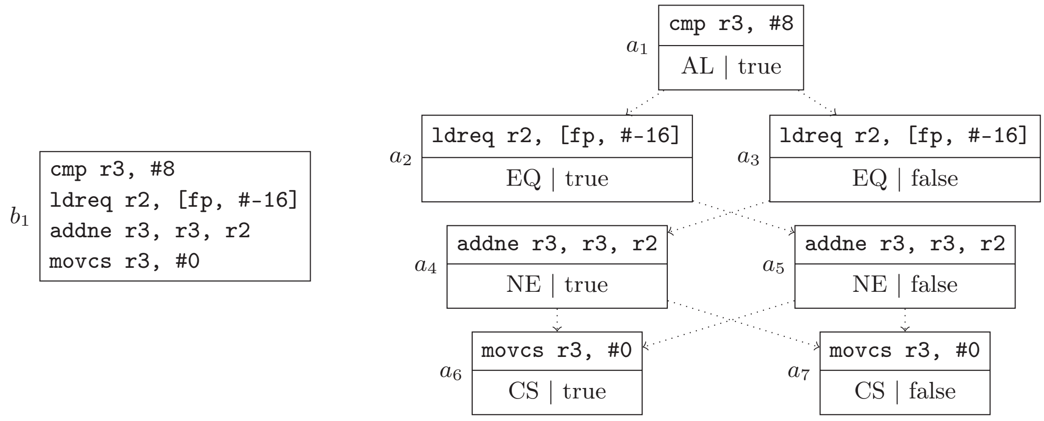


Figure 4.2: Stages of the IPCFG construction.

addition, the entry functions $f_\tau^\perp$ for each task τ which runs on c are given. A node $v \in V_c^f$ is a *basic block*, i.e., a sequence of instructions $(i_1, \ldots, i_n)$ which can only be entered at i_1 and only be exited at i_n in non-preempted execution. The edges $e \in E_c^f$ represent transfers of control, but function calls are not resolved. When a function f_2 is called by an instruction $i \in v \in V_c^{f_1}$, the block v ends but it only has one outgoing edge which points to its successor block in f_1. The main objective of the IPCFG is to overcome this limitation and to connect the G_c^f to each other via call and return edges. In addition, the following is desired:

- Implicit control-flow through the use of predicated execution as it is present on many architectures including the ARM architecture [ARM05] should be explicitly visible in the IPCFG. The *analysis graph* G_c^A from Figure 4.2 incorporates this, as well as the call/return edges between functions.
- The loop structure of the program should be easily visible from the IPCFG. This is optional, but a proper loop detection will raise the precision of the following analyses. This is addressed by the *Loop Nesting Analysis* shown in Figure 4.2.
- Invocations of the same function or loop body from different call sites or in different iterations will possibly show different runtime behavior. To be able to distinguish these *analysis contexts*, a *context graph* G_τ^C is created based on G_c^A which virtually inlines functions and unrolls loops according to user specifications.

We did not use standard data-flow analysis generators like [SSB09; HMM12] to construct the graphs and the analyses, since we will need to introduce a new type of context in the multi-core WCET analysis which requires full control over the underlying graph and the DFA algorithm.



(a) A basic block with predicated instructions.

(b) The resulting analysis blocks. All edges are predication edges.

Figure 4.3: Analysis block creation example.

4.1.1 Analysis Graph

Definition 9. *The* analysis graph $G_c^A = (V_c^A, E_c^A)$ *for the LLIR code of core* c *is a directed graph where each* analysis block $v = ((i_1, \ldots, i_n), p, q) \in V_c^A$ *is a tuple containing an instruction sequence* $(i_1, \ldots, i_n)$*, a predicate* p *from a set of machine-defined predicates* P *and a truth assignment* $q \in \mathbb{B}$ *that indicates whether* p *is true during the execution of* v*. Each edge* $e \in E_c^A$ *has a type* (e) *which is either BRANCH, CALL, RETURN or PREDICATION.*

The first step in the construction of the analysis graph is to split up each basic block into one or more analysis blocks. The compiler or assembly programmer may decide to *predicate* some instructions. In the case of the ARM architecture, this is possible for almost every instruction. Predication means that the instruction is only executed if certain CPU flag registers are in a state that is described by the predicate. As an example, the `cmp` instruction in Figure 4.3a sets the flag values according to the comparison of `r3` and the constant `8`. It is executed unconditionally which is expressed by the implicit predicate `al` (*always*), i.e., `cmp` is the same as `cmpal`. The `ldr` with predicate `eq` is only executed when the "equal"-flag is set, similarly `ne` (`cs`) is only executed when the "not equal"-flag ("carry"-flag) is set. In Figure 4.3b the blocks with the "true" value represent the case that the predicate is true, i.e., the instruction is executed, whereas the "false" blocks represent the skipping of the instruction by the CPU. Since some predicates are contradictory to each other, as e.g., `eq` and `ne`, edges between the contradicting analysis blocks can be removed. Of course, as soon as an instruction is executed which may alter the CPU flag registers the contradiction is potentially invalidated. We can therefore easily define a *split function* $split^A : V_c^f \to 2^{V_c^A} \times 2^{V_c^A \times V_c^A}$, which maps each basic block to a subgraph of analysis blocks and edges which model its predication-aware execution paths in analogy to Figure 4.3. This function can be implemented by a single scan over the basic block, which splits it into contiguous chunks with the same predicate which are then connected by predication edges according to their mutual exclusion behavior. $\delta_\perp^A(v) \subseteq V_c^A$ denotes the analysis blocks that model the *entry* into basic block v, whereas $\delta_\top^A(v) \subseteq V_c^A$ is the set of analysis blocks which are an *exit* of v. The set $\delta_\perp^A(v)$ has two elements if the first instruction of the block has a predicate other than `al`, else it has only one element. The same applies to $\delta_\top^A(v)$. The reverse mapping of an analysis block v^A to its basic block v is given by $\mu^A(v^A)$. For any type of graph, the successors and predecessors of node v are given by $\delta^+(v)$ and $\delta^-(v)$, respectively. For a function f, $\nu_\perp(f) \subseteq V_c^f$ gives the entry block and $\nu_\top(f) \subseteq V_c^f$ returns the set of return blocks for the function.

Algorithm 2 Analysis Graph Construction.

```
$G_c^A = \bigcup_{f \in c, v \in V_c^f} \left( split^A(v) \right)$        ▷ Initialize as union of basic block subgraphs
for $v^A \in V_c^A$ do
    if $call(v^A)$ then                        ▷ Add call and return edges for calls
        for $f \in targets(v^A)$ do
            $E_{\text{call}} = \{v^A\} \times \delta_\perp^A(\nu_\perp(f))$
            $\forall e \in E_{\text{call}} : type(e) \leftarrow \text{CALL}$
            $E_{\text{return}} = \delta_\top^A(\nu_\top(f)) \times \delta_\perp^A(\delta^+(\mu^A(v^A)))$
            $\forall e \in E_{\text{return}} : type(e) \leftarrow \text{RETURN}$
            $E_c^A \leftarrow E_c^A \cup E_{\text{call}} \cup E_{\text{return}}$
    else                                       ▷ Add local CFG edges for all other blocks
        $E_{\text{branch}} = \{v^A\} \times \delta_\perp^A(\delta^+(\mu^A(v^A)))$
        $\forall e \in E_{\text{return}} : type(e) \leftarrow \text{BRANCH}$
        $E_c^A \leftarrow E_c^A \cup E_{\text{branch}}$
```

Algorithm 2 summarizes the analysis graph construction. The functions *call* : $V_c^A \to \mathbb{B}$ and *targets* : $V_c^A \to 2^{V_c^A}$ compute whether the given block ends with a call and which functions this call may be targeted at. In the WCC, these functions are implemented by a *call resolver* as sketched in Figure 4.2, which tries to look up the function name if the call is carried out via a symbol name. In case of indirect calls via function pointers, the user must be queried for the call target.

4.1.2 Context Graph

The analysis graph already provides the possibility to perform an interprocedural data-flow analysis. Its main drawback is that it does not distinguish between different *contexts* of the code. A function can show different behavior when called from two sites with varying parameter sets, and the same is true for the iterations of code in a loop. As an example consider the analysis graph shown in Figure 4.4. When analyzing the recursive function `bar`, the information stemming from `foo1` and `foo2` will be merged. To overcome this, *virtual inlining* of functions and *virtual unrolling*

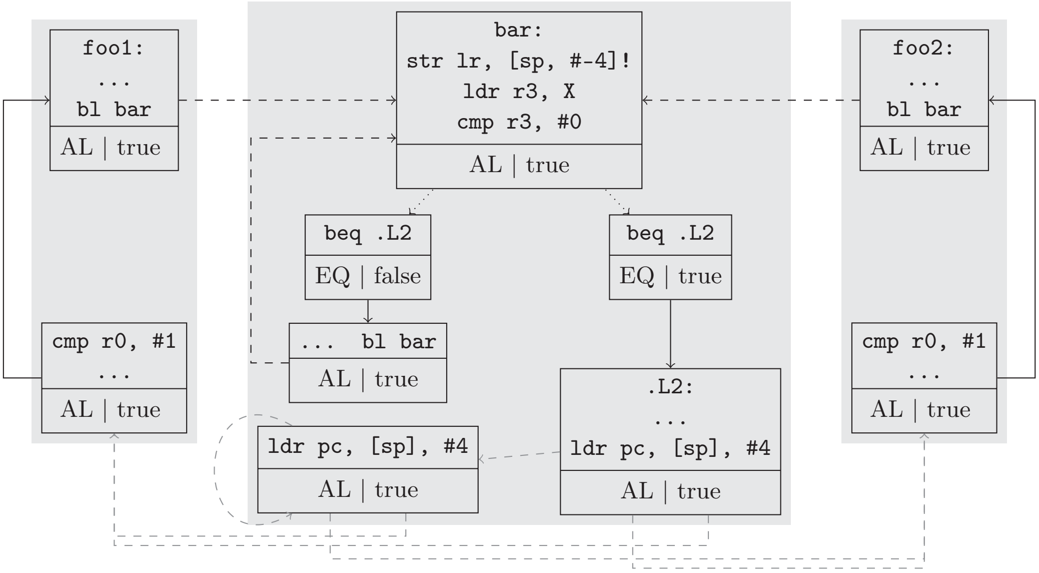


Figure 4.4: Example for an analysis graph with a directly recursive function. Solid edges are branches, black (gray) dashed edges are calls (returns) and dotted edges are predication edges.

of loops (VIVU) [LM97] is used to create *call contexts* and *iteration contexts*. Note that none of these graph transformations actually modifies any source or machine code. We are only concerned with the disambiguation of different execution contexts here, not with code transformations.

Definition 10. *For a task $\tau \in T_c$ with entry function $f_\tau^\perp$, a* call string *is a sequence $S = ((v_0, f_0), (v_1, f_1), (v_2, f_2), \ldots, (v_n, f_n))$ such that $f_0 = f_\tau^\perp$ and for all i, $v_i \in \bigcup_{v \in V_c^{f_i}} \delta_\mathrm{T}^A(v)$ has an outgoing call edge to function f_{i+1}. A* call context *for $(v_i, f_i) \in S$ is a graph G which models the execution of f_i when called via the prefix of S with length i.*

The creation of call contexts is shown in Algorithm 3. As long as the call string S has length $|S| > 1$, we inline it by cloning each function f_i with $i > 1$ in S and by redirecting the call and return edges from f_{i-1} to the new copy of f_i. In Algorithm 3 this is formulated through a recursion. Every $(V_\mathrm{dup}, E_\mathrm{dup})$, constructed during the virtual inlining, is a call context of the respective function f_n.

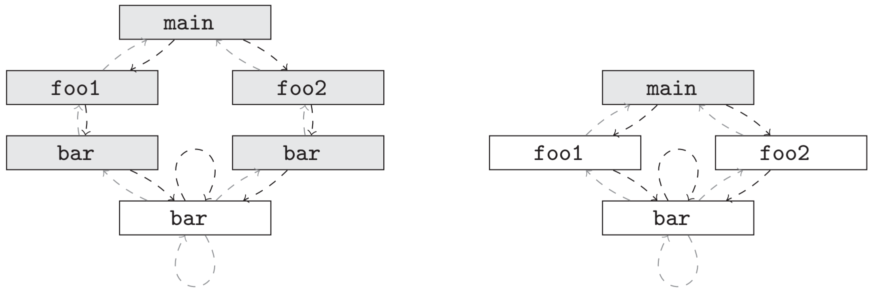
As an example, assume that in Figure 4.4 `foo1` and `foo2` are called from the `main` function of the task. The resulting context graph for the virtual inlining of

1. (`main`, `foo1`, `bar`, `bar`) and
2. (`main`, `foo2`, `bar`, `bar`)

(call nodes are omitted here for brevity) is shown in Figure 4.5a. To limit the graph size, all blocks of each function were collapsed to a single node in Figure 4.5a. Both of the above call strings have length 3 and it should be clear that the result graph

Algorithm 3 The virtual inlining algorithm.

```
function VirtualInline($S = ((v_0, f_0), \dots, (v_{n-1}, f_{n-1}), (v_n, f_n))$, $G^A_{\text{in}}$)
    if $|S| = 1$ then
        return $G^A_{\text{in}}$                          ▷ $S$ does not contain further calls.
    else
        $G^A_{n-1} \leftarrow (V^A_{n-1}, E^A_{n-1}) = \textsc{VirtualInline}(((v_0, f_0), \dots, (v_{n-1}, f_{n-1})), G^A_{\text{in}})$
                                                          ▷ Inline all but the last call
        $V_{\text{dup}} = \{\delta^A(v) \mid v \in V_c^{f_n}\}$          ▷ Clone function $f_n$ from $G^A_{n-1}$
        $E_{\text{dup}} = E^A_{n-1} \cap (V_{\text{dup}} \times V_{\text{dup}})$        ▷ And $f_n$'s edges
        return $G^A_{n-1} \cup copy(V_{\text{dup}}, E_{\text{dup}})$ with all call edges from $v_{n-1}$ to $f_n$
            and all return edges from $f_n$ to successors of $v_{n-1}$ redirected to
            $copy(V_{\text{dup}}, E_{\text{dup}})$.
```



(a) Maximum call string length 3.

(b) Maximum call string length 1.

Figure 4.5: Possible virtual inlining results for Figure 4.4. Call contexts are shown with gray background, default contexts in white.

grows linearly with the length of the call string. Also note, that by inlining the call strings of length 3 implicitly also all call strings of length less than 3 were inlined, due to the recursive nature of the definition.

If we do not have recursive functions in the task, the number of distinct call strings is finite. However, if we have recursive functions like `bar`, an unlimited number of call strings results. Therefore, the user has to specify limits for the inlining process. To this end, we use the limited-length call-string approach from [SP78]. The inlining proceeds is done for all call strings up to a length specified as the *maximum call string length*. If this length is reached, all further calls are directed to a *default context* of the callee instead of to a new call context. Figure 4.5a is an example for this procedure with a maximum call string length of 3. After the first visit to `bar`, all further calls to `bar` are modeled by its default context shown in white. If the maximum call string length is set to 1 instead, all calls are directed to default contexts as shown in Figure 4.5b.

Since the detection of call instructions at the machine code level can be non-trivial on some architectures, another option for context construction is to identify every movement of the stack pointer [LBS+10] and to build "call" contexts from these stack modification points. This method can also deal with obfuscated code which does not adhere to any calling convention, but is more costly in terms of analysis time. If the call string is not made explicit in the graph but carried along with the data items as an extension of the analysis lattice, it can be stored as a single numeric value in the best case [SZW+10]. This allows each data-flow analysis on the graph to use different context but also complicates and delays the individual analyses.

Since we need to set a bound on the call string length in case of recursive programs, we first need to find out whether recursion exists in the given task.

Definition 11. *A* spanning tree $G_{span} = (V, E_{span}, v_0)$ *for a CFG* $G = (V, E, v_0)$ *is a subgraph of* G *with directed edges* $E_{span} \subseteq E$ *such that* $\forall w \in V : v_0 \rightsquigarrow v \implies |P_{[v_0,w]}| = 1$*, i.e. a single unique path exists to each reachable node. The length of this path is given by* $\mathit{dfs}^{\mathit{depth}}(v)$*.*

A Depth-First Search *(DFS) on a CFG* $G = (V, E, v_0)$ *returns a classification of edges* $\mathit{dfs}^{\mathit{type}}(e) \in \{\mathit{TREE}, \mathit{BACK}, \mathit{CROSS}, \mathit{FORWARD}\}$ *for each edge* $e \in E$*. The tree edges form a spanning tree* G_{span} *of* G *in which back edges* (v, w) *have* $\mathit{dfs}^{\mathit{depth}}(v) > \mathit{dfs}^{\mathit{depth}}(w)$*, forward edges have* $\mathit{dfs}^{\mathit{depth}}(v) < \mathit{dfs}^{\mathit{depth}}(w)$*, and for cross edges* $v \not\rightsquigarrow w$ *in* G_{span}*.*

To detect recursions, we extend the analysis graph G_c^A by a virtual entry node v_0 and edges $\{v_0\} \times \bigcup_{c \in \{1,\ldots,n\}} \bigcup_{\tau \in T_c} \delta_\perp^A(\nu_\perp(f_\tau^\perp))$ where the tasks on core c are again given by T_c and for any $\tau \in T_c$ the entry point function is $f_\tau^\perp$. Recursive calls are contained if and only if an edge $e = (v, w)$ with $\mathit{type}(e) = \text{CALL}$ and $\mathit{dfs}^{\text{type}}(e) = \text{BACK}$ is found in a DFS on this extended graph. If the user did not specify an explicit maximum call string length, we only create a single default context for each recursive function.

The last aspect from Figure 4.2 that is still missing now, is the handling of loops. To handle them, we first need a proper definition of a loop.

Definition 12. *A node* $v \in V$ dominates *a node* $w \in V$ *in a CFG* $G = (V, E, v_0)$ *iff* $v = w$ *or* $\forall p \in P_{[v_0,w]}^{acyclic} : p = (v_0, \ldots, v, \ldots, w)$*. We denote this as* $v \operatorname{dom} w$*.*

If for a node $v_{head} \in V$*, back-edges* $e = (w, v_{head}) \in E$ *with* $v_{head} \operatorname{dom} w$ *exist, a* natural loop l *with head* v_{head}^l *is induced by these edges. The set of all natural loops in task* τ *is denoted by* $\circlearrowleft_\tau$*. For natural loops, a unique* nesting relation $<_\circlearrowleft$ *exists. Iff all back-edges of a graph* G *end in natural loops' heads,* G *is called* reducible *and has only a single DFS spanning tree* G_{span}*, else* G *is* irreducible.

Both the loop detection and the inter-procedural DFS can be performed in time $O(|V_c^A|)$ [ALS+07]. All natural loops have a single entry node but may have multiple back and exit edges as shown in Figure 4.6a.

Definition 13. *A* loop context *for loop* $l \in \circlearrowleft_\tau$*, iteration* i *is a graph* G_l^i *which models the i-th iteration (and only the i-th iteration) of* l *counted from* 1 *on.*

Algorithm 4 The virtual unrolling algorithm.

```
1:  function VirtualUnroll($G_{\text{in}}^A$, $l \in \circlearrowleft_\tau$, $n \in \mathbb{N}_0$)
2:      $V_l = \{v \in V_{\text{in}}^A) \mid (v_{\text{head}}^l \operatorname{dom} v) \land (v \rightsquigarrow v_{\text{head}}^l)\}$      ▷ Get loop body
3:      $E_l = \{(v,w) \in E_{\text{in}}^A \mid v \in V_l \lor w \in V_l)$      ▷ And all loop edges
4:      for $i \in \{1,\dots,n\}$ do      ▷ Create loop context $i$
5:          $G_l^i \leftarrow copy(V_l, E_l)$      ▷ Copy the loop
6:          if $i > 1$ then      ▷ If this is not the first iteration
7:              $E_l^i \leftarrow E_l^i \setminus \{(v,w) \mid v \notin V_l^i \land w \in V_l^i\}$
                                                   ▷ Remove the external entry edges
8:              $E_l^i \leftarrow E_l^i \cup \{(v, v_{\text{head}}^i) \mid (v,w) \in E_l^{i-1} \land dfs^{\text{type}}((v,w)) = \text{BACK}\}$
                                                   ▷ And replace them by back edges ...
9:              $E_l^{i-1} \leftarrow E_l^{i-1} \setminus \{e \mid e \in E_l^{i-1} \land dfs^{\text{type}}(e) = \text{BACK}\}$
                                                   ▷ ... from the previous iteration
10:         if $i \le B_{\text{min}}^l$ then      ▷ If a loop exit is impossible in iteration $i$
11:             $E_l^i \leftarrow E_l^i \setminus \{(v,w) \mid v \in V_l^i \land w \notin V_l^i\}$      ▷ Remove the exit edges
12:     return $(G_{\text{in}}^A \setminus (V_l, E_l)) \cup \bigcup_{i \in \{1,\dots,n\}} G_i^l$
```

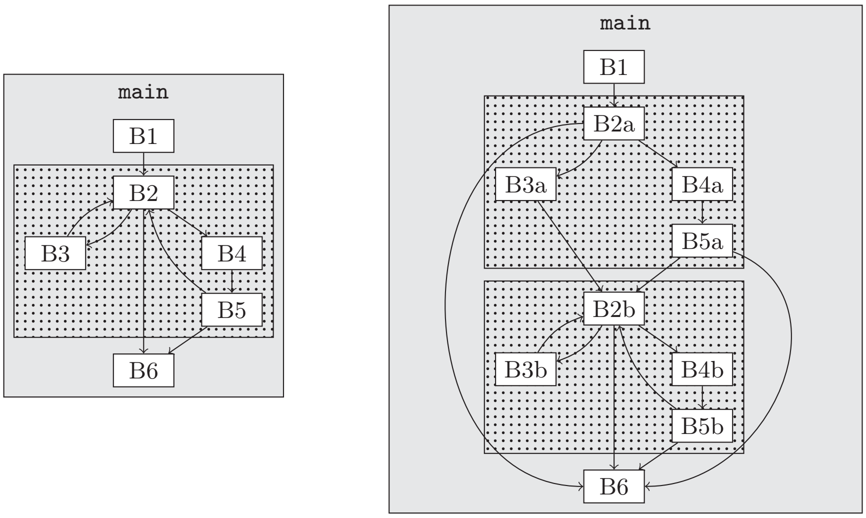
Algorithm 4 presents how loop contexts for the first $n-1$ iterations of loop l are constructed. In each of the algorithm's iterations i, we unroll the i-th iteration of l by duplicating l's nodes and edges. From the outside, l can only be entered in the first iteration which is why we remove the entry edges from all but the first iteration in line 7. The iterations $i > 1$ are only entered via the back-edges of preceding iterations (line 8 and 9).

The result of unrolling the first iteration of the loop from Figure 4.6a is shown in Figure 4.6b. If the minimum loop iteration count B_{min}^l is bigger or equal to the unrolled iteration's index i, we can remove the exit edges from the unrolled iteration (line 11 in Algorithm 4). For the example of Figure 4.6b, this implies that the edges $(\text{B2a}, \text{B6})$ and $(\text{B5a}, \text{B6})$ can be removed if the $B_{\text{min}}^l \ge 2$.

Definition 14. *A* context graph $G_\tau^C = (V_\tau^C, E_\tau^C, v_{0,\tau})$ *for a task* $\tau \in T_c$ *is a CFG which is a copy of an analysis graph* G_c^A *that was subject to virtual inlining and unrolling, where loops in* G_c^A *were classified starting from root node* $v_{0,\tau} = \delta_\perp^A(\nu_\perp(f_\tau^\perp))$.

A context graph is based on a set of contexts C. *Each context* c_i *has an associated subgraph* $G_\tau^{c_i} = (V_\tau^{c_i}, E_\tau^{c_i}, v_0^{c_i})$ *and there exists a set of* transition edges E_{trans} *between the contexts such that* $V_\tau^C = \bigcup_{c_i \in C} V_\tau^{c_i}$ *and* $E_\tau^C = \bigcup_{c_i \in C} E_\tau^{c_i} \cup E_{trans}$. *Each* c_i *is classified as either a* call, iteration *or* default *context as introduced above.*

The context graphs of the tasks will be the basis for all of the following analyses. For bigger maximum allowed call string lengths and maximum virtual unrolling factors, the size of C increases and so does the analysis precision, but also its duration.



(a) Original function with natural loop.

(b) Result of virtually unrolling the first iteration.

Figure 4.6: Example for virtual unrolling. Call contexts are shown with gray background, iteration contexts are dotted.

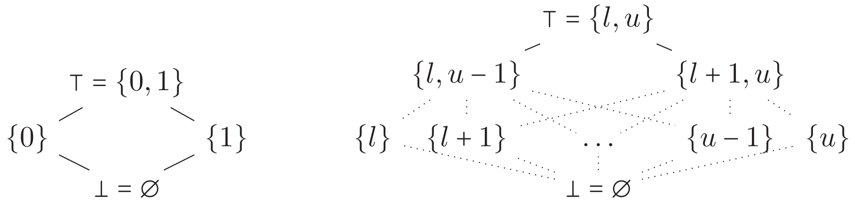
4.2 Value Analysis

Later analysis stages will require knowledge about the addresses which may be affected by a memory access or the value of a register which determines the runtime of an arithmetic operation, often seen in integer multiplication and division instructions. Therefore, the purpose of the *value analysis* is to determine safe approximations of the possible contents of memory cells in the system. To this end, we use abstract interpretation as introduced in Section 2.1. But even with an abstracted value semantics it is computationally infeasible to approximate the whole memory content of the system. Therefore, value analyses are usually restricted to the CPU registers, since these are most important for the timing behavior. To some extent, also the stack contents can be modeled, but we will restrict the analysis to the most relevant case of CPU registers here.

Thus, the value analysis must determine an approximation $val_v^{\text{in}} \in \mathbb{V}$ on an abstract value domain $\mathbb{V}$ for every node v in a context graph G_τ^C such that $val_v^{\text{in}}(r)$ safely approximates the content of any register r from a set of CPU registers $R = \{r_1, \dots, r_n\}$. We achieve this by means of a work-list DFA as shown in Algorithm 1 whose domain and transfer functions will be detailed in the following.

4.2.1 Abstract Value Domain

Some CPU operations treat register contents as *bit strings* like, e.g., shifting and logical operations, and some perform signed integer, unsigned integer and floating



(a) Single bit value lattice $\mathbb{B}_1$.

(b) Interval lattive $\mathbb{I}_{[l,u]}$

Figure 4.7: Hasse diagrams of value analysis domain components.

point arithmetic on them. Only the latter case can be excluded for the ARM7, since it has no floating point registers. Instead, it has 16 32-bit general purpose registers, which are used for bit operations, signed two's complement arithmetic and unsigned arithmetic. Therefore, we define the value domain $\mathbb{V}$ as

$$\mathbb{V} = R \to (\mathbb{B}_1^{32} \times \mathbb{I}_s \times \mathbb{I}_u) \tag{4.1}$$

$$\mathbb{I}_s = \mathbb{I}_{[-2^{31}, 2^{31}-1]} \tag{4.2}$$

$$\mathbb{I}_u = \mathbb{I}_{[0, 2^{32}-1]} \tag{4.3}$$

which models the interpretation of the registers' contents in these three domains. $\mathbb{B}_1$ is the set-based lattice of bit contents as shown in Figure 4.7a whereas $\mathbb{I}_{[l,u]}$ is the lattice of sub-intervals of $[l, u]$ ordered by inclusion as shown in Figure 4.7b. The meet operator $\sqcup : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ and the partial order $\sqsubseteq : \mathbb{V} \times \mathbb{V} \to \mathbb{B}$ are formed by piecewise application of the meet and partial order operators of the sub-lattices.

The transfer function $f_v : \mathbb{V} \to \mathbb{V}$ for a context block $v \in V_\tau^C$ iterates over the instructions in the block and applies their effect to the value state. Each instruction's effect on a register is defined to work on a single sub-lattice, e.g., a shift operation will work on the bit-vector sub-lattice and a signed addition will work on the signed interval sub-lattice. Once the respective sub-lattice value was updated, the results are transferred to the other domains by using *conversion functions* $\alpha_S^T : S \to T$ where $S, T \in \{\mathbb{B}_1^{32}, \mathbb{I}_s, \mathbb{I}_u\}$ are the source and target sub-lattice. For all S and T, α_S^T and α_T^S form a Galois connection (see Definition 3), i.e., $\forall s \in S : \alpha_T^S\left(\alpha_S^T(s)\right) \sqsupseteq s$. We do not detail the individual transfer functions here, but the ARMv4 architecture has 91 distinct instructions with up to 11 addressing modes that must be handled.

To speed up the convergence of the signed integer intervals we employ a widening with a the set I of all signed integer constants found in the program code and the maximum and minimum signed integer values -2^{31} and $2^{31} - 1$. The widening operator $\Delta : \mathbb{I}_s \to \mathbb{I}_s$ is then defined as

$$\Delta([a, b]) = [\max(\{i \mid i \in I, i \leq a\}), \min(\{i \mid i \in I, i \geq b\})] \tag{4.4}$$

An equivalent widening is defined for the unsigned case. For the bit vectors, widening is not needed, since the height of $\mathbb{B}_1$ is two, such that the height of $\mathbb{B}_1^{32}$ is 64, therefore any element from this lattice can only undergo 63 changes until it reaches the fixed point $\top$. In contrast, the interval lattices have height 2^{32}, which is still finite, but makes the convergence really slow.[2] In addition, we perform narrowing until each block was visited twice to re-gain precision that was lost due to the widening.

4.2.2 Challenges of Predicated Execution

The predicated execution as already mentioned in the analysis graph definition at Section 4.1.1 works by first setting a flag and then performing actions based on it. As an example, in block a_1 from Figure 4.3, the instruction `cmp r3, #8` sets the `EQ`-flag to `true` iff `r3` is equal to 8. The value analysis might be unable to infer a concrete value of `r3` at this point, i.e., $val^{\text{in}}_{a_1}(\texttt{r3})$ might be $\top$ and a naive implementation will therefore not be able to infer anything about the value of `r3` at a_2 either. Obviously, we know at a_2 that $val^{\text{in}}_{a_2}(\texttt{r3}) = [8,8]$ due to the implication set up with the preceding comparison. We can supply the value analysis with the possibility to infer this by adding *flag implications* to the value domain, i.e.,

$$\mathbb{V} = (R \to (\mathbb{B}^{32}_1 \times \mathbb{I}_s \times \mathbb{I}_u)) \times (F \times \{0,1\} \times R \times \mathbb{I}_s) \tag{4.5}$$

where F is the set of flag bits of the machine. Each implication entry (f,b,r,w) at a node v denotes that

$$f = b \implies val^{\text{in}}_v(r) \sqsubseteq w \tag{4.6}$$

This means that in each transfer function invocation, we may *restrict* the incoming value set $val^{\text{in}}_v(r)$ to the lattice *infimum* $val^{\text{in}}_v(r) \sqcap w$ iff there is a flag implication for r. The transfer functions are also responsible for registering new flag implications that are set up by instructions like `cmp` and for removing the implication when the base register r is overwritten with a new value. The basic idea of exploiting conditions to refine the variable values has already been used before, but here we adapted it to conditions which are only given as predication entries.

The meet function is applied to each $f \in F$ and $b \in \{0,1\}$ in separation as

$$\sqcup((f,b,r_1,w_1),(f,b,r_2,w_2)) = \begin{cases} (f,b,r_1,w_1 \sqcup w_2) & \text{if } r_1 = r_2 \\ (f,b,r_1,\top) & \text{else} \end{cases} \tag{4.7}$$

With this extension, the implications in $val^{\text{out}}_{a_1}$ will contain the entry $(\texttt{EQ},1,\texttt{r3},[8,8])$ such that at the beginning of f_{a_2} we can set

$$\begin{aligned} val^{\text{out}}_{a_2}(\texttt{r3}) &= val^{\text{in}}_{a_2}(\texttt{r3}) \sqcap [8,8] \\ &= \top \sqcap [8,8] \\ &= [8,8] \end{aligned}$$

[2] In Section 2.1 we have stated that the integer value lattice has infinite height. This is only true for high-level languages where integers of arbitrary length are supported. On the machine code level, every analysis domain is naturally limited by the register or memory size specification.

In general, the value analysis with flag implications is able to recognize bounds on the direct and derived induction variables in counting loops which are very frequent in embedded system code. A yet higher precision would be achieved by *relational congruences* [Cou01] but usually, convex abstractions like intervals are sufficient for the microarchitectural analysis.

4.3 Microarchitectural Analysis

The purpose of the microarchitectural analysis is to determine the possible execution duration for every block in a context graph. To be able to determine such durations, first the operation of the processor must be correctly formalized. We base our following introduction to the formal pipeline description on [Wil12], whereas a thorough treatment of concrete pipeline implementation techniques can be found in [GLM11].

Definition 15. *A* processor pipeline *is a finite-state machine* $\tilde{P} = (\tilde{Q}_P, \tilde{I}, \tilde{O}, \tilde{\delta}_P, \tilde{\lambda}_P)$ *with state set* $\tilde{Q}_P$*, input set* $\tilde{I}$*, output set* $\tilde{O}$*, state transfer function* $\tilde{\delta}_P : \tilde{Q}_P \times \tilde{I} \to \tilde{Q}_P$ *and output function* $\tilde{\lambda}_P : \tilde{Q}_P \times \tilde{I} \to \tilde{O}$*. Each state transition of* P *models a single clock cycle of the pipeline.* P *interacts with an* environment E *which can itself be represented as a state machine* $\tilde{E} = (\tilde{Q}_E, \tilde{O}, \tilde{I}, \tilde{\delta}_E, \tilde{\lambda}_E)$*.*

A concrete microarchitectural state *is an element of* $\tilde{Q}_M = \tilde{Q}_P \times \tilde{Q}_E$*, which forms the* concrete microarchitectural lattice $(\tilde{\mathbb{M}} = 2^{\tilde{Q}_M}, \cup)$*.*

The environment models the memory hierarchy components like caches and memories which contain the program P amongst others. We assume a deterministic execution without preemptions, therefore E is a deterministic FSM, too.

Definition 16. *The* concrete execution *of a program* $L = (i_0, \dots, i_n)$ *with start instruction* i_0 *and a set of terminal instructions* I_t *starting at an initial state* $\tilde{q}_0^p \in \tilde{Q}_P$, $\tilde{q}_0^e \in \tilde{Q}_E$ *is modeled by* $exec(L, \tilde{q}_0^p, \tilde{q}_0^e) = ((\tilde{q}_1^p, \tilde{q}_1^e), \dots, (\tilde{q}_k^p, \tilde{q}_k^e))$ *with*

$$\forall i > 0 : (\tilde{q}_i^p, \tilde{q}_i^e) = (\tilde{\delta}_P(\tilde{\lambda}_E(\tilde{q}_{i-1}^e)), \tilde{\delta}_E(\tilde{\lambda}_P(\tilde{q}_{i-1}^p))) \tag{4.8}$$

where state $\tilde{q}_0^p$ *is required to fetch* i_0 *from the environment state* $\tilde{q}_0^e$ *and the* trace $((\tilde{q}_1^p, \tilde{q}_1^e), \dots, (\tilde{q}_m^p, \tilde{q}_m^e))$ *models the execution of instructions until a terminal instruction* $i_t \in I_t$ *has been retired in* $(\tilde{q}_k^p, \tilde{q}_k^e)$*. Here,* retirement *means that the instruction has finally left the pipeline and its effect was made permanent.*

When we are not interested in distinguishing pipeline and environment state we abbreviate the microarchitectural state to $\tilde{q}_i^m = (\tilde{q}_i^p, \tilde{q}_i^e)$ *for all* i*. The exec function then becomes*

$$exec(L, \tilde{q}_0^m) = (\tilde{q}_1^m, \dots, \tilde{q}_k^m) \tag{4.9}$$

Since any basic block $b = (i_0, \dots, i_n)$ is also a valid program with a single terminal instruction i_n, we can determine its *concrete execution duration* from a given initial state $(\tilde{q}_0^p, \tilde{q}_0^e)$ as $|exec(b, \tilde{q}_0^p, \tilde{q}_0^e)| \in \mathbb{N}_0$. The duration may actually be zero, since in a superscalar processor state, all instructions of the block may have already been retired in parallel to the execution of the previous block. Obviously, we need to know about the possible valid initial states $\tilde{q}_0^p$ and $\tilde{q}_0^e$ at this point to derive the minimum and maximum execution durations.

These states can in principle be computed using abstract interpretation on $(\tilde{\mathbb{M}}, \cup)$. To achieve this we need to set the transfer function f_v^M for a context graph node $v = ((i_0, \dots, i_n), \cdot, \cdot)$ to

$$\forall m \in \tilde{\mathbb{M}} : f_v^{\tilde{\mathbb{M}}}(m) = \bigcup_{(\tilde{q}^m) \in m} \{\tilde{q}_k^m \mid \mathit{exec}((i_0, \dots, i_n), \tilde{q}^m) = (\tilde{q}_0^m, \dots, \tilde{q}_k^m)\} \tag{4.10}$$

where we use the notation from Equation 4.9 and $\tilde{q}_k^m$ is the state in which the last instruction of block v was retired. With these transfer functions, $((\tilde{\mathbb{M}}, \cup), \bigcup_{v \in V_\tau^C} \{f_v^{\tilde{\mathbb{M}}}\})$ is a monotone DFA framework (cf. Definition 5) which can be solved by fixed-point iteration.

Still, the concrete FSMs $\tilde{P}$ and $\tilde{E}$ have far to many states. To make the DFA solution computable we abstract $\tilde{P}$ and $\tilde{E}$ to *abstract state machines*.

Definition 17. *An* abstract pipeline model $P = (Q_P, I, O, \delta_P, \lambda_P)$ *and* abstract environment model $E = (Q_E, O, I, \delta_E, \lambda_E)$ *for a pipeline* $\tilde{P}$ *and environment* $\tilde{E}$ *are non-deterministic FSMs, i.e.,* $\delta_P : Q_P \times I \to 2^{Q_P}$, $\lambda_P : Q_P \times I \to 2^O$ *and* $\delta_E : Q_E \times O \to 2^{Q_P}$, $\lambda_P : Q_P \times O \to 2^I$. *In analogy to Definition 15, each state transition in* P *and* E *corresponds to one cycle step of the modeled pipeline.*

An abstract microarchitectural state *is an element of* $Q_M = Q_P \times Q_E$, *which forms the* abstract microarchitectural lattice $(\mathbb{M} = 2^{Q_M}, \cup)$. *Since each abstract state* $q^m \in \mathbb{M}$ *represents a number of concrete states* $\tilde{q}^m \in \tilde{\mathbb{M}}$, $(\mathbb{M}, \cup)$ *and* $(\tilde{\mathbb{M}}, \cup)$ *must be connected through a Galois connection* $\alpha_{\mathbb{M}} : \tilde{\mathbb{M}} \to \mathbb{M}$ *and* $\gamma_{\mathbb{M}} : \mathbb{M} \to \tilde{\mathbb{M}}$. *The input (output) sets* $\tilde{I}$ *and* I *(*$\tilde{O}$ *and* O*) must be covered by a Galois connection, in the same way.*

Finally, P *and* E *must be* valid *with respect to the Galois connections, i.e.,*

$$\forall \tilde{q}^p \in \tilde{Q}_P, \tilde{i} \in \tilde{I} : \tilde{\delta}_P(\tilde{q}^p, \tilde{i}) \in \gamma_{\mathbb{M}}(\delta_P(\alpha_{\mathbb{M}}(\tilde{q}^p), \alpha_{\mathbb{M}}(\tilde{i}))) \tag{4.11}$$
$$\forall \tilde{q}^p \in \tilde{Q}_P, \tilde{i} \in \tilde{I} : \tilde{\lambda}_P(\tilde{q}^p, \tilde{i}) \in \gamma_{\mathbb{M}}(\lambda_P(\alpha_{\mathbb{M}}(\tilde{q}^p), \alpha_{\mathbb{M}}(\tilde{i}))) \tag{4.12}$$
$$\forall \tilde{q}^e \in \tilde{Q}_E, \tilde{o} \in \tilde{O} : \tilde{\delta}_E(\tilde{q}^e, \tilde{o}) \in \gamma_{\mathbb{M}}(\delta_E(\alpha_{\mathbb{M}}(\tilde{q}^e), \alpha_{\mathbb{M}}(\tilde{o}))) \tag{4.13}$$
$$\forall \tilde{q}^e \in \tilde{Q}_E, \tilde{o} \in \tilde{O} : \tilde{\lambda}_E(\tilde{q}^e, \tilde{o}) \in \gamma_{\mathbb{M}}(\lambda_E(\alpha_{\mathbb{M}}(\tilde{q}^e), \alpha_{\mathbb{M}}(\tilde{o}))) \tag{4.14}$$

In the transition from $\tilde{P}$ to P, we remove all modeling of value changes of registers and main memory cells and all component-internal state like arithmetic unit states. The value changes are partly covered by the value analysis and the rest is abandoned to limit the complexity. In the environment, we remove the modeling of memory cell values and memory hierarchy component state which does not affect the timing. As a consequence of this reduction, the state transitions in P and E are non-deterministic. This non-determinism is needed to resolve situations in which the successor or output in the concrete FSM depends on a state component which is not modeled in the abstract FSM like, e.g., the program input values. Therefore, the abstract execution must gather all reachable abstract states.

Definition 18. *The* abstract execution *of a program $L = (i_0, \dots, i_n)$ with start instruction i_0 and a set of terminal instructions I_t starting at an abstract state $q_0^m \in Q_M$ is modeled by $exec(L, q_0^m) = (m_1, \dots, m_k)$ with $m_i \subseteq Q_M$ and*

$$\forall i > 0 : m_i = \bigcup_{\substack{(q_{i-1}^p, q_{i-1}^e) \in m_{i-1} \text{ with} \\ \neg retired((q_{i-1}^p, q_{i-1}^e))}} \{(q_i^p, q_i^e) = (\delta_P(\lambda_E(q_{i-1}^e)), \delta_E(\lambda_P(q_{i-1}^p)))\} \tag{4.15}$$

where retired : $Q_M \to \{true, false\}$ determines whether an instruction from I_t was retired in a given abstract state. Therefore, each set m_i holds those states which are reachable in the i-th cycle of an execution starting at any concrete initial state $\tilde{q}_0^m \in \gamma_{\mathbb{M}}(q_0^m)$. The states in which the program may have been completed in the abstract execution are given by

$$compstates(L, q_0^m) = \{q_c^m \mid \exists m_i \in exec(L, q_0^m) : q_c^m \in m_i \wedge retired(q_c^m)\} \tag{4.16}$$

The transfer functions on $\mathbb{M}$ are then defined as

$$\forall m \in \mathbb{M} : f_v^{\mathbb{M}}(m) = \bigcup_{q^m \in m} compstates(L, q^m) \tag{4.17}$$

In this way, we can compute the possible initial abstract microarchitectural states for each block in a context graph by data-flow analysis on the DFA framework $((\mathbb{M}, \cup), \bigcup_{v \in V_\tau^C} \{f_v^{\mathbb{M}}\})$.

Definition 19. *Given a context block $v = ((i_1, \dots, i_n), \cdot, \cdot)$ and a set of possible initial hardware states $q_v^{in} \in \mathbb{M}$, the* block duration *$\omega(v) \in \mathbb{I}_{[0,\infty]}$ is*

$$\omega(v) = [min(T), max(T)] \tag{4.18}$$

$$T = \{i \mid \exists m_i \in exec(L, q_0^m) : q_c^m \in m_i \wedge retired(q_c^m)\} \tag{4.19}$$

The lower (upper) bound of $\omega(v)$ is denoted as $\omega_{min}(v)$ ($\omega_{max}(v)$).

An example for the analysis of a block which yields a value of $\omega(v) = [2, 5]$ is given in Figure 4.8. The figure illustrates the computation of $q_v^{\text{out}} = f_v^{\mathbb{M}}(q_v^{\text{in}})$, where each edge represents one cycle step of the abstract microarchitectural states. To compute q_v^{out}, the cycle step must be invoked for all initial states and their successors until all instructions from v are retired at every sink of the resulting transition graph. The set of sinks is then called q_v^{out}. As a side product from this state computation, we can derive the execution time bound $\omega(v)$ which can be used in the path analysis to compute the longest (shortest) path through the program and by that the WCET (BCET).

In a timing-anomaly-free architecture as defined in Section 2.2.8, we may restrict the output of the transfer function to those states which correspond to the local

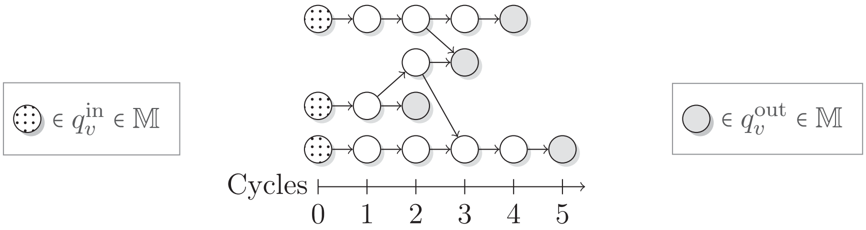


Figure 4.8: Example abstract microarchitectural states $q^m \in Q_M$ during the analysis of a context block $v \in V_\tau^C$ with $\omega(v) = [2, 5]$. Each state transition corresponds to one cycle step of the modeled pipeline.

worst-case successors. In that case, we are no longer building an approximation of *every* possible microarchitectural state with which a block may be entered but of those states with which it may be entered in an execution that *exclusively* consists of local worst-case transitions. In a timing-anomaly-free system this is sufficient for finding the WCET, since then we know that the global worst-case execution *does* only consist of local worst-case transitions. Formally, the transfer function for timing-anomaly-free systems becomes

$$\forall m \in \mathbb{M} : f_v^{\mathbb{M}}(m) = \bigcup_{q^m \in m} \{q_c^m \mid m_k \in \mathit{exec}(L, q^m) : q_c^m \in m_k \wedge \mathit{retired}(q_c^m)\} \tag{4.20}$$

i.e., we only draw the completion states from the longest traces, whose length is given by k as in Definition 18. For the example in Figure 4.8, this implies that only the gray states at time 4 and 5 would be part of q_v^{out}, since for the topmost initial state the longest execution path ends at time 4 and for the other two initial states the longest execution path ends at time 5. Therefore, we could derive a tighter execution time window of $\omega(v) = [4, 5]$ in a timing-anomaly-free system. Obviously, once we start to exploit the timing-anomaly-freedom to cut down the microarchitectural search space in this way, the resulting ω values are only guaranteed to be valid for the worst-case path. Therefore, we can no longer use them to determine a BCET of the task under analysis.

This FSM-based modeling approach has been applied successfully in many static WCET analyzers including the commercial product aiT. The microarchitectural state can then be represented as an element of 𝕄 as sketched above [SF99; LTH02; The04]. For complex microarchitectures, the state space that must be explored for a single block (as shown in Figure 4.8) can become quite big which may lead to higher analysis times. To avoid this, symbolic representations of the state transfer function [Wil12] can be used to avoid the state enumeration from Figure 4.8. A minor other direction of microarchitectural modeling is the representation of the pipeline structure by a directed graph [LRM06], but this approach integrates far less easily with an analysis of the environment (e.g., memory hierarchy elements) and with out-of-order processors where basic block executions may overlap.

4.3.1 ARM7TDMI Pipeline Model

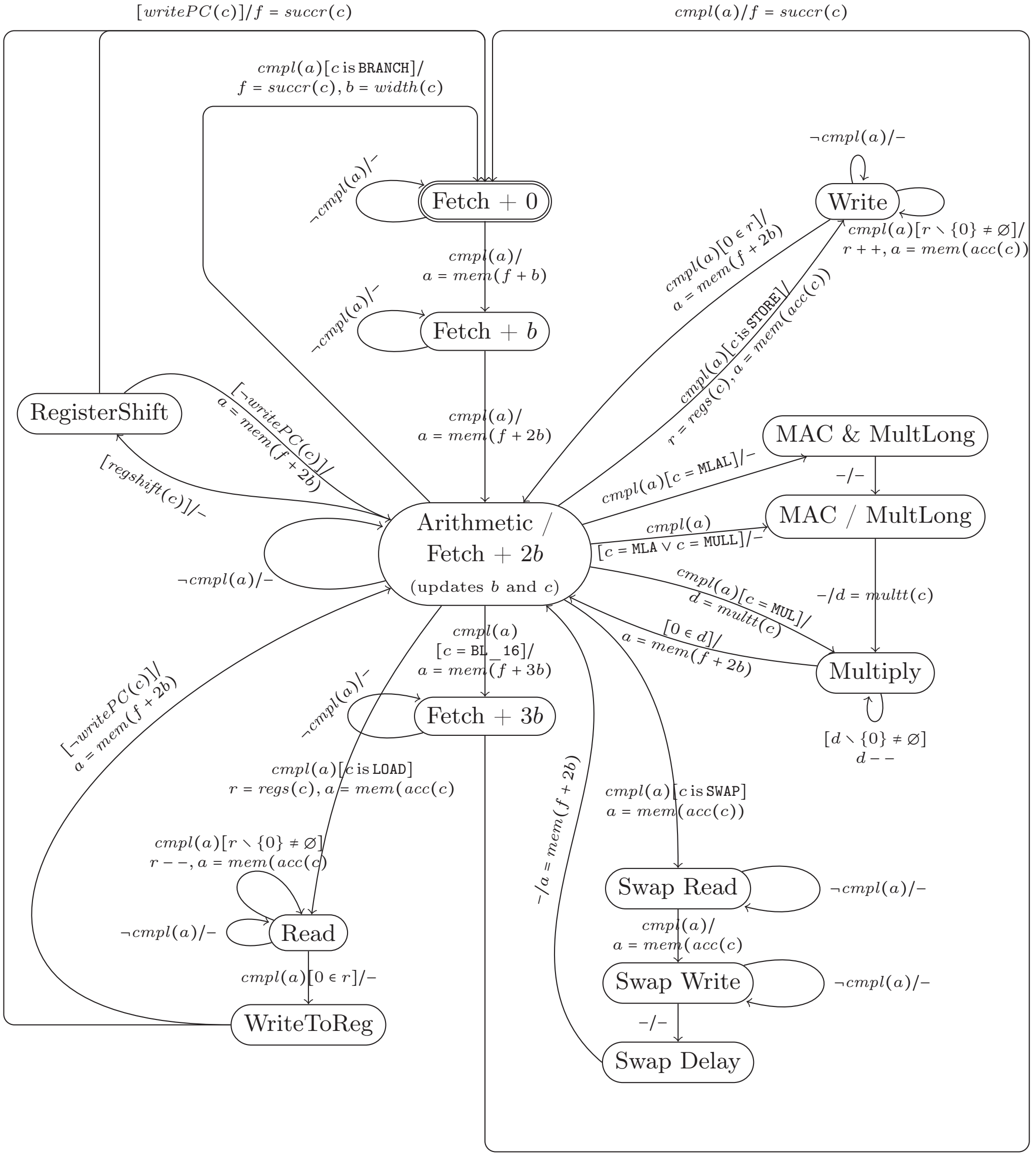
As stated, the construction of the abstract pipeline model $P = (Q_P, I, O, \delta_P, \lambda_P)$ consists of a slicing of the concrete state space into timing-relevant and non-relevant components. There are approaches which create P semi-automatically from formal hardware descriptions [SP10], but most pipeline models are generated manually by careful inspection of formal hardware descriptions, data sheets or by conducting extensive measurements on real hardware.

In the case of the ARM7TDMI, the timing is precisely specified in its technical manual [ARM04], which drastically eases the derivation of P. Figure 4.9 shows the basic structure of the timing model of the ARM7TDMI. We use an extended FSM model here, where the FSM state holds a fixed amount of finite-sized variables. Transitions have *guards* (marked in square brackets) and *actions* on the FSM variables to make the model more compact. Nevertheless, we can expand this modeling to a classical FSM due to the fixed size of the FSM variables.

The core has an in-order pipeline which is refilled each time a branch is taken. The initial filling of the pipeline as well as branch-induced refilling is modeled by the chain of "Fetch + X" states which increment the address from which the next instruction is fetched by the fetch width b. Since the ARM7TDMI supports both 32-bit ARM as well as 16-bit THUMB instructions, b is equal to either 2 or 4 bytes. Once the pipeline is filled, most instructions are executed in one cycle and the pipeline keeps on fetching new instructions and executing one-cycle instructions in parallel which is modeled by the self-loop at "Arithmetic / Fetch + 2b". However, the duration of multiplications may vary by up to 3 cycles as shown by the multiplication states on the right hand side. The duration depends on the value of the second operand of the multiplication. If the value analysis can precisely determine the operand, then we are able to provide a precise $multt(c)$ interval. For the example of a `MLAL` (multiply-accumulate on long word) instruction, this means that in the best case we get an estimation of $multt(c) = [3, 3]$, whereas in the worst case we fall back to $multt(c) = [3, 6]$. Similarly, the accessed address of a load, store or swap instruction must be determined in the helper function $acc(c)$ by reading the value analysis results. Some of the memory-accessing instructions store or load multiple registers at once for an efficient stack handling. The number of affected registers is encoded in the parameter $r = regs(c)$ in P.

Finally, the non-determinism in the operation of P can be observed at states which have transitions of which multiple may be ready at the same point in time. As an example, when the duration $d = multt(c)$ is equal to $[0, 1]$ at the "Multiply" state due to insufficient value analysis precision, the guard conditions $[0 \in d]$ and $[d \setminus \{0\} \neq \varnothing]$ are both true at the same time. In this case, the analysis must explore all possible result states as shown in Figure 4.8.

The $mem(c)$ function can only determine the duration of a memory access if it interacts with the *abstract environment* $E = (Q_E, O, I, \delta_E, \lambda_E)$ as defined in the beginning of Section 4.3. It therefore has to issue an abstract memory request $o \in O$



Field	Type	Description
a	O (Output)	Current access
c	I (Input)	Current instruction
$cmpl$	I (Input)	Access completions
f	$\mathbb{I}_u$	Addresses to fetch from
b	$\mathbb{I}_u$	Possible fetch width
d	$\mathbb{I}_u$	Remaining cycles in state
r	$\mathbb{I}_u$	Number of registers to be processed

Function	Description
$mem(c)$	Issue memory access (Output)
$acc(c)$	Possible access range of c
$width(c)$	Possible fetch widths at c
$multt(c)$	Multiplication duration at c
$regs(c)$	Number of registers processed by c
$succr(c)$	Address range of successors of the current basic block
$writePC(c)$	Whether c writes to the PC

Figure 4.9: The abstract pipeline model for the ARM7TDMI.

to the environment which the environment must respond to by signaling $cmpl(a)$ in its future output and by altering its state q^e, e.g., if the access touches a cache. As mentioned, the domain of the microarchitectural analysis is

$$\mathbb{M} = 2^{Q_M} = 2^{Q_P \times Q_E} \tag{4.21}$$

i.e., to determine the runtime of a memory access, each pipeline state $q_p \in Q_P$ interacts with its assigned environment state $q_e \in Q_E$.

To summarize the pipeline aspects, if $Q_P^{\#}$ is the set of FSM states from Figure 4.9, we can now formally define Q_P for the ARM7TDMI pipeline as

$$Q_P = Q_P^{\#} \times (\mathbb{I}_u)^4 \tag{4.22}$$

where the $\mathbb{I}_u$-components represent the FSM variables f, b, w and r from Figure 4.9.

For this extended FSM model, we also must adapt the meet function of the microarchitectural lattice. Previously, the meet function was the set union of the contained states, but here we may have elements with identical FSM state but different FSM variable values. Therefore, the meet operator on two elements $q_1^{\mathbb{M}}, q_2^{\mathbb{M}} \in \mathbb{M}$ is defined as

$$q_1^{\mathbb{M}} \sqcup q_2^{\mathbb{M}} = \bigcup_{q^{\#} \in Q_P^{\#}} \sqcup(\{q^m \mid q^m = ((q^{\#}, \dots), q^e) \in (q_1^{\mathbb{M}} \cup q_2^{\mathbb{M}})\}) \tag{4.23}$$

where $(q_1^p, q_1^e) \sqcup (q_1^p, q_1^e) = (q_1^p \sqcup q_2^p, q_1^e \sqcup q_2^e)$ and for all $q_1^p = (q^{\#}, f_1, b_1, w_1, r_1) \in Q_P$ and $q_2^p = (q^{\#}, f_2, b_2, w_2, r_2) \in Q_P$

$$q_1^p \sqcup q_2^p = \{(q^{\#}, f_1 \sqcup f_2, b_1 \sqcup b_2, w_1 \sqcup w_2, r_1 \sqcup r_2)\} \tag{4.24}$$

Thus, we exploit the fact that the FSM variables are all intervals which can be easily merged to reduce the state space size. Note that we could even go one step further into this direction by setting $Q_P = 2^{Q_P^{\#}} \times (\mathbb{I}_u)^4$. In this case we would lose every connection between FSM states $q^{\#} \in Q_P^{\#}$ and the FSM variables which allows us to represent even more concrete states with a single abstract one at the cost of reduced analysis precision. To keep up a high precision, we did not pursue this approach.

The environment model must include all microarchitectural components like buses, caches, memories and peripherals. In practice, usually neither SRAM memory modules nor peripherals are modeled in detail. For SRAM, a fixed access time (or a very small time window) is given which makes a more precise analysis of their behavior not necessary, since we are only concerned with time, here. Accesses to peripherals are rare enough that we can assume the full time window (best-case to worst-case) on each access.

For buses, at which the current core definitely is the sole master and thus will be granted the bus immediately, also few modeling is needed. The only thing that will regularly happen here, is that the target address of the access cannot be determined precisely, e.g. due to coarse results in the value analysis. In this situation, the transition in the bus module is non-deterministic, and all possible access targets have to explored separately as shown in Figure 4.8.

One important factor that *is* modeled in the environment state, however, are *caches* since they show both high execution time variation and access frequency. This makes a precise analysis of the cache behavior inevitable if the generated $\omega(v)$-values shall not be overly conservative.

4.3.2 Cache Analysis

The first cache models incorporated the analysis of direct-mapped caches into the path analysis [LBJ+95; LMW96]. Since these approaches were neither scalable nor could they be integrated with the rest of the microarchitectural modeling as sketched above, an efficient abstract model for set-associative caches with *Least Recently Used* (LRU) replacement was first devised in [FW99]. This approach can classify a cache access in a context block as "always hit (AH)", "always miss (AM)" or "unknown (U)". It was later refined to include the classification "first-access-miss-all-others-hit", also known as "persistent" [BC08].

The *First-In First-Out* (FIFO) replacement policy has proven to be much harder to analyze [GR10] and to consistently perform worse than LRU in terms of the necessary number of replacements [CN98]. FIFO and *Pseudo-LRU* (PLRU) caches can be analyzed using the concept of *relative competitiveness* [BRA09], i.e., their analysis can be reduced to the analysis of an appropriately shrunk LRU cache. This emphasizes the usefulness of LRU analysis.

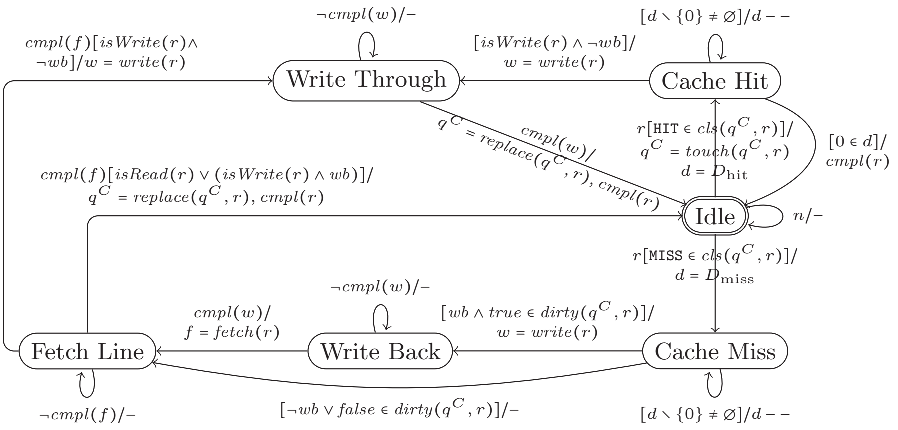
A basic abstract model of the cache operation can be seen in Figure 4.10. Accesses to the cache modeled by the input values r and n are classified with respect to the abstract cache state $q^C \in C$ via a classification function $cls(q^C, r) \subseteq \{\texttt{HIT}, \texttt{MISS}\}$ that determines whether the access must be a hit or a miss or may be both. Upon a miss, the content is fetched by issuing memory accesses to the component to fetch from in $fetch(r)$. Finally, the completion of the original request r is signaled to the source (i.e., the pipeline) by $cmpl(r)$. Note, that the cache content is *not* part of our abstract cache model, we therefore only communicate the completion without knowing what data is actually transferred by the cache.

The *cache analysis domain* which is part of the environment state Q_E is therefore defined as

$$\mathbb{C} = 2^{Q_C} \times C \tag{4.25}$$

where Q_C is the set of states in the abstract FSM from Figure 4.10 and C is the domain of abstract cache states as detailed in the following.

For a set-associative cache, the state of the cache $q^C \in C$ is usually maintained for each cache set $s \in S$ independently [FW99] as a *set state* $q^C_s \in q^C$. Alternatively, the cache state can be represented as a tuple of explicit set states [CR09]. This results in higher precision but also far higher analysis duration, which is why we use the former approach from [FW99]. An abstract memory request r is mapped to the set of cache sets which it may affect by an *abstract assignment function* $sets(r) \subseteq S$.


¬cmpl(w)/–
cmpl(f)[isWrite(r)∧ ¬wb]/w = write(r)
Write Through
[isWrite(r) ∧ ¬wb]/ w = write(r)
[d ∖ {0} ≠ ∅]/d − −
Cache Hit
q^C = replace(q^C, r), cmpl(r)
cmpl(w)/
r[HIT ∈ cls(q^C, r)]/ q^C = touch(q^C, r) d = D_hit
[0 ∈ d]/ cmpl(r)
cmpl(f)[isRead(r) ∨ (isWrite(r) ∧ wb)]/ q^C = replace(q^C, r), cmpl(r)
Idle
n/–
r[MISS ∈ cls(q^C, r)]/ d = D_miss
¬cmpl(w)/–
cmpl(w)/ f = fetch(r)
[wb ∧ true ∈ dirty(q^C, r)]/ w = write(r)
Fetch Line
Write Back
Cache Miss
¬cmpl(f)/–
[¬wb ∨ false ∈ dirty(q^C, r)]/–
[d ∖ {0} ≠ ∅]/d − −

Field	Type	Description
n	Input	No access
r	Input	Memory access r
cmpl	Input	Request completions
d	$\mathbb{N}_0$	Processing delay
q^C	C	Abstract cache state
wb	$\{true, false\}$	Write policy

Function	Description
cls	Classify access
touch	Update cache state on hit
replace	Update cache state on eviction
dirty	Determine if line may be dirty
write	Issue write request (Output)
fetch	Issue fetch request (Output)
cmpl	Signal access completion (Output)

Figure 4.10: A simplified abstract cache model. One idle cycle is enforced between successive accesses to reduce the graph size.

Since the address of an access as determined by the value analysis may be unknown, $sets(r) = S$ is possible[3], but for a fixed address, we have $|sets(r)| = 1$. Analogously, the possible *tags* of the memory block that is targeted by access r are given by $tags(r) \subseteq T$ where T is the set of all tags. Therefore, we can define the cache state functions by delegating the work to the set states as

$$cls(q^C, r) = \bigcup_{s \in sets(r), t \in tags(r)} cls(q^C_s, t) \tag{4.26}$$

$$touch(q^C, r) = \bigsqcup_{s \in sets(r), t \in tags(r)} touch(q^C, s, t) \tag{4.27}$$

$$replace(q^C, r) = \bigsqcup_{s \in sets(r), t \in tags(r)} replace(q^C, s, t) \tag{4.28}$$

Every q^C_s must respect the replacement policy of the cache which we assume to be LRU as mentioned above. A concrete LRU set state $\tilde{q}^C_s$ for an associativity $\hat{a}$ is

[3] In general, this is one crucial dependency of the cache analysis due to which also dynamic memory allocation is usually prohibited since the resulting addresses are unpredictable. Approaches to circumvent this are cache-aware allocators [HBH+11] and *relational cache analysis* [HG12; Weg12] which works on an *access position relation* instead of on absolute address values.

an *age function* $T_s \to A$ that maps a subset $T_c \subseteq T$ to an age set $A = \{1, \dots, \hat{a}, \infty\}$ with the concrete semantics

$$cls(\tilde{q}_s^C, t) = \begin{cases} \texttt{HIT} & \text{if } \tilde{q}_s^C(t) \neq \infty \\ \texttt{MISS} & \text{else} \end{cases} \tag{4.29}$$

$$touch(\tilde{q}^C, s, t) = \bigcup_{s' \in S} \begin{cases} \tilde{q}_s^{C\prime}(t') = \begin{cases} 1 & \text{if } t' = t \\ \tilde{q}_s^C(t') + 1 & \tilde{q}_s^C(t') < \tilde{q}_s^C(t) \\ \tilde{q}_s^C(t') & \text{else} \end{cases} & \text{if } s = s' \\ \tilde{q}_s^C & \text{else} \end{cases} \tag{4.30}$$

$$replace(\tilde{q}^C, s, t) = \bigcup_{s' \in S} \begin{cases} \tilde{q}_s^{C\prime}(t') = \begin{cases} 1 & \text{if } t' = t \\ \tilde{q}_s^C(t') + 1 & \text{else} \end{cases} & \text{if } s = s' \\ \tilde{q}_s^C & \text{else} \end{cases} \tag{4.31}$$

This concrete semantics is abstracted to an *age range* for each tag, i.e., $\forall t \in T :$ $q_s^C(t) = [a_{min}^t, a_{max}^t]$ such that $a_{min}^t, a_{max}^t \in \{1, \dots, a, \infty\}$ and a_{min}^t (a_{max}^t) is the minimum (maximum) age of tag t at the current block. a_{min}^t (a_{max}^t) is also called *may*-information (*must*-information) since it can be used to determine what may (must) be in the cache as shown in the following definition of $cls(q^C, t)$

$$cls(q_s^C, t) = \begin{cases} \{\texttt{HIT}\} & \text{if } a_{max}^t \neq \infty \\ \{\texttt{MISS}\} & \text{if } a_{min}^t = \infty \\ \{\texttt{HIT}, \texttt{MISS}\} & \text{else} \end{cases} \tag{4.32}$$

The *touch* and *replace* functions coincide in this domain ($replace = touch$) with

$$touch(q^C, s, t) = \bigcup_{s' \in S} \begin{cases} q_s^{C\prime}(t') = [a_{min}^{t' \to t}, a_{max}^{t' \to t}] & \text{if } s = s' \\ q_s^C & \text{else} \end{cases} \tag{4.33}$$

$$a_{min}^{t' \to t} = \begin{cases} 1 & \text{if } t = t' \\ a_{min}^{t'} + 1 & \text{if } a_{min}^{t'} \leq a_{min}^t \\ a_{min}^{t'} & \text{else} \end{cases} \tag{4.34}$$

$$a_{max}^{t' \to t} = \begin{cases} 1 & \text{if } t = t' \\ a_{max}^{t'} + 1 & \text{if } a_{max}^{t'} < a_{max}^t \\ a_{max}^{t'} & \text{else} \end{cases} \tag{4.35}$$

where the addition on ages is again only defined on A, i.e., $\hat{a} + 1 = \infty$. The meet function on $\mathbb{C}$ is then defined as the interval union, i.e.,

$$q_1^C \sqcup q_2^C(s)(t) = [\min(q_1^C(s)(t), q_2^C(s)(t)), \max(q_1^C(s)(t), q_2^C(s)(t))] \tag{4.36}$$

The initial DFA information for the start node is the top element $\top^{\mathbb{C}}$ of $\mathbb{C}$, i.e., (Q_C, q_{init}^C) with $\forall s \in S, t \in T : q_{\text{init}}^C(s)(t) = [0, \infty]$. In the implementation, we obviously do not want to represent all age intervals explicitly, since many entries will be identical. As an example, *all* entries in q^C_{init} are identical. We therefore use *interval maps* to compress ranges of sets and tags with identical abstract information to a single value.

Multiple cache levels can be handled by directing the output of the abstract L1 cache to the abstract L2 cache which may then in turn perform non-deterministic state transitions based on the forwarded request. If the cache hierarchy is analyzed in separation from the rest of the system (i.e., pipeline, busses), multiple cache levels can more efficiently be analyzed by computing *cache access classifications (CAC)* per hierarchy level [HP08; LHP09], but this requires a compositional timing model as discussed in Section 2.2.9.

4.4 Path Analysis

For a program L and a set of initial system states $\tilde{Q}_0 \subseteq \tilde{Q}_M$, we have

$$\text{WCET}_{\text{real}} = \max_{q^m_0 \in \tilde{Q}_0} \{|exec(L, q^m_0)|\} \tag{4.37}$$

Usually, $\tilde{Q}_0$ contains all possible program inputs and one or more initial hardware states. The goal of the path analysis is to derive a $\text{WCET}_{\text{est}} \geq \text{WCET}_{\text{real}}$ with the help of the block durations $\omega(v) \in \mathbb{I}_{[0,\infty]}$ generated by the microarchitectural analysis. The first step towards this is to define an alternative notation for a program execution as a sequence of context blocks and concrete block durations, i.e.,

$$bbexec(L, q^m_0) = ((v^C_1, w_1), \dots, (v^C_k, w_k)) \tag{4.38}$$

as opposed to $exec(L, q^m)$ from Definition 16 which defines an execution as a sequence of system states q^m_i. This can easily be achieved by mapping each contiguous sequence of system states q^m_i with length w which “belong” to the same block v^C along the lines of Definition 19 to an element (v^C, w).

From the correctness of the abstract microarchitectural model from Definition 17 and the prerequisite that the initial DFA information $m_0 \in \mathbb{M}$ covers $\tilde{Q}_0$, i.e., $\gamma_{\mathbb{M}}(m_0) \supseteq \tilde{Q}_0$, we know that

$$\forall (v^C_i, w_i) \in bbexec(L, \hat{q}^m) : w_i \in \omega(v^C_i) \tag{4.39}$$

where $\hat{q}^m$ is the initial worst-case state that maximizes Equation 4.37. Thus, it follows that

$$\text{WCET}_{\text{real}} = \sum_{v^C_i \in bbexec(L,\hat{q}^m)} w_i \leq \sum_{v^C_i \in bbexec(L,\hat{q}^m)} \omega_{\max}(v^C_i) = \text{WCET}^{\text{opt}}_{\text{est}} \tag{4.40}$$

Of course, since *bbexec* returns a sequence, those v^C_i which occur multiple times in the sequence must also be counted multiple times in the sums above. The *path analysis* is now responsible for finding a $\text{WCET}_{\text{est}} \geq \text{WCET}^{\text{opt}}_{\text{est}}$.

Since the node sequence returned by *bbexec* is a path, we call a sum of weight values for each node a *path weight*. In Equation 4.40, the weight of the worst-case path described by *bbexec* is computed with respect to w_i and $\omega_{\max}$. Unfortunately, this worst-case path is unknown in general. Therefore $\text{WCET}_{\text{est}}^{\text{opt}}$ can again only be approximated. We define $\text{WCET}_{\text{est}}^{\text{feasible}}$ to be the weight of the *feasible* path from the context graph source to its sink which has maximum weight with respect to $\omega_{\max}$. Since the worst-case block sequence must be a feasible path as defined in Section 2.1 we know that $\text{WCET}_{\text{est}}^{\text{feasible}} \geq \text{WCET}_{\text{est}}^{\text{opt}}$.

Unfortunately, it is undecidable whether a path is feasible in general. Therefore user annotations in the form of *flow facts* as introduced in Section 3.3 are needed to specify which paths are feasible. Since the user may choose not to exclude *every* infeasible path but only a subset P^{ex} which must be excluded to achieve a reasonably tight WCET, this procedure yields a $WCET_{\text{est}}^{\text{ann}} \geq WCET_{\text{est}}^{\text{feasible}}$. Those paths which are always in P^{ex} are infinite paths through loops which would otherwise lead to an infinite $WCET_{\text{est}}^{\text{ann}}$ For simple loops, these bounds can also be detected automatically to remove some annotation effort of the user [CM07]. Also, mutual exclusion of special types of context blocks can be automatically assessed in some cases [CBR13], but not in general.

The first approach to path analysis was *tree-based analysis*, which operated on a high-level syntax tree of the program [PK89; PS91; CP01; AGP03]. It allows for an efficient evaluation but is limited since for a precise microarchitectural analysis, the mapping of high-level statements and low-level instructions needs to be known, which is highly non-trivial. A high-level CFG and its corresponding low-level CFGs may be substantially different from each other. For hybrid WCET analyses, the tree-based method can be attractive since no CFG reconstruction is needed then [BB06].

Other approaches include the integration of microarchitectural analysis with the path analysis [EGL11] and explicit path enumeration [CMR+05] which both have proven to not scale well. The latter can be combined with abstract interpretation on a path domain to answer more complex path problems [KFM13].

However, for the classical path problem, the *Implicit Path Enumeration Technique* (IPET) [LM97] is still the de-facto standard. It creates an *Integer Linear Program* (ILP) that models the feasible paths and finds one with maximal weight. Though ILP solving is NP-complete in general, the IPET ILP for reducible programs is totally unimodular [LM97], and totally unimodular ILPs can be solved in polynomial time [KV12].

Since Equation 4.40 already indicates that the $\text{WCET}_{\text{est}}^{\text{ann}}$ can be formed as a summation over block frequencies times the block weight, the IPET ILP for a context graph $G_\tau^C = (V_\tau^C, E_\tau^C, v_{0,\tau})$ uses the objective function

$$\max \sum_{v \in V_\tau^C} \omega_{\max}(v) \cdot x_v \tag{4.41}$$

where x_v is the WCEC of the block v, i.e., the number of times v is visited on the worst-case execution path. The maximization is subject to the following flow conservation constraints which force the generated solution to be a *path* according to the definition in Section 2.1

$$\forall v \in V^C_\tau : \sum_{v_{\text{pred}} \in \delta^-(v)} x_{(v_{\text{pred}},v)} = x_v = \sum_{v_{\text{succ}} \in \delta^+(v)} x_{(v,v_{\text{succ}})} \tag{4.42}$$

where $x_{(v,w)}$ for an edge $(v,w) \in E^C_\tau$ is the WCEC of edge (v,w).

For the IPET, the context graph is extended by a *virtual source* v^+ with edges (v^+,v) for all $v \in \delta^A_\perp(\nu_\perp(f^\perp\tau))$ and a *virtual sink* v^- with edges (v,v^-) for all $v \in \delta^A_\top(\nu_\top(f^\perp\tau))$. These edges are also counted in Equation 4.42, but $v^+,v^- \notin V^C_\tau$. Instead, we have the following initializing and terminating conditions for v^+ and v^-

$$1 = \sum_{v_{\text{succ}} \in \delta^+(v^+)} x_{(v^+,v_{\text{succ}})} \tag{4.43}$$

$$\sum_{v_{\text{pred}} \in \delta^-(v^-)} x_{(v_{\text{pred}},v^-)} = 1 \tag{4.44}$$

Equation 4.43 enforces that one of the edges from v^+ towards an entry node of τ must be executed once, which models the program start. Due to Equation 4.42 the execution frequencies x_v and x_e form a *flow* through the graph which according to Equation 4.44 must end by supplying one flow unit, i.e., one execution, to any of the edges towards the supersink v^-. Still, we will add more constraints in the following, to exclude some infeasible paths from this flow and to bound the amount of flow per node, i.e., the length of the modeled path.

For every call context c with subgraph $G^c_\tau = (V^c_\tau, E^c_\tau)$, we define the set of incoming call edges as $E^c_{\text{callers}} = \{(v,w) \in E^C_\tau \mid v \notin E^c_\tau \wedge w \in E^c_\tau\}$. For each $(v,w) \in E^c_{\text{callers}}$ we define the associated return edge set $E^c_{(v,w)-\text{returns}} = \{(x,y) \in E^C_\tau \mid x \in E^c_\tau \wedge \textit{context}(v) = \textit{context}(y)\}$, i.e., the return edges which lead back into the context of v. These are used to enforce that a path which enters a call context must also use a return edge towards the caller as specified in the constraint

$$\forall \text{call contexts } c : \forall e_{\text{call}} \in E^c_{\text{callers}} : x_{e_{\text{call}}} = \sum_{e_{\text{ret}} \in E^c_{e-\text{returns}}} x_{e_{\text{ret}}} \tag{4.45}$$

Each loop bound of the form `min` $B^l_{\texttt{min}}$ `max` $B^l_{\texttt{max}}$ as introduced in Section 3.3 is attached to a natural loop $l \in \circlearrowleft_\tau$ according to Definition 12. The loop by definition has a unique entry/head node v^l_{head} and a set of back-edges E^l_{back}, such that we can bound the loop iterations in the IPET by limiting the back-edge usage based on the frequency of the entry-edge usages as given in the following constraints

$$\sum_{e_{\text{back}} \in E^l_{\text{back}}} x_{e_{\text{back}}} \leq B^l_{\text{max}} \cdot \left(\sum_{e_{\text{entry}} \in \{(v,v^l_{\text{head}}) \in E^C_\tau\} \setminus E^l_{\text{back}}} x_{e_{\text{entry}}} \right) \tag{4.46}$$

$$\sum_{e_{\text{back}} \in E^l_{\text{back}}} x_{e_{\text{back}}} \geq B^l_{\text{min}} \cdot \left(\sum_{e_{\text{entry}} \in \{(v,v^l_{\text{head}}) \in E^C_\tau\} \setminus E^l_{\text{back}}} x_{e_{\text{entry}}} \right) \tag{4.47}$$

Equation 4.46 is needed to make the WCET finite, whereas Equation 4.47 is optional and will improve the precision of the BCET of the application.

Each flow restriction $c_1^l b_1^l + \ldots + c_n^l b_n^l \leq c_1^r b_1^r + \ldots + c_m^r b_m^r$ is itself already an inequality which relates block execution frequencies b_i to each other, and it therefore can be directly integrated into the ILP by replacing each b_i with its respective block variable as

$$c_1^l x_{b_1^l} + \ldots + c_n^l x_{b_n^l} \leq c_1^r x_{b_1^r} + \ldots + c_m^r x_{b_m^r} \tag{4.48}$$

The flow restrictions can be used flexibly to model path infeasibility, to express dependencies between non-adjacent blocks and to precisely bound complex loop behavior [KKP+11].

The value of the objective function from Equation 4.41 under the constraints from Equation 4.42 to Equation 4.48 is a valid $\mathrm{WCET}_{\mathrm{est}}^{\mathrm{ann}} \geq \mathrm{WCET}_{\mathrm{real}}$. When the objective in Equation 4.41 is *minimized* and $\omega_{\min}$ is used instead of $\omega_{\max}$, a $BCET_{\mathrm{est}}^{\mathrm{ann}} \leq BCET_{\mathrm{real}}$ is produced, analogously.

Whenever we refer to a WCET (BCET) of a task in the following, this always denotes the $\mathrm{WCET}_{\mathrm{est}}^{\mathrm{ann}}$ ($\mathrm{BCET}_{\mathrm{est}}^{\mathrm{ann}}$) unless stated otherwise.

4.5 Evaluation

We evaluated our analysis framework as described in the last sections for a subset of the benchmark suites

- the MRTC benchmarks [Mäl05] (various embedded benchmarks, collected specifically for WCET analysis)
- the UTDSP benchmark suite [LCS92] (DSP kernels and applications)
- the DSPStone Benchmarks [ZVS+94] (DSP-oriented benchmarks)
- the MediaBench collection [LPM97] (multimedia and communications)
- the MiBench suite [GRE+01] (commercially representative embedded benchmarks)
- the NetBench suite [MMH01] (network processor benchmarks)
- the PolyBench/C suite [Pou12] (static control calculations)
- the StreamIt benchmarks [Str14] (streaming applications)

Not all of the benchmarks from the individual suites could be integrated, since they must be prepared for WCET analysis through determination and annotation of flow facts, replacement of dynamic memory allocation (if any) with a static allocation and possibly by fixing compiler incompatibilities, which can be quite time-consuming as already mentioned in the case studies from Section 2.2.7. A detailed list of all benchmarks and their respective properties can be found in Appendix A. All benchmarks were compiled with optimization level O0, which triggers a compilation without further machine code optimization.

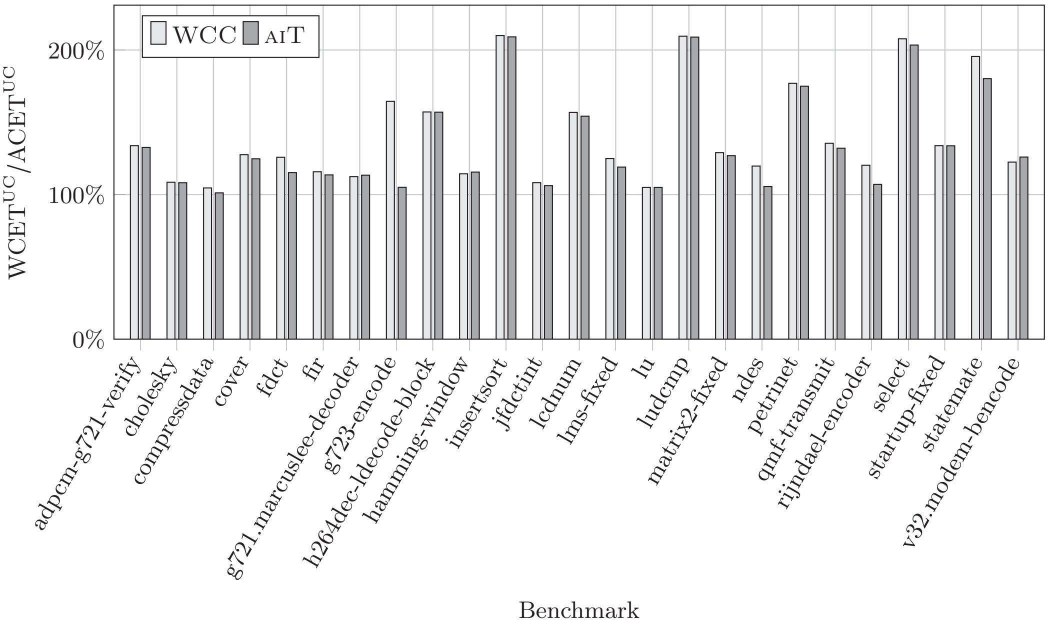


Figure 4.11: WCET performance of the WCC-internal single-core analysis framework for a system without cache usage (superscript UC).

$\frac{\mathrm{WCET}^{\mathrm{UC}}_{\mathrm{WCC}}}{\mathrm{WCET}^{\mathrm{UC}}_{\mathrm{aiT}}}$	$\frac{\mathrm{AnalysisDuration}^{\mathrm{UC}}_{\mathrm{WCC}}}{\mathrm{AnalysisDuration}^{\mathrm{UC}}_{\mathrm{aiT}}}$	$\mathrm{AnalysisDuration}^{\mathrm{UC}}_{\mathrm{WCC}}$	$\frac{\mathrm{WCET}^{\mathrm{UC}}_{\mathrm{WCC}}}{\mathrm{ACET}^{\mathrm{UC}}}$	$\frac{\mathrm{WCET}^{\mathrm{UC}}_{\mathrm{aiT}}}{\mathrm{ACET}^{\mathrm{UC}}}$
104.46%	37.22%	1.59s	163.13%	156.17%

Table 4.1: Average results for the analysis of uncached execution of a single task.

We cannot determine the $\mathrm{WCET}_{\mathrm{real}}$ to compare our WCET results against it. Instead, we measure the ACET and with ACET ≤ $\mathrm{WCET}_{\mathrm{real}}$ the *true overestimation* O_{real} of our analysis can be bounded as follows:

$$O_{\mathrm{real}} = \frac{\mathrm{WCET}^{\mathrm{ann}}_{\mathrm{est}}}{\mathrm{WCET}_{\mathrm{real}}} \leq \frac{\mathrm{WCET}^{\mathrm{ann}}_{\mathrm{est}}}{\mathrm{ACET}} = O_{\mathrm{max}} \tag{4.49}$$

Therefore, we always show O_{max}, i.e., the *maximum overestimation* when discussing WCET results. This is a dimensionless number by definition, which we may present as a percentage, but also as an absolute value where 1 represents 100%, obviously.

The maximum overestimation of our WCET analysis framework for the case of a system with deactivated caches is shown in Figure 4.11 for a representative subset of benchmarks. To disambiguate the results from the WCET and ACET for a system with activated caches we use the superscript "UC". It is visible that most of the time, the WCC-internal analyzer is almost as good as the commercial analyzer aiT, on average[4] it is only 4.46% worse than aiT as shown in the first column of Table 4.1. Both analyzers exploit the fact that the single-core ARM7TDMI platform is free of timing anomalies (cf. Section 2.2.8) to speed up the analysis by following the local worst-case only during the microarchitectural analysis. The compilation and analysis with the WCC-internal analyzer takes 1.59*s* on average which is only 37.22% of the time that is needed for an analysis with aiT. Since the communication of the WCC and aiT is done via files (cf. Figure 3.1) whereas the WCC-internal analyzer uses only in-memory structures, this comparison is biased against aiT. Still, it shows that the analysis speed is competitive without actually claiming superiority to aiT here. The overestimation compared to the measured execution time is 63.13% for the WCC and 56.17% for aiT as shown in the last two columns of Table 4.1. This overestimation mainly stems from

- Imprecise results of the value analysis. Especially when the target of an indirect memory access cannot be determined precisely, the analysis must assume an access to the memory module with the highest access duration (Flash with 5 cycles compared to 1 cycle for the scratchpads). If this happens in a loop, it can drastically impair the analysis precision.
- Imprecise flow facts for loops with variable iteration counts. For those loops we only specified minimum and maximum bounds. More precise results could potentially be obtained with the usage of custom flow restrictions.

The slight advantage of aiT can be explained with the first point since aiT's value analyses are more precise when pointer computation or array accesses are involved.

As a second scenario, we also analyzed the benchmarks for a system with activated instruction cache, i.e., the benchmark code is cached but not the data objects. Since the benchmarks vary in size, the instruction cache size was adjusted to match 50% of the benchmark's code size rounded to the closest power of two. In this scenario, we compare the analysis precision of the WCC with activated cache analysis ($\mathrm{WCET}^{\mathrm{C}}_{\mathrm{WCC}}$) with the with the results of an analysis that assumes all cache accesses to be hits ($\mathrm{WCET}^{\mathrm{C,AH}}_{\mathrm{WCC}}$) or misses ($\mathrm{WCET}^{\mathrm{C,AM}}_{\mathrm{WCC}}$). In Figure 4.12 we can see that the cache analysis outperforms the pessimistic analysis $\mathrm{WCET}^{\mathrm{C,AM}}_{\mathrm{WCC}}$ by far in most cases. The optimistic analysis $\mathrm{WCET}^{\mathrm{C,AH}}_{\mathrm{WCC}}$ does *not* provide safe WCET estimates as expected.

The overestimation however is still considerably higher than in the uncached case. Since we are analyzing an instruction cache here and the ARM7TDMI has a very predictable fetch behavior, the analysis of fetch operations is not the problem. What *is* a problem for the cache analysis is the fact that

- the platform is a Von-Neumann architecture, i.e., data and instruction accesses may target the same memory modules,

[4] Here and in the following, we exclusively use the *geometrical mean* for relative numbers (like relative WCETs), since arithmetical means of relative numbers can lead to invalid and even contradictory conclusions depending on the selected comparison base [FW86].

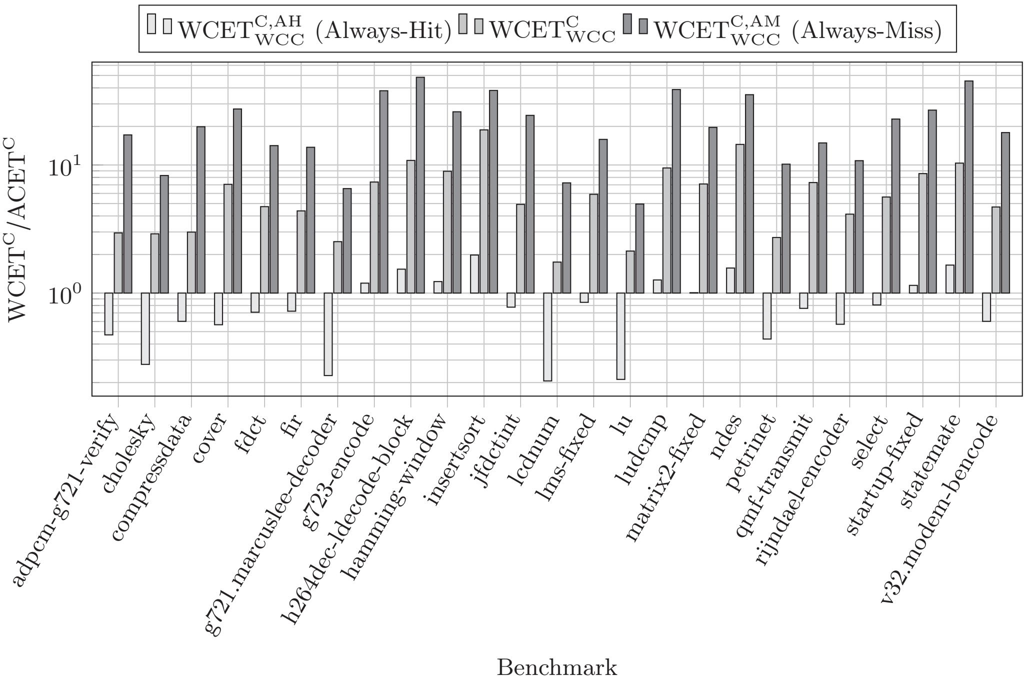


Figure 4.12: Single-core WCET results for a system with activated cache (superscript C).

- the ARM compiler actually uses this and embeds data items into the instruction stream for a better performance,
- the value analysis in its current form often cannot determine the targets of indirect loads and stores especially for pointer arithmetic and array accesses.

Therefore, the cache analysis has to update the instruction cache state also in the case of loads or stores to unknown (data) targets. Depending on the number of these types of accesses, this degenerates the cache state precision. Since for every miss, a cache line refill is triggered (cf. Figure 4.10), the effective penalty for a miss is $1\,\mathrm{cycle} + (32\,\mathrm{B}/4\,\mathrm{B}) \cdot 3\,\mathrm{cycles} = 25\,\mathrm{cycles}$ (see the parameter list in Table 3.1), thus a decrease in cache state precision directly translates to a decrease in WCET precision.

The average results for the cached case are shown in Table 4.2. aiT for the ARM7TDMI also features a cache analysis, which can be configured to some extent. However, it does *not* model the precisely same cache settings as our analyzer. Only the line size, cache size and associativity are adjustable in aiT, but none of the other parameters from Table 3.1. Still, the WCET results are roughly comparable, therefore we also included the average result for aiT in Table 4.2. Since aiT employs a superior value analysis it is clearly better here, leading to a maximum overestimation which is 2.26 times as good as the one of the WCC and 8.16 times better than the pessimistic analysis, respectively. However, Table 4.2 also indicates

$\frac{\text{WCET}^{\text{C,AH}}_{\text{WCC}}}{\text{ACET}^{\text{C}}}$	$\frac{\text{WCET}^{\text{C}}_{\text{WCC}}}{\text{ACET}^{\text{C}}}$	$\frac{\text{WCET}^{\text{C,AM}}_{\text{WCC}}}{\text{ACET}^{\text{C}}}$	$\frac{\text{WCET}^{\text{C}}_{\text{aiT}}}{\text{ACET}^{\text{C}}}$	$\frac{\text{AnalysisDuration}^{\text{C}}_{\text{WCC}}}{\text{AnalysisDuration}^{\text{UC}}_{\text{WCC}}}$
88.02%	645.17%	2,318.00%	284.58%	117.52%

Table 4.2: Average results for the analysis of cached execution of a single task.

that $\text{WCET}^{\text{C}}_{\text{WCC}}$ is 3.56 times better than $\text{WCET}^{\text{C,AM}}_{\text{WCC}}$ and that the analysis takes an acceptable 17.52% more time than for the uncached architecture.

By manual annotation of register value ranges and flow restrictions the precision of the WCETs can be drastically improved in both scenarios. Since this is a quite time-consuming job, which requires a detailed review of each benchmark, this was not pursued here. The precision of the single-core analysis is sufficient to demonstrate the multi-core analysis techniques that will be discussed in the next section, even without such further annotations.

Chapter 5

Multi-Core WCET Analysis

Contents

5.1 Introduction

In this chapter, we will first investigate which challenges are posed to WCET analysis due to the advent of multi-core architectures. We will then review existing approaches to these challenges and finally present two different multi-core WCET analysis algorithms which build upon the single-core analysis as presented in the previous chapter.

5.2 Multi-Core Challenges

The memory hierarchy has a significant impact on the average-case performance of a system. Measurements show that modern CPUs spend 90% of their time waiting for memory in real-life benchmarks [Jac09, Chapter 2]. Since also the speed difference between the memory levels can amount to orders of magnitude [PH11, Chapter 2.1], each access may have a considerable impact on the resulting ACET. As already seen in the evaluation in Section 4.5, the impact on the WCET is even higher since it needs to account for every reachable hardware state. If infeasible accesses cannot be excluded, they will thus degrade the WCET precision.

In a multi-core system, each core's pipeline can be analyzed in isolation but the memory hierarchy state can not since typically the cores share some part of the memory hierarchy, e.g., in the form of shared caches, shared buses and shared memory. Thus, a task τ_1 running on core c_i may be able to impair the timing of task τ_2 running on core $c_j \neq c_i$. This complicates the WCET analysis, since it has to account for all possible interference behaviors and still has to be as precise as possible. A good overview on how the predictability of a system suffers from shared resources is given in [ABD+13]. The authors distinguish between

- bandwidth resources, mainly *shared buses* and
- storage resources, mainly *shared caches.*

We will review the problems that arise due to these two resource types in the following. Shared memory in the form of S-RAM is usually not a problem, since, it has an approximately constant access time. For D-RAM memory the challenges are the same as in the single-core case, namely that the periodic refresh can hardly be modeled in the system state [BM11]. Therefore we limit ourselves to S-RAM in this work.

5.2.1 Shared Caches

The "memory wall" problem [Mar11, Chapter 3.4] that the speed of the CPU is increasing faster than the speed of RAM is even more pressing in the multi-core case than in the single-core one, since the requests from more than one CPU must be served by the slow memory. Cache hierarchies therefore are a critical factor for average-case multi-core performance [BJM11].

As we have seen in Section 4.3.2 the static analysis of cache behavior must approximate all possible access sequences with which the current context block position may be reached. In the case of a shared cache these sequences are no longer limited to accesses from the current task but they can also contain accesses from other tasks. The classical single-CFG-based data-flow analysis cannot infer anything about their location in the sequence, thus we only have the option to either loose precision and provide rough over-approximations, or to develop new analysis approaches. We will discuss both of these approaches in the following.

As an additional complication, shared caches are sometimes also kept *coherent* [SHW11] by a hardware protocol. In this case, changes to one private cache will propagate to other private caches. Effectively, this turns every coherent cache into a shared cache, which can be handled by similar methods as shared caches. Still, the mechanism used to implement the coherency is more complex in both its microarchitectural implementation and its timing behavior. Since the analysis of shared caches alone is difficult already, this additional complication makes it very close to impossible to provide precise WCET estimates for cache-coherent systems. We will therefore assume absence of cache coherency mechanisms in the following.

5.2.2 Shared Interconnection Structures

Before any shared resource, like a shared cache, can be accessed by multiple cores, there must be a shared interconnection structure which allows this type of access. An example for this is the shared bus in Figure 3.4. The target module, be it a shared cache or a shared memory, can only be accessed by one core at a time, therefore the interconnection structure must provide an *arbitration* to resolve potential conflicts. Compared to the shared cache analysis, this is a more elementary problem, since the timing analysis of the employed arbitration method affects every access, also for systems without shared caches. We will therefore focus on the analysis of the arbitration timing behavior in the following discussions.

Interconnection structures are generally subdivided into *interconnection networks* and *buses*. In accordance with established terminology, devices which can initiate communication requests are called *masters* (e.g., core pipelines, caches) whereas those who can not are called *slaves* (e.g., memory).

Interconnection Networks

*Interconnection Network*s (ICNs), also called *switched-media on-chip networks* [PH11, Appendix F], are composed of a number of point-to-point links and switches which route packets through the network. The most prominent and widely-used example for such an on-chip network is PCI-Express, which is installed in virtually every modern PC.

In general, the timing behavior of a transmission through an ICN depends on the collisions that may occur in the network due to other transmissions which want to access the *same* or a *different* slave node. For cases where n masters each issue a communication request to one of n *different* slaves and no slave is the target of more than one request, there exist ICN topologies which are *strictly collision-free* [SJ96], i.e., all of these requests can be satisfied in constant time. Unfortunately, in WCET analysis we often cannot exclude the case that multiple masters (cores) may access the *same* slave (memory module). In such a case, only one access at a time can be granted which leads to an inevitable amount of blocking.

These conflicts can be elided for read accesses which want to read the same memory location as for example in the logarithmic HYPERCORE network [BHQ+07; Bay08; Plu10], but again we may be unable to prove that definitely the same address is accessed.

From a timing perspective, ICNs potentially lead to higher average-case performance, but their complex state space complicates WCET analysis. The methods that we are going to use for the analysis of buses, i.e., abstract state machines and abstract arbitration functions, are also applicable to ICNs but the modeling effort increases whereas the WCET precision will drop. In the important case of time-triggered ICNs which offer deterministic transmission times for real-time systems, the methods used for buses are even immediately applicable without changes or precision loss. Therefore, we focus on shared buses in the following.

Buses

As shown in Figure 3.4, a *bus* is a shared medium which connects multiple masters and slaves, but only a single physical signal can be transmitted over the bus at each point in time. Therefore, if multiple masters want to use the same bus, they must share it by either

- *Frequency-Division Multiplexing* (FDM)
- *Code-Division Multiplexing* (CDM)
- *Time-Division Multiplexing* (TDM)

Frequency-Division Multiplexing (FDM) lets multiple masters transmit their data in parallel, but physically modulated to different frequency bands. This limits the bandwidth and thus the transmission speed that is available to each master. The author is not aware of any implementation of a bus using FDM. This technique is successfully used to multiplex signals over telephone lines carrying analog telephony, ISDN and DSL signals in different frequency bands, but apparently the miniaturization to chip-level buses poses electro-technical problems.

Code-Division Multiplexing (CDM) is similar to FDM in that multiple masters transmit their signal at the same time. The difference is, that the signals are not modulated purely in the frequency domain. They undergo a convolution with a predefined *code signal.* These code signals must be defined in such a way that the original signals can be extracted even after the signals from multiple masters have been mixed with each other. Though some approaches to CDM on the chip-level exist [BKJ+01], they have the same conceptual disadvantage as FDM, namely that each CDM channel has a limited bandwidth and transmission speed. Overall, this is still a niche topic, which does not seem relevant at the moment.

In *Time-Division Multiplexing* (TDM) only a single core may use the bus at any time, i.e., if multiple masters want to use the bus the order in which their accesses are granted must be determined by an *arbiter*. This method is widely employed in digital communication technology as, e.g., in GSM and ISDN and in most computer buses like USB, PCI and ISA.

TDM is usually implemented with non-preemptable accesses [SJ96, Section 3.1.3] since preemptions would require buffering or abortion of accesses which complicates the hardware or decreases the performance. Therefore, we also assume non-preemptable accesses in the following. We also assume a *maximum access time* $T^B_{\max}$ that specifies the maximum duration for accesses to a device behind the shared bus B. In practice, this can be computed from the device configuration registers in the example of PCI [Tec14].

For the arbiter, three major categories of arbiters are employed:

Fair arbitration Also called *Round-Robin*, this arbiter rotates the bus access among all masters. It maintains an *active master* $m_a \in \{0, \ldots, n_c - 1\}$. When an access finishes, m_a is cyclically advanced to the next master which requests the bus. Thus, each master can acquire the bus after at most $n_c - 1$ others have performed their accesses.

In the case of our reference architecture from Figure 3.4, each bus connection may be a master, i.e., from the point of view of the shared bus, a core and its L1 caches are the same master, though they may issue requests independently. Access collisions on the local core bus cannot occur though, since the ARM7 blocks upon an cache miss.

Practical applications of this scheme include ARM AMBA, the PCI [Alt00] bus and PROFIBUS [WB05, Sec. 4.4.2]. Under AMBA and PCI, the arbitration protocol is configurable but includes fair arbitration. In the case of PCI, a timeout is imposed to enforce a predefined maximum access time [Buc00, Section 4.3].

(Static) Priority-based arbitration Here, a unique priority $p_i \in \{1, \ldots, n_c\}$ is assigned to each master m_i. If there are multiple requests only the request from the master with the highest priority is granted. Nevertheless, since accesses are non-interruptible, even the highest-priority master may have to wait until an ongoing transaction is completed.

Again, AMBA and PCI [Tec14] can be configured to use priority-driven arbitration. Another well-known application is the CAN bus [WB05, Sec. 4.4.3] though this is typically not used as an on-chip bus.

Time-triggered arbitration The basic notion that time-triggered operation contributes to timing-predictability is evident through the large body of work invested into the *Time-Triggered Architecture* [KB03] and into time-triggered scheduling [Liu00, Chapter 6]. In a time-triggered arbitration scheme, a centralized arbiter must assign the bus based on the current time, i.e., the number of passed bus cycles. Alternatively, a distributed implementation is possible where synchronized, distributed clocks [Lam78] are used to control local bus guards.

Time-Division Multiple Access (TDMA) creates a schedule consisting of n_l slots of size l_s and assigns an owner master $o_i \in \{1,\dots,n_c\}$ to each slot. The current position in the schedule is determined by taking the current clock tick modulo $n_l l_s$. In each slot i, only the owner is granted access to the bus and only in the interval $[il_s,\dots,(i+1)l_s - T^B_{\max}]$. The subtraction of $T^B_{\max}$ is necessary to make sure that accesses complete before the next slot begins.

Priority Division (PD) [SRK11] is a generalization of TDMA. Instead of assigning an owner o_i it assigns unique priorities $p_{ij} \in \{0,1,\dots,n_c\}$ for each slot i and each master j. For each slot i, the bus is granted to the requesting master with the highest positive priority, but again only in the time frame $[il_s,\dots,(i+1)l_s - T^B_{\max}]$. Only those masters with priority 0 are excluded from arbitration, which can be used to emulate TDMA behavior.

Real-life implementations of TDMA include the Time-Triggered Protocol [PK08], Aethereal [GH10; HG11], FlexRay [Mar11, Section 3.5.4] and PROFInet/IRT [WB05, Section 4.4.11]. Technically, Aethereal is not a bus but an ICN which is based on a design-time allocation of time-triggered schedules and communication paths for each master. Therefore, from the analysis perspective it can be seen as a generalization of a TDMA-scheduled bus, with the difference that each core is presented with an individual access schedule.

5.3 Related Work

In the following we review existing approaches to the analysis of shared resources in multi-cores. The related work can be partitioned into works which try to analyze existing hardware and those which propose new hardware structures to accommodate the needs of WCET analysis.

5.3.1 WCET Analysis Approaches for Multi-Cores

As mentioned in Section 2.2.9, the behavior of each component can be analyzed separately in an (γ,α)-compositional timing analysis. Unfortunately, as we have seen in Section 4.3, the microarchitectural analysis in general needs to maintain a product state of all hardware component states to achieve reasonable precision. Therefore, a separate analysis of individual hardware components, i.e., a $(1,0)$-compositional analysis, incurs a loss of precision in general.

Approaches which can exclusively analyze shared caches or buses and which do not integrate into a general WCET analysis framework like shown in Chapter 4 are therefore of limited use. They often assume that some constant penalty can be added or subtracted for each additional miss or hit or bus event, which is only true for hard-to-obtain $(1, 0)$-compositional WCET analysis values.

Separate Analysis of Shared Caches

The first attempts that focused on exclusively analyzing the shared cache behavior are [YZ08; ZY09] where an "ad-hoc" method for analyzing the possible interference is given. The method can be used to bound the cache interference but it does not integrate well with any of the known WCET analysis concepts.

An integration of sharing effects into the single-task cache analysis as described in Section 4.3.2 was given by Li et al. [LSL+09]. There, a summary of the possible cache effects that all other concurrent tasks in the system may have is computed. Based on this summary, the block age map is updated and the classification of accesses is altered to "unknown" if the respective access may suffer from interference with other cores. Effectively, this is still the only option that was found up to now, namely to precompute a worst-case summary of all possibly concurrently ongoing cache actions and apply this worst-case summary to the local cache results. This concept was also taken over into the integrated multi-core analysis frameworks in [KFM+14; CCR+14] and is a baseline for this thesis.

To account for systems where tasks may migrate between cores, Hardy [HP09] generalized the CRPD concept to a *cache-related migration delay* which can also be statically computed. In addition, Lesage, Hardy and Puaut propose a cache-bypassing heuristic in [LHP10] to reduce the amount of shared cache interference.

Separate Analysis of Shared Buses

For shared buses, some approaches compute the maximum number and duration of bus accesses independently of the task WCET analysis and add the extra delay to the task WCETs later [AEL10; DAN+11].

A very similar family of approaches is the usage of the *Real-Time Calculus* (RTC) for deriving the worst-case duration of the shared memory accesses which is later added to the total task WCET or WCRT. A prominent example of this approach is given by Schliecker et al. for plain shared memory [SNN+08] and for resources locked under the global priority ceiling protocol [NSE09]. Pellizzoni et al. detail how to apply the RTC-based approach to COTS hardware [PSC+10] and how to isolate aperiodic events by usage of a dedicated *hardware server device* [PC10].

Schranzhofer, Chen and Thiele [SCT09] employ the RTC-based approaches to asses various access models which restrict *when* inside the task shared resource access can be made. They find that tasks which fit into a *read-execute-write* phase model produce especially low WCETs. This approach was later refined towards more precise WCRT results for TDMA arbitration [SCT10] and towards integration of schedulability results [SPC+10].

All of these approaches require $(1, 0)$-compositional WCET analyses to keep up the validity of the generated WCET as mentioned above. Therefore, they are more suitable for soft real-time applications on *Components Off The Shelf* (COTS), where precision and analysis speed are more important than WCET safeness.

Integrated Analysis of Shared Resources

The first attempt to combine the cache analysis of [LSL+09] with a handling of the bus accesses was made in [CRM10]. A *loop alignment* is used to analyze accesses to the TDMA bus from inside of loops in the tasks. This concept was carried further in [KFM+11] where a static analysis of TDMA bus hardware states was proposed. Finally, [CKR+12] extended this approach by a graph-based pipeline handling taken from [LRM06] and the correctness of the loop alignment was proven in [KFM+14]. These publications are the basis for the content of Section 5.4.

A different approach, which combines an abstract interpretation-based cache analysis with model checking-based bus analysis was presented by Lv [LYG+10]. The reported analysis times are better than for a purely model-checking-based approach [GEL+10], but for more complex pipelines the same problems will be perceived as in the purely model-checking case [Wil04].

A topic that is not taken into account by any of the above approaches is the synchronization structure of the program given by “lock”, “release” and “barrier” operations. Gustavsson employed the model-checker UPPAAL to generate synchronization-aware multi-core WCET values [GEL+10], but the analysis takes multiple hours for a system with a simple pipeline, no bus modeling and tasks which correspond to 10 lines of C code. Therefore, this approach was continued on a high-level code representation in [GGL12]. The latter approach is a purely theoretical one without any evaluation or implementation.

Ozatkas, Rochange and Sainrat [ORS14] approach the analysis of synchronization behavior from a different perspective. They devised a method of estimating the *worst-case stall time* due to synchronization. Potop and Puaut proposed an integrated source-level framework for incorporating synchronization inside of tasks into the WCRT computation [PP13] which has the potential to scale far better than [GEL+10].

Practical Experiences

Nowotsch and Paulitsch [NP13] demonstrate that predictability is also desirable from the industry point of view, not only to ease WCET analysis but also to increase the determinism in a multi-core system and with it the repeatability of experiments on those systems. They also propose a method for subdividing the WCET into a core-local part and an additive component caused by interference of other cores [NPH+14]. As discussed above, this requires $(1, 0)$-compositional WCET values.

Experiences on applying a WCET analysis on a benchmark executed on a predictable MERASA multicore are given by Rochange et al. [RBS+10]. The results are formed manually by addition of single-core WCETs for the individual cores.

5.3.2 WCET-friendly Multi-Core Architecture Design

Many publications try to design and promote hardware which is more predictable and more WCET-friendly than existing multi-cores. All of these approaches of course require the respective custom hardware to be available and most of them have only been implemented on an FPGA up to now.

Mische et al. [MUK+08; MGU+10] propose the CarCore architecture which is built upon the Infineon TriCore. It contains exactly one hard real-time task which is treated as if it was the only task in the system, i.e., all hardware components must immediately service its requests and abort those of lower-priority tasks. In this way, the WCET analysis from the single-core case can be re-used and multiple non-real-time tasks can coexist with the hard real-time task without the ability to interfere with its timing. Multiple hard real-time tasks can be aggregated into one meta task which can itself host multiple sub-tasks by time-sharing [MUK+08].

In a follow-up work, Mische et al. [MMU11] discuss a many-core architecture with predictable timing, which is achieved by highly predictable in-order cores and a TDMA-arbitrated mesh network, such that network communication runtime and task WCET can be analyzed independently and added up afterwards.

Paolieri et al. [PQC+09b] propose the integration of a *worst-case computation mode* into the bus arbiter and later also into the memory controller [PQC+09a], such that the WCET of individual basic blocks can be measured. Though the idea seems appealing, it also comes at the price of a drastical WCET overestimation. A later publication of the same author tries to leverage this problem by not enforcing the worst-case in hardware but by adding a watchdog module that can assert that blocks of instructions are finished within a user-definable time frame [PM11]. This does not yield safe WCET estimates but eases the verification of the timing at least for the test benchmarks.

Wilhelm at el. have given advices on the design of predictable multi-core systems in terms of both hardware and software [WFC+09; CFG+10] which have become known as the *PROMPT principles* (PRedictability Of MultiProcessor Timing). They also give advice on how to configure a commodity system as predictable as possible [KSP+12].

The PRET architecture [LRL10; LRB+12] offers fixed durations of every single instruction and is thus highly predictable. Still, the attempt is made to also deliver an acceptable average-case performance by implementing a thread-interleaved pipeline which executes multiple threads in fine-grained alternation.

Pitter and Schoeberl [PS10] introduce the *Java-Optimized Processor (JOP)* and analyze the WCET of programs running on it for different bus arbitration strategies. They also examine the performance loss that is incurred by tailoring the architecture towards predictability instead of towards average-case performance.

Bui et al. present extensions of an instruction set that might be useful for WCET analysis, including an instruction to set a deadline for certain code blocks [BLL+11].

XMOS is the first industrial chip producer that has produced a timing-predictable multi-core – the XCORE – whose timing behavior can easily be analyzed, and which therefore is even delivered with a custom-built WCET analyzer tool [XMO13].

A new, more predictable cache coherence mechanism, the *On-Demand Coherent Cache* is proposed by Pyka, Rohde and Uhrig [PRU13]. The basic idea is that cache coherence is deactivated most of the time and only switched on selectively for very small program regions. In this case, only these regions are affected by the overestimation related to cache coherence timing estimation.

To make multicores more predictable for a single high-priority thread, Gudidevuni and Zhang [GZ10] propose a priority-driven cache replacement policy, but this is more measurement-oriented and limited to a single high-priority thread. From the WCET analysis perspective, it does not yield much benefit.

Finally, Münch, Paulitsch and Herkersdorf show that temporal separation, which is advocated for in many of the WCET papers referenced above, can be achieved also on commodity interconnects like PCI-Express [MPH14].

5.4 Partitioned Multi-Core WCET Analysis

In this section, we investigate the achievable precision for a multi-core WCET analysis which analyzes each core and each task in isolation. For each task, the analysis structure shown in Figure 4.1 stays valid but the individual stages are modified such that interferences by tasks on other cores are taken into account. This procedure is motivated by the fact that

- the complexity of the resulting analysis is lower than in the case of an exploration of the whole system's state (state space cross product) and
- it allows a certification of a single task without knowledge of the complete system. This is often a necessity in modern development teams where different suppliers may independently produce software components.

The resulting structure is shown in Figure 5.1, where each $\tau_{i,j}$ denotes task number j allocated to core i. T_i is again the set of tasks allocated to core i. For the analysis of shared caches we will require some (static) information about the other tasks in the system. Therefore, this introduces a (loose) coupling between the individual core analyses. Since we only require the range of *memory addresses* that tasks on other cores may access, this can also be resolved by a "contract" between the tasks concerning the memory space usage.

We do not analyze the content of memory modules during the value analysis. Therefore sharing is not a problem here. But even if we did, setting the value of shared variables to $\top$ suffices to account for parallel modifications.

For the path analysis, *explicit synchronization* is a new control-flow element. One possibility to account for it are manually determined bounds on the iteration count of spin-locks, i.e., a fall-back to loop bounds. A different option is to incorporate synchronization or communication edges into a parallel context graph, as done

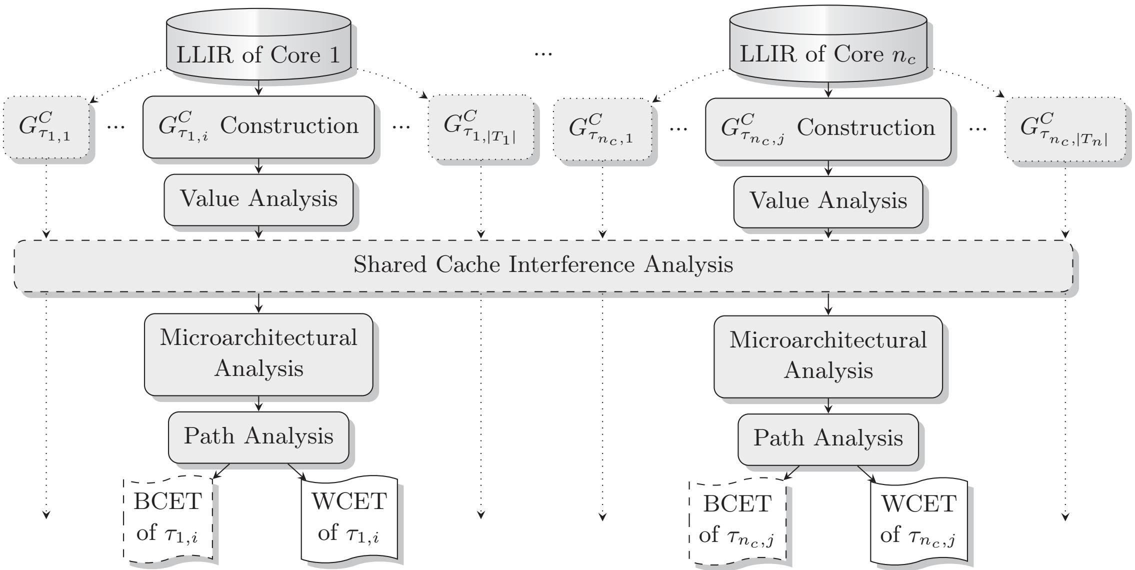


Figure 5.1: Structure of the core- and task-partitioned WCET analysis with optional cache interference analysis.

in [PP13]. Unfortunately, this technique is limited to synchronization statements outside of loops up to now since synchronization in a loop breaks the network-flow-based semantics of the IPET (cf. Section 4.4). We therefore do not handle explicit synchronization but require the tasks to either

- bound it through traditional loop bounds or
- use *implicit synchronization*. In a system where tasks are time-triggered and WCETs are known, mutual exclusion can often be verified through the timing itself, thereby removing the need for runtime-intensive explicit synchronization.

In the next subsections, we therefore focus on the handling of microarchitectural interference in the shared cache and bus.

5.4.1 Shared Cache Handling

We adopt the technique from [LSL+09] by first deriving the cache sets $sets(\tau) \subseteq S$ that may be accessed by a task τ. For each set $s \in sets(\tau)$, the tags that may be accessed by τ in this set are given by $tags(\tau, s)$. Once the value analysis has been completed, these functions can easily be derived by inspecting the task code, gathering all addresses that may be accessed by the instructions of the task and computing their sets and tags.

In addition, we need some notion of which tasks may run in parallel to a task τ from core c, called $par(\tau, c)$. Conservatively, we first assume $par(\tau, c) = \bigcup_{x \neq c} T_x$, i.e., τ may run in parallel to all tasks from other cores. The computation of $sets(\tau)$,

$tags(\tau)$ and $par(\tau, c)$ constitutes the "Shared Cache Interference Analysis" stage from Figure 5.1.

The "Microarchitectural Analysis" stage then uses the interference data to update the single-core cache results. It must alter Equation 4.32 for classifying an access to set s and tag t from the current task τ and core c to

$$cls(q_s^C, t) = \begin{cases} \{\texttt{HIT}\} & \text{if } a_{max}^t < \hat{a} - |\bigcup_{\tau_p \in par(\tau,c)} tags(\tau_p, s)| \\ \{\texttt{MISS}\} & \text{if } a_{min}^t = \infty \wedge |\bigcup_{\tau_p \in par(\tau,c)} tags(\tau_p, s)| = 0 \\ \{\texttt{HIT}, \texttt{MISS}\} & \text{else} \end{cases} \tag{5.1}$$

where $\hat{a}$ is again the associativity of the cache and $q_s^C(t) = [a_{min}^t, a_{max}^t]$. The rationale behind this is, that cache hits can still be guaranteed if there are not enough potential accesses by other cores ($\bigcup_{\tau_p \in par(\tau,c)} tags(\tau_p, s)$) to evict the element from the cache. Misses are only kept guaranteed if there is no other core which might have loaded the element in parallel. Otherwise, the classification is degraded to "unknown" ($\{\texttt{HIT}, \texttt{MISS}\}$) and the microarchitectural analysis has to explore both possibilities.

Obviously, this is a rather coarse-grained estimation which will classify *all* shared cache accesses as "unknown" once a certain number of parallel tasks is reached. In Section 5.5, we will present a computationally intensive technique to generate more precise results.

Algorithm 5 Refinement of the shared cache results for timing-anomaly-free architectures.

```
Set $\forall \tau, c : par^0(\tau, c) \leftarrow \bigcup_{x \neq c} T_x$ and $i \leftarrow 0$
Compute hyperperiod $p^H \leftarrow lcm(p_1, p_2, \ldots, p_{|\bigcup_{c \in \{1,\ldots,n_c\}} T_c|})$
repeat
    $i \leftarrow i + 1$
    for $\tau \in \bigcup_c T_c$ do
        Compute WCET$^i(\tau)$ as shown in Figure 5.1
    for $c \in \{1, \ldots, n_c\}$ do
        Job set $J_c \leftarrow \{(t_i, \tau_i) \mid \tau_i \in T_c \wedge \exists k \in \mathbb{N} : t_i = kp_j \wedge t_i < p^H\}$
        Job sequence $S_c \leftarrow sort(J_c)$ in ascending order of "spawn times" $t_i$
        for $\tau \in T_c$ do
            $\delta^i(\tau) = \varnothing$
        for $(t_j, \tau_j) \in S_c$ do
            $\delta^i(\tau_j) \leftarrow \delta^i(\tau_j) \cup [t_i, \max(t_{j-1} + \mathrm{WCET}^i(\tau_{j-1}), t_j) + \mathrm{WCET}^i(\tau_j)]$
    Update $\forall \tau, c : par^i(\tau, c) \leftarrow \{\tau_o | \tau_o \notin T_c \wedge \delta^i(\tau_o) \cap \delta^i(\tau) \neq \varnothing\}$
until $\forall \tau, c : par^i(\tau, c) = par^{i-1}(\tau, c)$
return WCET$^i$
```

However, if the system under analysis is free of timing-anomalies and we know the period p_i for each task $\tau_j \in \bigcup_c T_c$, then we can refine the results through the steps shown in Algorithm 5. In line 1 we initialize *par* with our conservative assumption mentioned above. Line 2 computes the *hyperperiod* of the task set as the least common multiple of the periods. The main loop then recomputes the task WCETs (line 6), determines from the periods and the WCETs the *lifetime windows* of the tasks (line 13). Since a task τ_j may be blocked by a preceding task executing on the same core, the job sequence S_c is computed in line 9 for each core c, first. The maximization operator in line 13 accounts for possible blocking. Those tasks whose lifetime windows do not overlap cannot be executed in parallel and are thus excluded from par^i in line 14. These steps are then repeated until *par* – and thus also the WCET – reaches a fixed point. The resulting WCETs are then returned in line 15.

Theorem 1. *For timing-anomaly-free systems, Algorithm 5 terminates and provides a valid overapproximation of $par(\tau, c)$ for all tasks τ and cores c.*

Proof. To show the termination, it is sufficient to prove that par^i is monotonically decreasing in i, i.e.,

$$\forall i : \forall \tau, c : par^i(\tau, c) \subseteq par^{i-1}(\tau, c) \tag{5.2}$$

We prove this by induction over i. In the first iteration with $i = 1$ we have $par^1(\tau, c) \subseteq par^0(\tau, c) = \bigcup_{x \neq c} T_x$ which is trivially true according to the definition of $par^1(\tau, c)$ in line 14 of Algorithm 5.

For the induction step, we assume the existence of a task τ_e on a core c with $par^{i+1}(\tau_e, c) \supset par^i(\tau_e, c)$ and show that this leads to a contradiction. If we assume that it existed, there must be at least one task $\tau_e^o \in par^{i+1}(\tau_e, c) \setminus par^i(\tau_e, c)$. According to the definition of $par^{i+1}(\tau_e, c)$ this means that $\delta^{i+1}(\tau_e) \cap \delta^{i+1}(\tau_e^o) \neq \varnothing$ whereas $\delta^i(\tau_e) \cap \delta^i(\tau_e^o) = \varnothing$. The lower bounds of the windows that constitute any $\delta(\tau)$ are the "spawn times" t_i which never change (cf. line 13). Therefore, $\delta^{i+1}(\tau_e) \cap \delta^{i+1}(\tau_e^o) \neq \varnothing$ is only possible if the upper bound of any such window has grown, i.e., if $\text{WCET}^{i+1}(\tau) > \text{WCET}^i(\tau)$ for $\tau = \tau_e$ or $\tau = \tau_e^o$ or τ being a predecessor of τ_e or τ_e^o in the respective job sequence S_c. To complete the contradiction we show that such a WCET growth is not possible for *any* task τ, i.e.,

$$par^i(\tau, c) \subseteq par^{i-1}(\tau, c) \implies \text{WCET}^{i+1}(\tau) \leq \text{WCET}^i(\tau) \tag{5.3}$$

Any WCET^{i+1} is computed based upon par^i in Algorithm 5. A smaller *par* induces *less or equal* "unknown" classifications in Equation 5.1. This means that for any shared cache access r with $cls(q^C, r) = \{\texttt{HIT}, \texttt{MISS}\}$, either the classification stays unchanged or

- r is now classified as $\{\texttt{HIT}\}$. In a timing-anomaly-free system, the local worst-case is always also the global worst-case (cf. Figure 2.5), i.e., for the context block v_r that issued r we have $\omega_{max}^{i+1}(v_r) < \omega_{max}^i(v_r)$. Thus, $\text{WCET}^{i+1}(\tau) \leq \text{WCET}^i(\tau)$ depending on whether v_r is part of the WCEP.

Shared component	Bounded Access Delay	State-Permeable
Shared cache	Yes	Yes
Shared bus (PRIO)	No	Yes
Shared bus (FAIR)	Yes	Yes
Shared bus (TDMA)	Yes	No
Shared bus (PD)	Configuration-dependent	

Table 5.1: Properties of shared resources in multi-cores.

- r is now classified as {`MISS`}, i.e., the local worst-case has stayed the same which leads to $\omega^{i+1}_{max}(v_r) = \omega^{i}_{max}(v_r)$ and $\text{WCET}^{i+1}(\tau) = \text{WCET}^{i}(\tau)$, respectively.

With the monotony of par^i the correctness can easily be shown by induction over i and the prerequisite that the WCET computation for a given par^{i-1} is correct. □

Algorithm 5 and Theorem 1 were first given in [LSL+09]. In this thesis, they are generalized to periodic tasks.

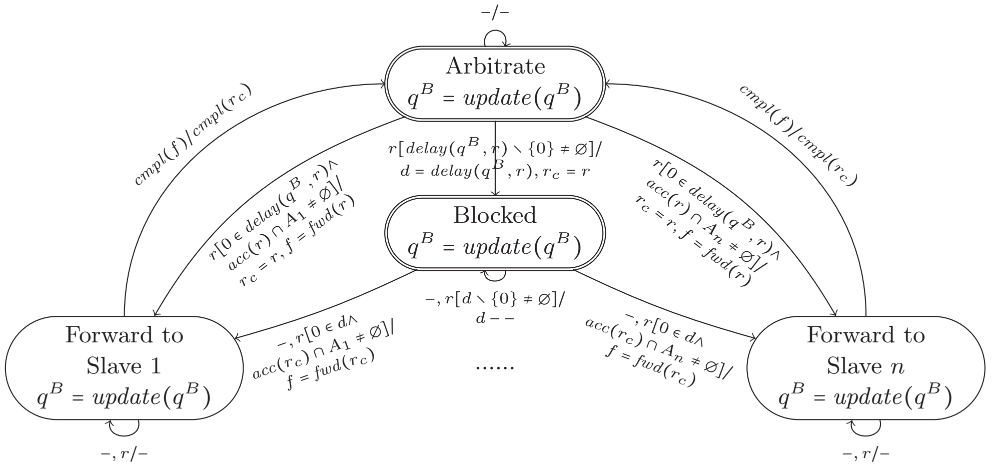
5.4.2 Shared Bus Analysis Preliminaries

As pointed out in Section 5.2.2, we focus on the analysis of shared buses, since shared interconnection networks can be reduced to this case for the important class of time-triggered arbitration. This analysis stage is inevitable in multi-cores if any shared memory is going to be used, whereas shared caches are not required in all cases. In fact, real-world architectures actually avoid even non-shared caches in areas where time-predictability and power-efficiency matters more than simplicity of programming such as smartphone MPSoCs and gaming consoles [WEE+08, Section 11.2]. To cite from the latter article:

> *"The complexity of analysis moves from the behavior of the individual cores to the interplay between them as they access memory."*
> [WEE+08]

In Section 5.2.2, we have already presented the four main types of arbitration that we will consider. They are listed in Table 5.1 together with a classification of analysis properties.

Definition 20. *A shared resource R has* bounded access delay *iff for every access request r at time t there exists a number of time steps $D^{max} \in \mathbb{N}_0$ such that r is guaranteed to have been granted access at time $t + D^{max}$. R is called* state-permeable *iff an access by a core c_i can lead to a change of the timing-behavior of the resource as observed by a core $c_j \neq c_i$.*


–/–
Arbitrate
$q^B = update(q^B)$
$cmpl(f)/cmpl(r_c)$
$cmpl(f)/cmpl(r_c)$
$r[delay(q^B, r) \setminus \{0\} \neq \emptyset]/$
$d = delay(q^B, r), r_c = r$
$r[0 \in delay(q^B, r) \wedge$
$acc(r) \cap A_1 \neq \emptyset]/$
$r_c = r, f = fwd(r)$
$r[0 \in delay(q^B, r) \wedge$
$acc(r) \cap A_n \neq \emptyset]/$
$r_c = r, f = fwd(r)$
Blocked
$q^B = update(q^B)$
$-, r[d \setminus \{0\} \neq \emptyset]/$
$d--$
$-, r[0 \in d \wedge$
$acc(r_c) \cap A_1 \neq \emptyset]/$
$f = fwd(r_c)$
$-, r[0 \in d \wedge$
$acc(r_c) \cap A_n \neq \emptyset]/$
$f = fwd(r_c)$
Forward to
Slave 1
$q^B = update(q^B)$
......
Forward to
Slave n
$q^B = update(q^B)$
$-, r/-$
$-, r/-$

Field	Type	Description
r	Input	An incoming request
cmpl	Input	Request completions
r_c	R	The current request
q^B	B	Abstract arbiter state
d	$\mathbb{I}_u$	Blocking duration
A_i	$\mathbb{I}_u$	Address range of slave i

Function	Description
cmpl	Signal access completion (Output)
fwd	Forward a request (Output)
delay	Possible access delay computation
update	Update bus state
acc	Determine request address range

Figure 5.2: The abstract bus timing model.

Resources without bounded access delay cannot be analyzed by a task- or core-partitioned WCET analysis, since the access opponents must be known to make the possible delay finite. Therefore, the PRIO arbitration cannot be analyzed with the techniques presented in this section.

The task- or core-partitioned analysis of a resource which is state-permeable but has bounded access delay is possible, but it will incur inevitable overestimation since the other cores' timing-relevant modifications of the resource state cannot be captured precisely. Therefore, shared caches, FAIR and PD arbitration can only be handled by worst-case assumptions. Section 5.5 presents a computationally more expensive methodology for precise analyses of resources with unbounded delay or state-permeability.

The ideal case, from the point of view of the analysis, is a bounded-delay non-state-permeable resource. The only known example for this are TDMA-arbitrated interconnects and buses. For such a resource, a precise analysis is possible, even without knowing the concurrently executing tasks. Therefore, the following sections will have an emphasis on the precise analysis of TDMA.

In Figure 5.2, the abstract state machine for a shared bus is shown in analogy to the abstract pipeline and cache FSMs from Section 4.3. Since we are conducting a task-partitioned analysis here, the input is a bus access request r from the current task or no access denoted as "–". No information is available about accesses which are possibly issued in parallel by other cores. If the bus is free (state "Arbitrate") the function $\mathit{delay}(q^B, r)$ is used to determine the maximum blocking duration caused by other tasks or the arbitration scheme. Depending on whether this duration may be zero, the transitions to "Blocked" and to slaves which are registered for addresses from the possibly accessed address range $\mathit{acc}(r)$ are enabled. As mentioned in Section 4.3, the microarchitectural analysis must explore all non-deterministic transitions if multiple ones are enabled. Once a request is completed at the slave side, which may be a shared cache or a shared memory in our architecture from Figure 3.4, the completion is signaled through $\mathit{cmpl}(f)$ and is forwarded to the master as $\mathit{cmpl}(r_c)$. Depending on the arbitration policy, the bus arbiter also keeps an internal state q^B which may be used in the arbitration decision and may be updated in each cycle through $\mathit{update}(q^B)$.

Similar to the cache analysis, with Q_B being the set of FSM states from Figure 5.2 and B being the set of abstract arbiter states we can define the *abstract bus domain* $\mathbb{B}$ which is part of the microarchitectural environment domain Q_E as

$$\mathbb{B} = 2^{Q_B} \times B \tag{5.4}$$

In the following we will explore different possibilities of implementing the domain B and its *delay* and *update* operations for the arbitration types from Table 5.1. These results were developed exclusively by the author of this thesis and a preliminary version of them was published in [KFM+11] and [KFM+14]. The generalization to the Priority Division policy was published by the author in [KHM+13].

5.4.3 Basic Bus Domains

As discussed in Section 5.2.2, for each shared bus a *maximum slave access duration* $T^B_{\max}$ is known, i.e., any transaction that was granted the bus must be completed within $T^B_{\max}$ cycles.

In the discussion of Definition 20 and Table 5.1, we have already seen that the PRIO arbitration cannot be analyzed by a partitioned analysis, since in general we cannot prove that access r, issued by the current task τ, does not suffer from starvation on the bus. Thus, for τ running on core c we set

$$B_{\text{PRIO}} = \{-\} \tag{5.5}$$

$$\forall r \in R : \mathit{delay}_{\text{PRIO}}(-, r) = \begin{cases} [0, T^B_{\max} - 1] & \text{if } \forall i \in \{0, \ldots, n_c\} : p_c \geq p_i \\ \infty & \text{else} \end{cases} \tag{5.6}$$

Even for the maximum priority core, a non-zero delay is possible since running accesses are not preemptable. Still, this is only of very limited use. We will therefore ignore PRIO for the rest of this section. A more precise analysis is presented in Section 5.5.

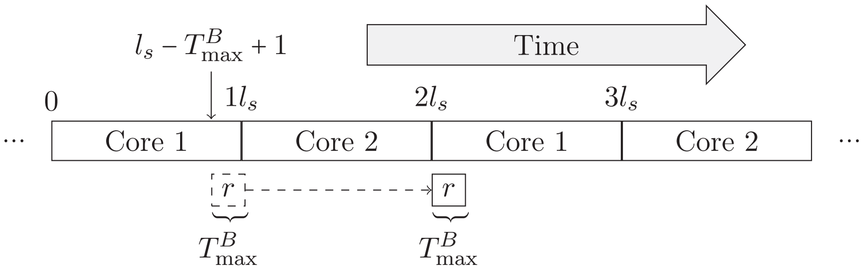


Figure 5.3: An example for a TDMA bus access which is maximally delayed.

Similarly to PRIO, the FAIR arbitration can also not be analyzed precisely here. But other than PRIO, it has a bounded access delay which prevents starvation of any access. We can therefore analyze it with

$$B_{\text{FAIR}} = \{-\} \tag{5.7}$$

$$\forall r \in R : \mathit{delay}_{\text{FAIR}}(-, r) = [0, (n_c - 1)T^B_{\max}] \tag{5.8}$$

where n_c is the number of cores. For an access from $\tau \in T_c$ the bus is free in the best-case (delay 0) whereas in the worst-case the active master m_a is equal to $(c+1) \bmod n_c$ and all cores $c_o \neq c$ have issued requests at the current cycle, too (delay $(n_c - 1)T^B_{\max}$). Since we have no information about other concurrent accesses, we must account for best- and worst-case and all in-between cases as shown above.

For TDMA, the worst-case happens when the access hits the bus at the first cycle where it cannot be granted any more, which is shown in Figure 5.3 for $n = 2$. The generalized worst-case delay amounts to $(n_l - 1)l_s + (T^B_{\max} - 1)$ cycles. Since we require the slots to be big enough to accommodate any access to a target behind the bus, i.e., $l_s \geq T^B_{\max}$, this worst-case is even *worse* than for FAIR. Fortunately though, if we assume that the access in Figure 5.3 is issued at time 0 instead of at time $l_s - T^B_{\max} + 1$, then we can infer that the delay must be equal to zero. So, we can determine the arbitration delay *without* knowing anything about possible concurrently occurring accesses under TDMA.

Definition 21. *The absolute time in the analyzed system is measured in CPU clock cycles*[1]*. An* absolute point in time *in an execution is given as $t \in \mathbb{N}_0$ which means the t-th clock cycle after the start of the system. The* offset *o of a point in time t is computed as $o = (t \bmod n_l l_s)$. An* offset set *$O \subseteq \mathbb{N}_0/(n_l l_s)$ is a set of offsets, i.e., a subset of the modulo ring $\mathbb{N}_0/(n_l l_s)$.*

We will represent the position in the cyclic TDMA schedule through offset sets in the following. Therefore, we set

$$B_{\text{TDMA}} = 2^{\mathbb{N}_0/(n_l l_s)} \tag{5.9}$$

[1] We assume a constant clock frequency.

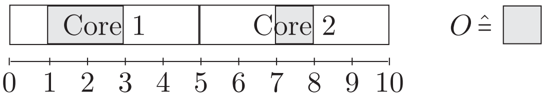


Figure 5.4: The TDMA offset set $O = \{1, 2, 7\}$ in a 2-core schedule with $l_s = 5$.

Effectively, this introduces the notion of "points in time" into the microarchitectural analysis itself, though only in the compressed form of offsets. In the classical single-core analysis frameworks, the step towards points in time as opposed to points in a state space is done only *after* the microarchitectural analysis has terminated as sketched in Section 4.3 and Section 4.4.

The TDMA delay function is dependent on the core c on which the currently analyzed task is executed. For all $q^B \in B_{\text{TDMA}}$ and $r \in R$ it is defined as

$$\mathit{delay}_{\text{TDMA}}(q^B, r) = \bigcup_{o \in (q^B + T_a)} \begin{cases} 0 & \text{if } o \in \gamma(c) \\ \min_{o_g \in \gamma(c)} \{o_g - o \mod n_l l_s\} & \text{else} \end{cases} \tag{5.10}$$

with a *grant window* of offsets $\gamma(c)$ for core c defined as

$$\gamma(c) = \bigcup_{i \text{ with } o_i = c} [i l_s, (i+1) l_s - T^B_{\max}] \tag{5.11}$$

The first case corresponds to an access inside of one of c's slots summarized through $\gamma(c)$, whereas the second case is an access outside of $\gamma(c)$. The equation uses $(q^B + T_a)$ as the *effective offset set* where T_a is the time that is needed for the arbitration itself. We generally assume $T_a = 1$, which was already reflected in Table 3.1 and by the fact that there is only one "Arbitrate" state in Figure 5.2 which has no self-loop. The shift by T_a is needed since o is the offset at the time of the *arbitration* whereas $o + T_a$ is the offset at the time of the *bus access.*

As an example for a delay computation consider the offset set given in Figure 5.4. Assuming $T^B_{\max} = 2$ and an access from core 1, i.e., $c = 0$, offsets 1 and 2 fall into case one and contribute delay 0, whereas offset 7 lies in case two and produces a delay of 3. Therefore, in this case, $\mathit{delay}_{\text{TDMA}}(\{1, 2, 7\}, r) = \{0, 3\}$.

In the "Arbitrate" and "Blocked" states from Figure 5.2 we can only register the passing of a CPU cycle, therefore the *update* for these states is

$$\forall q^B \in B : \mathit{update}^{\text{Default}}_{\text{TDMA}}(q^B) = \{o + 1 \bmod n_l l_s | o \in q^B\} \tag{5.12}$$

In the transition to the "Forward to Slave X" states we however know that the bus has been granted at the current cycle, i.e., we must be at an offset within the current core's slot. For the "Forward" states we define *update* on all $q^B \in B$ as

$$\mathit{update}^{\text{Forward}}_{\text{TDMA}}(q^B) = \{o + 1 \bmod n_l l_s | o \in q^B\} \cap \gamma(c) \tag{5.13}$$

To make B usable as the bus domain in the microarchitectural analysis, we finally need to provide a meet operator for B which is simply given by the set union:

$$\forall q^B_1, q^B_2 \in B : q^B_1 \sqcup q^B_2 = q^B_1 \cup q^B_2 \tag{5.14}$$

In the implementation, we represent the offset sets efficiently as sets of offset intervals, which speeds up most operations on them since instead of examining every single offset only each interval's lower and upper bound must be inspected or updated.

For the time-triggered priority division (PD) arbiter, we distinguish between an offset window $\gamma_{\text{MUST}}(c)$ in which accesses from c *must* be granted and a window $\gamma_{\text{MAY}}(c)$ in which accesses *may* be granted. The possible delay due to a non-preemptable lower-priority access must be accounted for in the determination of the grant window $\gamma_{\text{MUST}}(c)$, which is reflected by the subtraction of $(T^B_{\max} - 1)$ in Equation 5.15.

$$\gamma_{\text{MUST}}(c) = \bigcup_{i \text{ with } \forall j: p_{ic} \geq p_{ij}} [il_s, (i+1)l_s - T^B_{\max} - (T^B_{\max} - 1)] \tag{5.15}$$

$$\gamma_{\text{MAY}}(c) = \bigcup_{i \text{ with } p_{ic} > 0} [il_s, (i+1)l_s - T^B_{\max}] \tag{5.16}$$

For the delay function, the core which has maximum priority in a slot is subject to the same delay as in the PRIO case (see Equation 5.6). If a core has nonzero priority, it *may* be able to access, but it not guaranteed to do so, whereas if it has zero priority, it remains blocked. This leads to the delay function

$$delay_{\text{PD}}(q^B, r) = \bigcup_{o \in (q^B + T_a)} \begin{cases} [0, T^B_{\max} - 1] & \text{if } o \in \gamma_{\text{MUST}}(c) \\ [0, \min_{o_g \in \gamma_{\text{MUST}}(c)} \{o_g - o \mod n_l l_s\}] & \text{if } o \in \gamma_{\text{MAY}}(c) \\ \min_{o_g \in \gamma_{\text{MUST}}(c)} \{o_g - o \mod n_l l_s\} & \text{else} \end{cases} \tag{5.17}$$

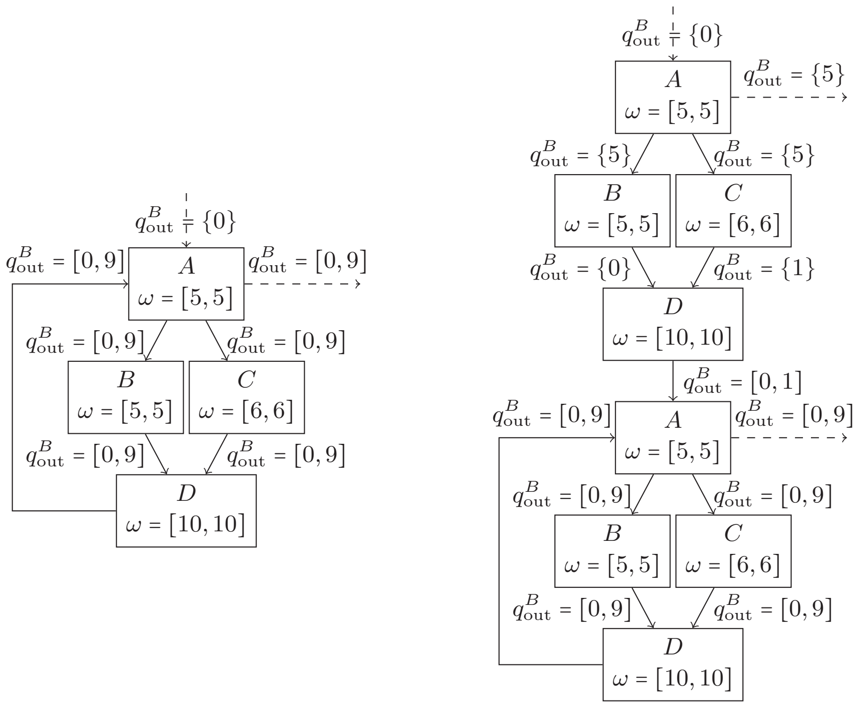
with $\gamma(c) = \gamma_{\text{MUST}}(c) \cup \gamma_{\text{MAY}}(c)$, the meet and update functions are the same as for TDMA.

For both the TDMA and PD analysis, we also still need the initial data-flow information q^B_0 (labeled l_0 in Definition 6). We could assume $q^B_0 = \{0, \ldots, n_l l_s\}$ but for the purpose of higher result precision, we assume that the tasks start synchronized with the TDMA schedule at offset 0, i.e., $q^B_0 = \{0\}$. In our platform, this is achieved by a memory-mapped *delay register* of the bus arbiter which delays the access until the offset $n_l l_s - 1$ is reached, such that the first instruction of a task starts at offset 0. However as noted above, this is completely optional and the impact of $q^B_0 = \{0, \ldots, n_l l_s\}$ on the precision is limited as discussed in the following.

With this framework, we can analyze time-triggered arbitration to some extent. One problematic aspect is, that once we have lost precision due to meet operations as specified in Equation 5.14, we can hardly regain it. The only point where we actually *gain* precision is in Equation 5.13 by intersecting with $\gamma(c)$. Thus, once we have reached a bus state $q^B \supseteq \gamma(c)$ we can no longer reach any q^B_{next} with $|q^B_{\text{next}}| < \gamma(c)$. This is undesirable, since any increase of the cardinality of q^B may introduce a new value into the union in Equation 5.10. To be able to reach smaller offset sets in the analysis again, we therefore have to extend it as detailed in the following sections.

5.4.4 Loop Unrolling

The main problem for most abstract interpretation-based analyses are loops, because the data-flow information is repeatedly joined together with the meet operator at the loop head until it stabilizes as sketched in Algorithm 1. As an example, consider the code fragment shown in Figure 5.5a. For simplicity of presentation, we assume a constant block execution time ω for each block and no offset refinement after Equation 5.13. With the given parameters $n_l = 2$ and $l_s = 5$, we have a schedule length of 10 cycles. The path $A \rightarrow B \rightarrow D$ has a total length of 20, i.e., any offset that enters at A is mirrored back to A from D. This is different for path $A \rightarrow C \rightarrow D$ with length 21. When entering with offset 0 at A, we reach C with offset 5 and D and A are then reached with offset 1. The meet operator at A yields $\{0\} \sqcup \{1\} = [0,1]$ which is again propagated through the loop. Therefore, in the next analysis iteration all offsets in $[0,9]$ are added to q^B_{out} of D in ascending order. Since $[0,9]$ is the "top" element $\top^B$ of the offset lattice, convergence is reached then.



$q^B_{\text{out}} \overset{\perp}{\top} \{0\}$
$q^B_{\text{out}} = [0,9]$
A
$\omega = [5,5]$
$q^B_{\text{out}} = [0,9]$
B
$\omega = [5,5]$
C
$\omega = [6,6]$
D
$\omega = [10,10]$
$q^B_{\text{out}} = \{5\}$
$q^B_{\text{out}} = \{0\}$
$q^B_{\text{out}} = \{1\}$
$q^B_{\text{out}} = [0,1]$

(a) The original loop with converged offsets.

(b) The unrolled loop with converged offsets.

Figure 5.5: An example for the divergence of TDMA offsets due to loops in the program with parameters $n_l = 2$ and $l_s = 5$.

The easiest way to avoid this behavior is to unroll the loop. Figure 5.5b shows an unrolling of the first iteration and the respective offset results as discussed above. As visible, only the unrolled iterations profit from this transformation, i.e., to obtain really precise results, we need to *fully unroll* the loop which was first proposed in [AEP+08] and extended in [KFM+11, "Global Convergence Analysis"]. However, this approach has a number of significant drawbacks:

- A full unrolling is possible only for those loops for which we have an explicit maximum loop iteration count (compare Section 3.3). For loops which are implicitly bounded by flow restrictions, this technique is not applicable.
- The unrolling may *severely* impair the analysis duration. Since the TDMA analysis only works as a part of the integrated microarchitectural analysis, the duration is scaled linearly with the loop bound for *all* modules including pipeline and cache analysis.

5.4.5 Offset Contexts

As already visible from Figure 5.5, the TDMA behavior is cyclic. Even though the loop from Figure 5.5a may have thousands of iterations, every iteration i will start with offset $i \mod n_l l_s$. Thus, to be precise with respect to the TDMA offsets, we only need to distinguish $n_l l_s$ different execution scenarios, i.e., one scenario for each offset with which the loop may start. Classical DFA unrolling contexts as presented in Section 4.1.2 form sequential chains of contexts. In the special case of TDMA analysis, we however need *cyclic* contexts whose interdependencies are statically unknown and only become clear *during* the analysis itself. We therefore introduce a new type of contexts, named *offset contexts*, into the analysis:

Definition 22. *An* offset context c^l_o *for an offset* $o \in B_{TDMA}$ *and a natural loop* $l \in \circlearrowleft_\tau$ *is a context* $G^{c^l_o}_\tau = (V^{c^l_o}_\tau, E^{c^l_o}_\tau, v^{c^l_o}_0)$ *within the context graph* G^C_τ *as defined in Definition 14 with* $v^{c^l_o}_0$ *being the head of loop* l*. The* sibling set $C^l = \{c^l_0, \dots, c^l_{n_l l_s - 1}\}$ *contains the offset contexts of a given loop for all possible offsets. Each sibling set is accompanied by an unrolled iteration context* $c^l_\perp$ *for the same loop which models the first loop iteration. The set of nodes* $v \in V^{c^l_o}_\tau$ *which are sources of back-edges is called* $V^{c^l}_{back} \subseteq V^{c^l_o}_\tau$*.*

This basic idea was first published by the author in [KFM+11], but based on a syntax-directed WCET analysis which worked by solving a separated graph-flow problem. We build upon this idea here, but we seamlessly integrate the offset analysis into the context graph itself.

An illustration of how offsets contexts for a loop are built, or *unfolded*, is shown in Figure 5.6. It shows the offset contexts that are unfolded when processing the loop from Figure 5.5a. To make the graph reasonably small, each node in Figure 5.6 represents one copy of the loop body, i.e., the nodes A, B, C, and D from Figure 5.5a. Every edge from one copy of the body to another one is a copy of the back edge (A, D). The first iteration of the loop is unrolled as presented previously. This is done to capture cache effects, since the first iteration of a loop will very often "warm up" the cache and we want to separate this behavior from the one of the successive

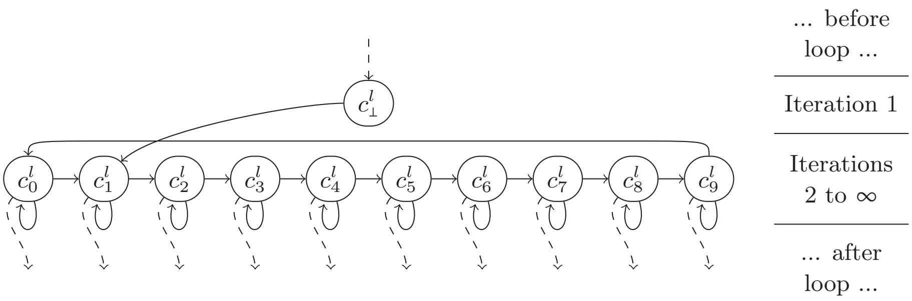


Figure 5.6: The unfolded offset contexts for the example shown in Figure 5.5a. The loop is assumed to iterate at least twice and each loop body has $\omega(l) = [20, 21]$.

iterations, modeled by the sibling set of offset contexts c_0^l to c_9^l. If the loop *may* exit after the first iteration (lower loop bound of 1), we also add a dashed exit edge from $c_\perp^l$. If it *must* exit after or before the first iteration (upper loop bound of 1 or 0), the unfolding does not alter the loop at all.

Unrolling and unfolding operations always work on the whole loop body. If a loop l contains a nested loop l', i.e., $l >_{\circlearrowleft} l'$, and l' has been either unrolled or unfolded previously, then all contexts of l' will be duplicated in the unrolling or unfolding of l. This leads to a considerable increase in the context graph size, but in the case of unfolding of offset contexts, this increase is bounded by the analysis parameters. In the case of unrolling, the increase is only limited by the loop bound values, which may be arbitrarily huge. Both techniques also affect the path analysis, since they transform the representation of the loop within the context graph. Therefore, Equation 4.46 and Equation 4.47 must summarize over all duplicates of the back-edges. In the case of offset contexts, only the edges that lead into $c_\perp^l$ are counted as *entry-edges* whereas all edges between $c_\perp^l$ and the c_o^l are *back-edges*.

Obviously, the graph structure as shown in Figure 5.6 is already part of the *result* of the offset analysis, because we initially have no idea which transitions between $c_\perp^l$ and the c_o^l are possible. Therefore, the analysis will start with no edge between $c_\perp^l$ and any c_o^l, i.e., all solid edges in Figure 5.6 are not added when the offset contexts are unfolded. They are added *during the microarchitectural analysis*, which requires some changes to the generic DFA algorithm. The resulting analysis algorithm is shown in Algorithm 6.

The main changes in comparison to the generic DFA work list algorithm depicted in Algorithm 1 are the lines 3–4, 11–12 and 16–17. In lines 3–4, the offset contexts are expanded for each natural loop as shown in Figure 5.6. No offset contexts are generated for non-reducible loops. As in the generic algorithm, the start node's incoming microarchitectural state is set to $q_0^m \in \mathbb{M}$ whereas all other nodes are initialized with $\perp \in \mathbb{M}$ (lines 5–7). The main analysis loop in line 8 iterates until all data-flow values – microarchitectural states in this case – have converged. To achieve this, the meet and transfer operators of the microarchitectural domain as

Algorithm 6 The offset context specific microarchitectural analysis.

```
1: function OffsetAwareDFA($((\mathbb{M}, \sqsubseteq), F)$, $(G^C_\tau, q^m_0)$)
2:     worklist ← $v^C_{0,\tau}$
3:     for $l \in \circlearrowleft_\tau$ in ascending order of $<_{\circlearrowleft}$ do        ▷ (Inner loops first)
4:         $unfoldOffsetContexts(l)$
5:     for $v \in V^C_\tau$ do        ▷ Initialization of the microarch. state ...
6:         if $v = v^C_{0,\tau}$ then $q^{in}_v \leftarrow q^m_0, q^{out}_v \leftarrow \bot$        ▷ ... for the start node ...
7:         else $q^{in}_v \leftarrow \bot, q^{out}_v \leftarrow \bot$        ▷ ... and for all other nodes.
8:     while worklist $\neq \varnothing$ do        ▷ Loop until a fixed point was found
9:         $v \leftarrow$ pop(worklist)
10:        $q^{in}_v \leftarrow \bigsqcup_{(u,v) \in E^C_\tau} q^{out}_u$
11:        if $v = v^{c^l_o}_0$ then        ▷ When offset context $c^l_o$ is entered ...
12:            $q^{B,in}_v = \{o\}$        ▷ ... set the incoming offset to $o$.
13:        $q^{tmp}_v \leftarrow f^{\mathbb{M}}_v(q^{in}_v)$        ▷ Apply microarch. transfer function of node $v$
14:        if $q^{out}_v \neq q^{tmp}_v$ then
15:            $q^{out}_v \leftarrow q^{tmp}_v$        ▷ Take over new outgoing value
16:            if $\exists c^l : v \in V^{c^l}_{\text{back}}$ then        ▷ For offset context back-edge sources ...
17:                $E^C_\tau \leftarrow E^C_\tau \cup \{(v, v^{c^l_o}_0) | o \in q^{B,out}_v\}$        ▷ add edges to target siblings.
18:            for $(v, w) \in E^C_\tau$ do
19:                push(worklist, $w$)        ▷ Propagate changes to all successors
20:    return $(G^C_\tau, \{v \rightarrow q^{in}_v \,|\, v \in V^C_\tau\})$
```

defined in Equation 4.23 and Equation 4.17 are invoked in lines 10 and 13 and new results are propagated through the graph in lines 18–19.

In comparison to the analysis from Section 5.4.3, the additional precision is gained by setting the incoming offsets to o if we reach the head node of an offset context c^l_o in line 12. This is valid, since context c^l_o exclusively models the situation that loop l is *entered with offset o*.

Of course, if we use offset contexts in this way, we must ensure that each node v which has an outgoing edge to the head node of l has edges to each offset context c^l_o for all $o \in q^{B,out}_v$. Since offset contexts are only formed for reducible loops, and the first loop iteration is unrolled[2], the transition into and between offset contexts is only possible via l's back-edges. We add copies of the back-edges leading to the respective offset contexts in lines 16–17. Edges into the first iteration, modeled by $c^l_\bot$, are already created during the unfolding. Therefore, the only edges that we add in lines 16–17 are edges between $c^l_\bot$ and the c^l_o and between different offset contexts $c^l_{o_1}$ and $c^l_{o_2}$, i.e., the solid edges from Figure 5.6.

[2] The unrolling is done to distinguish the cache behavior of the first iteration from that of the successive ones. Nevertheless the offset contexts in principle also work without the unrolling or with a different unrolling.

In this way, each offset context c_o^l which becomes reachable through the edge additions is analyzed using the existing microarchitectural analysis leading to $\omega(v)$ for each $v \in V_\tau^{c_o^l}$. The determination of the longest path through the context sibling set is done by the path analysis which uses the generated $\omega(v)$ values and the resulting context graph. The original formulation in [KFM+11; KFM+14] solved a dedicated dynamic flow problem for this purpose, but this is actually not needed as shown here.

Theorem 2. *The DFA with offset contexts as shown in Algorithm 6 yields correct microarchitectural states q_v^{in} for each $v \in V_\tau^C$ and terminates in finite time.*

Proof. If task τ contains no loops, Algorithm 6 coincides with Algorithm 1. In this case, the correctness follows from the Galois connection between $\mathbb{M}$ and the concrete microarchitectural states M (see Definition 17) and from the correctness of the transfer functions $f_v^{\mathbb{M}}$ (see Equation 4.17). The termination follows from the fact that the $f_v^{\mathbb{M}}$ are monotonic (cf. Section 2.1).

If τ contains loops, any data-flow item q_v^{in} that is generated by Algorithm 1 for a node $v \in V_\tau^{c^l_{[0,\infty]}}$ in the default loop iteration context $c^l_{[0,\infty]}$ is "distributed" among $c_\perp^l$ and the contexts c_0^l to $c^l_{n_l l_s - 1}$. Therefore,

$$\hat{q}^l = \bigsqcup_{c \in \{c_\perp^l, c_0^l, \dots, c^l_{n_l l_s - 1}\}} q_{v_c}^{in} \quad \wedge \quad \hat{q}^l \sqsupseteq q_v^{in} \tag{5.18}$$

where v_c is the copy of v in context c. Since all offset contexts have exit edges towards the loop successors, any loop successor obtains the value $\hat{q}^l \sqsupseteq q_v^{in}$ from the loop in its meet operation, i.e., the safety of the approximation is maintained for this node and all of its successors.

Finally, the termination of the analysis follows from the fact that we only *add* edges to the context graph in Algorithm 6 but we never remove them again. Thus, in the worst case each sibling set becomes a complete graph in the course of the analysis. No more graph changes are possible after this point. After the graph structure has converged, the termination follows from the same monotonicity argument as in the sequential or non-unfolded case. □

Note that the analysis using offset contexts is also feasible for tasks with irreducible CFGs, where some loops are no natural loops according to Definition 12. In this case, offset contexts are only constructed for the natural loops on the CFG, whereas non-natural loops are analyzed with the basic method from Section 5.4.3 without any unrolling or unfolding.

5.4.6 Offset Relocation

The previous approaches focused on tracking the offset development in more detail through the use of classical iteration contexts in the case of loop unrolling or cyclic offset contexts in the case of loop unfolding. The orthogonal approach of *relocating* offsets was first mentioned in [CRM10].

Definition 23. *An* offset relocation *to offset $o \in B$ at a block $v \in V_\tau^C$ means*

1. *setting $q_v^{B,in} = \{o\}$ and*
2. *artificially increasing $\omega_{max}(v)$ by $n_l l_s$ cycles.*

Thus, the relocation allows to *recover* from imprecise offset information by setting the incoming offsets of a block to a fixed value. This comes at the expense of analysis precision, since we have to add $n_l l_s$ to the block duration. It can only be beneficial if shared bus accesses follow in v or its successor blocks, which may be classified more precisely with the narrow offset value. If we have an imprecise $q_v^{B,in} = \{0, \ldots, n_l l_s - 1\}$ and multiple accesses to the bus, each of them may incur a delay of up to $(n_l - 1)l_s + T_{\max}^B - 1$. If the delay for each of these can be reduced, this can easily compensate the additional delay of $n_l l_s$ cycles for the offset relocation. A formal proof of correctness of this procedure and a discussion of it was first published by the author in [KFM+14].

To be able to reason about the validity of the relocation, we need to make some assumptions about the analyzed machine:

Definition 24. *Given a multi-core hardware platform with a TDMA-arbitrated shared resource R, a program L, an initial state $\tilde{q}_o^m \in \tilde{Q}_M$, an execution $exec(L, \tilde{q}_0^m) = (\tilde{q}_1^m, \ldots, \tilde{q}_k^m)$ (see Definition 16) and a state $\tilde{q}_i^m \in exec(L, \tilde{q}_0^m)$ that issues an access request to a shared resource R, the platform is* TDMA-compositional *if and only if*

- *an alternative execution $exec'(L, \tilde{q}_0^m) = (\tilde{q}_1^m, \ldots, \tilde{q}_i^m, \ldots, \tilde{q}_j^m)$ can be constructed such that the number of cycles for which R continuously stays in state "Blocked" (cf. Figure 5.2) after $\tilde{q}_i^m$ is longer in $exec'(L, \tilde{q}_0^m)$ than in $exec(L, \tilde{q}_0^m)$ and*
- *the change of length of the suffix $(\tilde{q}_i^m, \ldots, \tilde{q}_j^m) \subseteq exec'(L, \tilde{q}_0^m)$ compared to the suffix $(\tilde{q}_i^m, \ldots, \tilde{q}_k^m) \subseteq exec(L, \tilde{q}_0^m)$ is exclusively caused by R, i.e., the number of cycles that are spent in the "Blocked" state. In particular, all states from $(\tilde{q}_i^m, \ldots, \tilde{q}_k^m)$ in which no resource access is performed must be found in the suffix $(\tilde{q}_i^m, \ldots, \tilde{q}_j^m)$, too.*

More intuitively, TDMA-compositionality means that we can artificially delay the granting of an access and the only changes to the system timing that follow from this extra delay are changes of the arbitration delays of following accesses. Cycles which were devoted to computation or other pipeline actions are not affected. Other than the concept of *timing compositionality* mentioned in Section 2.2.9, TDMA-compositionality does *not* imply that the bus is analyzed in isolation from the other parts of the system. TDMA-compositionality merely eases the bus analysis as we will see in the following, but is otherwise unrelated to timing compositionality.

A typical example of a TDMA-compositional platform are platforms with strict in-order pipelines where the pipeline is stalled until an instruction is completed. In this case, the following actions of the pipeline are not affected by the extra delay introduced into the arbitration, except for those which try to access the resource again. Our ARM7TDMI-based system from Section 3.4 is an example of such a platform.

A counterexample are superscalar out-of-order processors in which resource accesses may overlap in time with other computations. In such a case, the following instructions may be executed in parallel to the delayed resource access, therefore contradicting with the last sentence of Definition 24.

The above example for TDMA-compositionality was also a timing-anomaly-free system. An example of a system which is TDMA-compositional but *not* free from timing anomalies is a platform with in-order pipelines, caches and branch prediction. Reineke et al. [RWT+06] have shown this combination to suffer from timing anomalies. Still, the system is TDMA-compositional as long as strict in-order cores are used.

In the following, we will show how TDMA-compositionality can be used to enhance the precision of the offset analyses for TDMA-compositional systems.

Lemma 1. ***(Offset Relocation Lemma)*** *Given a context graph $G^C_\tau = \left(V^C_\tau, E^C_\tau\right)$ and a path P through G^C_τ, two executions of the path P, one starting at TDMA offset o_1 and the other starting at TDMA offset o_2 with the initial state of all other hardware components being identical between the two executions, will lead to a difference in execution time of at most $n_l l_s$ cycles between the two execution scenarios if a TDMA-compositional platform is used.*

Proof. The execution of the path P is defined as a sequence of states $\left(\tilde{q}^m_0, \dots, \tilde{q}^m_k\right)$. We transform this into a sequence of *events* $S = (s_0, \dots, s_k)$ such that all contiguous groups of states which model the processing of an access to the shared resource are represented by a single s_i and all contiguous states which do not model a shared resource access are likewise represented by a s_j. Thus, each $s \in S$ is either an access to the shared resource or a block of local computations. We therefore define a set of accesses A and a set of "processing" blocks B such that $\forall s \in S : s \in A \oplus s \in B$, where $\oplus$ is the logical *exclusive-or*. For each access $s_i \in A$, we define the time from the access request to the access grant as δ_i and the time that it takes to perform the access as γ_i. Similarly, we define the runtime of each $s_i \in B$ as α_i. Due to the TDMA-compositionality we know that α_i and γ_i are constant among the two execution scenarios. Thus, what changes between the scenarios are the δ_i values but not the execution times of local computations or the resource accesses themselves.

To simplify our computations below, we allow $s_i \in B$ to have length 0, that is $\alpha_i = 0$. In this way, we can ensure that each pair of accesses is separated by a block of computation, though this block may have length 0 ($\forall j \in 0, \dots, i-1 : s_j \in A \Leftrightarrow s_{j+1} \in B$). Without loss of generality, we assume $s_0 \in A$.

The arbitration delay δ_j that an access $s_j \in A$ incurs, will change among the two execution scenarios. The arbitration delay that is incurred by access s_j in the execution scenario starting at offset o_1 (o_2) will be denoted by δ^1_j (δ^2_j), respectively.

In the same way, we will refer to the absolute point in time at which the access request is issued in the two scenarios as β_j^1 and β_j^2. What we would like to prove now, is:

$$\forall j \in \{0, 2, 4, \dots\} : |\beta_j^1 - \beta_j^2| \leq n_l l_s \tag{5.19}$$

Note that the Lemma is equivalent to $|\beta_k^1 - \beta_k^2| \leq n_l l_s$ where k is the last execution event. For $s_k \in A$, Equation 5.19 with $j = k$ is equivalent to the Lemma, and for $s_k \in B$, the Lemma follows from Equation 5.19 with $j = k - 1$ and the constant runtime of s_k. Therefore, it is sufficient to prove that Equation 5.19 holds.

To do this, we first define the dependencies among the β and δ values, which are as follows:

$$\forall j \in \{0, 2, 4, \dots\} : \beta_j = \beta_{j-2} + \delta_{j-2} + \gamma_{j-2} + \alpha_{j-1} \tag{5.20}$$

$$\forall j \in \{0, 2, 4, \dots\} : \delta_j = \mathit{delay}_{\text{TDMA}}(\beta_j, s_j) + \gamma_j \mod n_l l_s \tag{5.21}$$

In Equation 5.20, the access request time for s_j is computed as the sum of

- β_{j-2} – the request time of the last access (in s_{j-2})
- δ_{j-2} – the arbitration delay that this request incurs
- γ_{j-2} – the access duration itself
- and α_{j-1}, the duration of the consecutive block of computation s_{j-1}

The value for δ_j is then easily derived in Equation 5.21 from β_j with the usual TDMA arbitration computation via the known $\mathit{delay}_{\text{TDMA}}$ function from Equation 5.10. These dependencies are valid for both execution scenarios since they model the execution of the path P. The only variability comes from the accesses to the shared resource, for which the waiting time δ_j is computed using the known $\mathit{delay}_{\text{TDMA}}$ function. To initialize the computation, we must fulfill the conditions

$$\beta_0^1 = o_1 \mod n_l l_s \tag{5.22}$$

$$\beta_0^2 = o_2 \mod n_l l_s \tag{5.23}$$

Since we are finally only interested in the difference $\Delta_j = |\beta_j^1 - \beta_j^2|$, we simply set $\beta_0^1 := o_1$ and $\beta_0^2 := o_2$. We prove Equation 5.19 by induction over the sequence S.

Base case ($j = 0$): In this case, we have $\Delta_0 = |\beta_0^1 - \beta_0^2| = |o_1 - o_2|$ which is less than $n_l l_s$ by definition of the TDMA offsets.

Induction step ($j - 2 \to j$): By inserting the definitions of β_j into Δ_j we obtain:

$$\Delta_j = \left| \left(\beta_{j-2}^1 + \delta_{j-2}^1 + \gamma_{j-2} + \alpha_{j-1}\right) - \left(\beta_{j-2}^2 + \delta_{j-2}^2 + \gamma_{j-2} + \alpha_{j-1}\right) \right| \tag{5.24}$$

$$= \left| \left(\beta_{j-2}^1 + \delta_{j-2}^1\right) - \left(\beta_{j-2}^2 + \delta_{j-2}^2\right) \right| \tag{5.25}$$

$$= \left| \left(\xi_{j-2}^1 - \xi_{j-2}^2\right) \right| \tag{5.26}$$

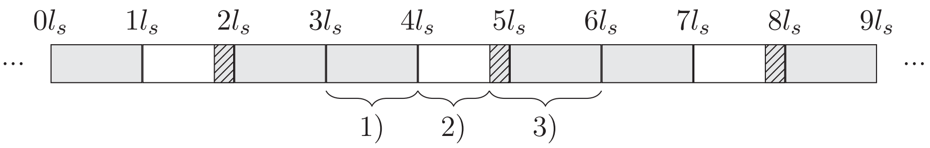


Figure 5.7: Illustration of proof scenario for the Relocation Lemma.

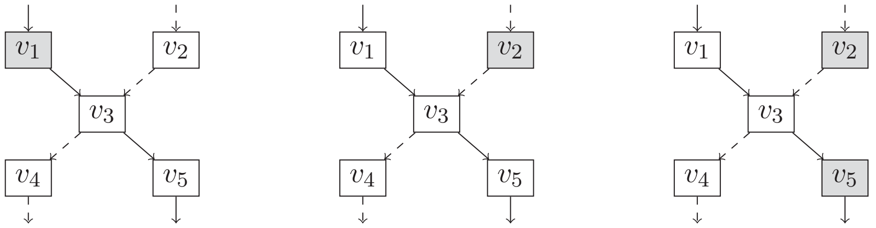
where ξ_j is a shorthand for $\beta_j + \delta_j$, i.e., the time when the access is granted. Obviously, Δ_j only depends on the access request times and waiting times of the preceding resource access. From the induction hypothesis we know that $\Delta_{j-2} \le n_l l_s$, that is, the preceding access is requested in a window of size $n_l l_s$ cycles in both execution scenarios. What we then have to show is, that both accesses are granted in a new window of size $n_l l_s$ cycles.

Figure 5.7 illustrates a scenario from the perspective of the second core in a configuration with three cores in total. Three TDMA periods are shown in the Figure. The gray areas are those where an access request from core two will have to wait, and the white ones are the areas where an access request from core two will be granted immediately. The hatched area represents the cycles from $T^B_{\max} - 1$ cycles before the slot end until the slot end in which accesses are not granted as explained in Figure 5.3. Due to the cyclicity of the TDMA schedule, it is sufficient to assume that $\beta^1_{j-2} \in [3l_s, 6l_s)$ and to place β^2_{j-2} at positions which are at most $n_l l_s$ cycles away from β^1_{j-2}. We then have to prove that $|\left(\xi^1_{j-2} - \xi^2_{j-2}\right)| \le n_l l_s$ holds. We will examine the individual cases with the help of the example, since this is more intuitive. The notation could nevertheless be generalized to describe the cases in a fully abstract way – in this case, only the position and size of the white box will change in all TDMA periods. All three following cases are also marked with their case number in Figure 5.7:

1) $\beta^1_{j-2} \in [3l_s, 4l_s - 1]$: Here, we have to wait for the core's slot, thus $\xi^1_{j-2} = 4l_s$. If β^2_{j-2} is in another gray area, then ξ^2_{j-2} will be either $1l_s$, $4l_s$ or $7l_s$. If β^2_{j-2} is in a white area, then it follows that $\xi^2_{j-2} \in \left[1l_s, 2l_s - (T^B_{\max} - 1)\right] \cup \left[4l_s, 5l_s - (T^B_{\max} - 1)\right]$. In all cases, $|\xi^1_{j-2} - \xi^2_{j-2}| \le n_l l_s$ holds.
2) $\beta^1_{j-2} \in \left[4l_s, 5l_s - (T^B_{\max} - 1)\right]$: Since the access can be granted immediately here, we have $\xi^1_{j-2} \in \left[4l_s, 5l_s - (T^B_{\max} - 1)\right]$. If β^2_{j-2} is in a gray area, then ξ^2_{j-2} will be either $4l_s$ or $7l_s$. In this case, $|\xi^1_{j-2} - \xi^2_{j-2}| \le n_l l_s$ directly holds. For the case that β^2_{j-2} is in a white area, the Lemma follows from the induction hypothesis that $|\beta^1_{j-2} - \beta^2_{j-2}|$ since in this case $\xi_{j-2} = \beta_{j-2}$.
3) $\beta^1_{j-2} \in \left[5l_s + (T^B_{\max} - 1), 6l_s - 1\right]$: This case is symmetrical to the first one.

Since in all these cases $|\xi^1_{j-2} - \xi^2_{j-2}| \le n_l l_s$ holds and we known from Equation 5.26 that $\Delta_j = |\left(\xi^1_{j-2} - \xi^2_{j-2}\right)|$, the induction step is complete and the lemma is proven. □

Lemma 1 provides the background of why we may safely use the offset relocation from Definition 23 in TDMA-compositional systems. During the analysis, we may



(a) Relocation applied on WCEP **(b)** Relocation applied outside of WCEP **(c)** Relocation applied outside of and on WCEP

Figure 5.8: Different scenarios for applying the offset relocation heuristic.

judiciously apply the relocation before line 13 in Algorithm 6. We do not determine in advance what happens "inside" a context block, i.e., whether contains shared bus accesses first or computation cycles first and therefore, we always relocate to offset 0. Also, we must apply the relocation sparsely since every application may incur a loss of precision. Therefore, we heuristically only apply the relocation to 0 at loop heads of loops which actually contain shared bus accesses. Whether a loop may potentially access the bus or not can be verified by inspecting the possible memory access targets determined by the value analysis. Note that this is only a heuristic, we could also apply the Lemma at arbitrary other program points.

With the offset relocation, we are no longer over-approximating the set of possible offsets. Nevertheless, we are still generating safe WCET overestimations and the analysis is still guaranteed to terminate, which we will both show in the following.

Theorem 3. *For a $WCET_{est}$ which is determined by the path analysis using the $\omega(v)$-values determined by the microarchitectural analysis with offset relocations, $WCET_{est} \geq WCET_{real}$ holds.*

Proof. For cases where no offset relocations were applied, $WCET_{\text{est}} \geq WCET_{\text{real}}$ follows from the correctness arguments presented in Section 4.4.

For cases where relocations were done, Figure 5.8 shows the possible relocation scenarios. In each sub-figure, the offset relocation was applied at the gray blocks, all white blocks have their offsets computed without offset relocation. The solid arrows mark the WCEP, whereas the dashed arrows represent control-flow edges which are not part of the WCEP $P = (b_1, b_2, \ldots, b_n)$. Note that the WCEP is unveiled by the path analysis, which runs *after* the microarchitectural analysis, including the bus analysis. We consider a single application of the heuristic at a node v_A and distinguish three cases:

- The WCEP contains v_A: This case is illustrated in Figure 5.8a with $v_A = v_1$. Since we applied the heuristic, we added a penalty of $n_l l_s$ cycles to v_A's WCET. Thus, the correctness of the computed WCET follows from Lemma 1.
- The WCEP does not contain v_A: This case is shown in Figure 5.8b and Figure 5.8c with $v_A = v_2$. Then, the application of the heuristic may only affect

the WCET results due to merged-in offset information, when there is a path in the CFG from v_A to any node b_w on the WCEP $P = \left(v_1^P, v_2^P, \ldots, v_n^P\right)$, e.g., path (v_2, v_3) in Figure 5.8b and Figure 5.8c. This can only lead to additional offsets in the offset set of b_w. Again we have two cases:

- If there is no further node $v_y \in P_{\text{tail}}$ with $P_{\text{tail}} = \left\{v_i^P | w \leq i \leq n\right\} \subseteq P$ where the heuristic was applied too (e.g. as in Figure 5.8b), then these additional offsets will only make the WCET results for the blocks in P_{tail} less precise, but they do not endanger their safeness. This is guaranteed since the microarchitectural transfer function, which is then applied for all blocks in P_{tail}, is monotonic and thus all original offsets, coming from blocks on the WCEP, are conserved.
- If such a node v_y does exist (e.g. v_5 in Figure 5.8c), then the application of the heuristic renders the offset set with which v_y is reached irrelevant, since the offset result for v_y will be relocated to 0 anyways. Therefore, the correctness of the WCET also follows in this case.

Thus, the safeness of the WCET is retained in all cases. □

An example for how the relocation can be used to generate more precise WCET values can be found in Figure 5.9. The code shown between the dotted horizontal lines is a loop body that was unrolled once. In this case, the upper half of the figure shows the analysis of the first iteration and the lower half of the figure shows the analysis of the following iterations of the loop. The figure shows the outgoing offsets $q_v^{B,out}$ for each block v on the left and right hand side of each arc. In addition, using the notation of Lemma 1, it shows the arbitration delay δ_i for each shared resource access s_i and the runtime α_i for each block of computation s_i. The analysis run A, shown in black on the left side, starts with the information that the top block may be entered with offsets $\{0, \ldots, 9\}$ and obtains a total arbitration delay of $14 + 10(X-1)$ if the relocation heuristic is applied at node s_1 and X is the loop bound of the analyzed loop. In contrast, the analysis run B, shown in gray on the right side, starts with the more precise offset information $\{2\}$ and obtains a WCET of $12X$ if the relocation heuristic is not applied. Obviously analysis run A will produce a more precise WCET result than analysis run B for all $X \geq 3$, even though it started with more imprecise offset information.

The existing termination guarantee of the analysis was given through the monotonicity of the microarchitectural transfer functions. Setting the outgoing offset set to $q_v^{B,out} = 0 + \omega(v) \mod n_l l_s$ as done by the relocation is obviously monotonic in $q_v^{B,in}$ since it does not even depend on $q_v^{B,in}$. Therefore, the termination is also guaranteed with the offset relocation.

5.4.7 Timing-Anomaly-Free Analysis

Similar to the single-core case, we can use the fact that a platform is free of timing-anomalies to reduce the outgoing microarchitectural states of a context block to

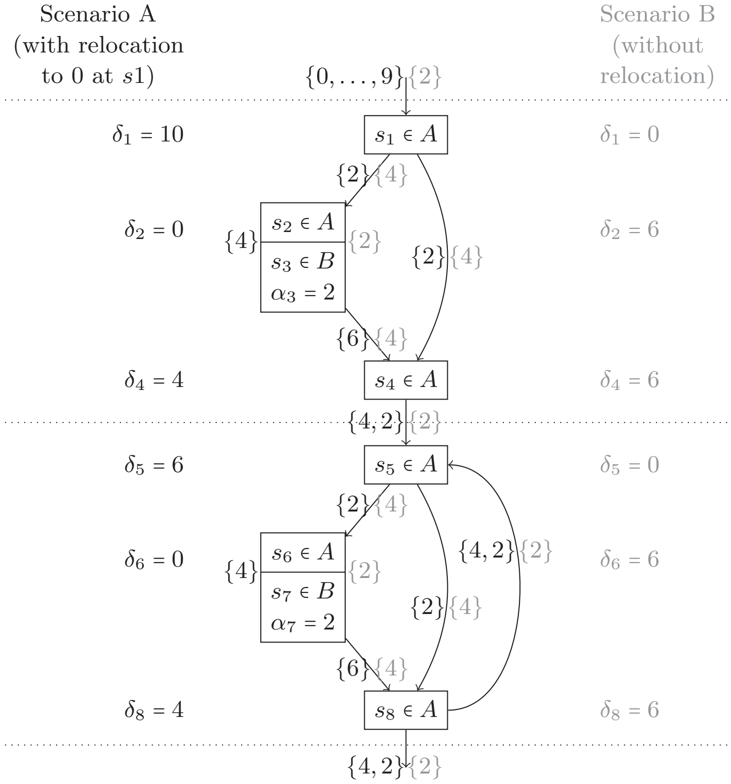
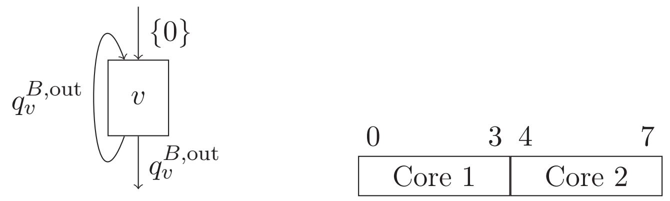


Figure 5.9: An example of the application of the offset relocation in a 2-core system with sloth length $l_s = 5$ and access duration $\gamma_i = 2$.

those which represent local worst-case behavior. Following the notation of Section 4.3 and the example from Figure 4.8, the analysis that uses the timing-anomaly-freedom (called *TA-free* in the following) can safely discard all outgoing states $q^M \in q_v^{out}$ of a context node $v \in V_\tau^C$ which are not maximizing $\omega(v)$ as defined in Definition 19. In Figure 4.8, this implies that only the lower-right, gray out-state needs to be retained in q_v^{out}, whereas all others can be dropped. Since every $q^M \in q_v^{out}$ may contribute different result offsets, this obviously increases the analysis precision.

However, this breaks one important prerequisite that was required for any DFA framework in Definition 5, namely that the transfer functions of the framework must be *monotonic*. If we are consequently over-approximating the offset sets as done in Section 5.4.3, the $q_v^{B,in}$ and $q_v^{B,out}$ for all blocks v will monotonically grow during the analysis. If we prune away the local non-worst-case successor state, we no longer have this guarantee and, therefore, the data-flow results are no longer guaranteed to converge.

An example for such a behavior is given in Figure 5.10a. We assume that the block v is reached with a single microarchitectural state $q^{M,\text{in}} \in q_v^{\mathbb{M},\text{in}}$. As shown in Section 4.3, the microarchitectural analysis, which includes the TDMA offset



(a) An example task executing on core 1.

(b) The TDMA schedule for the example.

Figure 5.10: An example of non-converging TDMA offset results when naively exploiting the absence of timing-anomalies.

analysis, then proceeds by applying the transfer function from Equation 4.17 to $q^{M,\text{in}}$ to generate the outgoing data-flow value $q_v^{\mathbb{M},\text{out}}$. According to Definition 17 and Definition 16 this transfer is done by performing cycle steps on the abstract state and gathering all reachable result states, as sketched in Figure 4.8. For the example, further assume that the first cycle step in $q^{M,\text{in}}$ issues a shared bus request, and that once the request is granted, it takes 3 cycles to complete. Then, additional 5 cycles are spent on pipeline computations in v. We further assume that the abstract bus state, i.e., the TDMA offset set, $q_v^{B,\text{in}} \in q^{M,\text{in}}$ is equal to $\{0\}$ and that the system has two cores and slot length 4 as shown in Figure 5.10b. The microarchitectural analysis will repeatedly visit the node v, merge the incoming offset information at the head of v using Equation 4.23 and recompute the outgoing offsets using the microarchitectural transfer function as mentioned above. For simplicity, we assume that during the whole process the pipeline does not show non-deterministic behavior, i.e., that $q^{\mathbb{M},\text{in}} = \{q^{M,\text{in}}\}$ and $q^{\mathbb{M},\text{out}} = \{q^{M,\text{out}}\}$ stay valid and only the value of $q^{M,\text{in}}$ and $q^{M,\text{out}}$ changes. With $q_v^{B,\text{in}}$ and $q_v^{B,\text{out}}$ we denote the TDMA bus states which are part of $q^{M,\text{in}}$ and $q^{M,\text{out}}$.

1. In the first iteration, $q_v^{B,\text{in}} = \{0\} \cup \varnothing$ since initially, $q_v^{B,\text{out}} = \varnothing$. The access in v is then issued at offset 0, granted at offset 1 and completed at offset 4. This leads to a runtime of 9 cycles for v, and thus to a $q_v^{B,\text{out}} = \{0+9 \mod 8\} = \{1\}$.
2. Now we have $q_v^{B,\text{in}} = \{0\} \cup \{1\} = \{0,1\}$. The execution scenario for offset 0 stays the same, but the execution with start offset 1 incurs a 7-cycle delay before the access can be granted, leading to a runtime of 15 cycles. Thus, the scenario with offset 1 is the local worst-case and therefore $q_v^{B,\text{out}} = \{1 + 15 \mod 8\} = \{0\}$.
3. In this analysis iteration, we again have $q_v^{B,\text{in}} = \{0\}$ as in step 1. Therefore, steps 1 and 2 will repeat infinitely often from here on.

Since the data-flow information $q_v^{B,\text{out}}$ never stabilizes in this example, an analysis of TDMA offsets which naively exploits timing-anomaly-freedom to cut down its search space will not terminate in general.

Obviously, these convergence problems only occur due to progressions of offset information at loop heads as we have seen in the example. All of the approaches presented in Section 5.4.4, Section 5.4.5 and Section 5.4.6 eliminate this problem. The unrolling removes all back-edges and the offset contexts and offset relocation lead to a constant incoming offset at the loop head. Therefore, a partitioned WCET analysis using TDMA offset analysis *and* the pruning of local non-worst-case states is *only* feasible if either the full unrolling, offset context or offset relocation approach is used.

5.4.8 Evaluation

In the following, we present an evaluation of the performance of the presented analysis techniques and of the different arbitration strategies in comparison to each other. Parts of this evaluation were published in [KHM+13] and [KFM+11; KFM+14]. In the two latter publications, the experiments were done for a platform based on SimpleScalar cores instead of ARM ones.

Due to the lack of standard multicore real-time benchmarks, we chose to execute independent tasks from the benchmark suites already mentioned in Section 4.5 on the single cores, amounting to 154 flow-fact-annotated, independent *benchmark tasks* in total. In the experiments, we grouped together benchmarks with similar runtime and executed *packages* with one benchmark per core. The packages were formed by sorting the benchmarks in the order of their single-core ACET and then having a window of size 2/4/8 slide over this list, collecting all 153/151/147 possible combinations. All cores start their assigned task synchronously and execution finishes when all tasks have been completed. Thus, since the benchmarks have different runtimes, there will be some amount of inevitable *completion time jitter*. All the benchmarks read their inputs and store their outputs in the shared non-cached RAM, whereas all program code as well as the stack was allocated in the scratchpad memory of the individual cores. This emulates the (reasonable) scenario that I/O is done via a shared device, whereas code and local data are kept in local memories for performance reasons. As in Section 4.5, all benchmarks were compiled without further compiler optimization (optimization level O0). This is frequently done for safety-critical code, due to the fear of errors introduced by potentially incorrect compiler optimizations. All other parameters were set to their default values according to Table 3.1.

Concerning the parametrization of the arbitration methods we have selected simple heuristics to demonstrate some key impacts. For PRIO, the priorities were assigned such that $t_i > t_j \Leftrightarrow p_i > p_j$ where t_i is the single-core runtime of the task mapped to core i. We use this strategy, also known as *largest job first*, here to speed up long-running tasks and thus to decrease the completion time jitter. For TDMA (and also for PD) we set the slot size $l_s = T^B_{\max}$ to keep delay times as small as possible. Our experiments have shown that higher slot lengths most often impose both higher WCET and ACET values. Also, for TDMA we set $o_i = i$ such that each core owns a single slot. For PD, each slot i is "owned" by core i by setting $p_{ii} = n$. Priorities for all other cores are distributed in the same way as for PRIO,

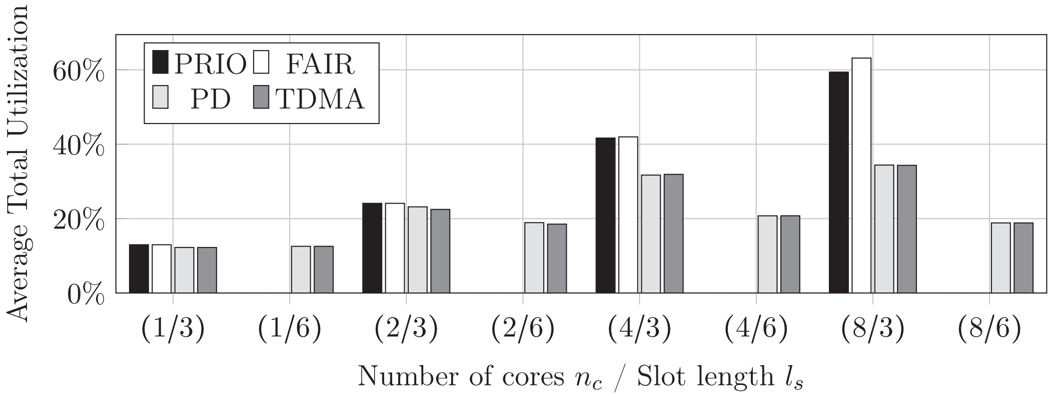


Figure 5.11: Average total bus utilization.

i.e., in the order of single-core task runtime. In the experiments, these values proved to be good default values. The impact of different configuration options and their optimization will be further explored in Section 6.1.

Runtime properties

Before delving into the WCET results, we will examine some runtime aspects of the benchmarks which are needed to judge the WCET results.

Figure 5.11 shows the geometrical mean utilization resulting for different values of n_c and l_s. Here, and in the following, the results for FAIR and PRIO are only given for $l_s = 3$ since they do not use the concept of a slot length at all. As expected, FAIR and PRIO show superior utilization, since these are *work-conserving* arbitration methods, i.e., as long as there are active requests, they do not insert wait cycles. TDMA shows some increase in utilization with rising n_c, but it is stagnating at around 20% due to slots which remain unused by their owners. For $n_c = 8$ the utilization is actually *decreasing* again below 20%. PD is also not work-conserving, since it must delay requests when they cannot be served in the current slot, which may happen frequently for our setting of $l_s = T^B_{\max}$. Still, PD shows a linear increase in utilization, which is twice as high as for TDMA, which is also reflected in the average ACETs of the benchmarks as shown in Figure 5.12.

In general, the ACET per task is inversely proportional to the achieved utilization values. The dotted areas in Figure 5.12 show the portion of the ACET which is used for computation and local memory accesses (stack and program code, see Section 3.4), the crosshatched areas show the portion in which the task is *using* the shared bus, and the areas with vertical bars show the percentage of the ACET in which the task is *waiting* for the shared bus. For TDMA it becomes visible that, e.g., for the configuration with 8 cores, the tasks are on average using more cycles for waiting than for performing computation and actual memory accesses.

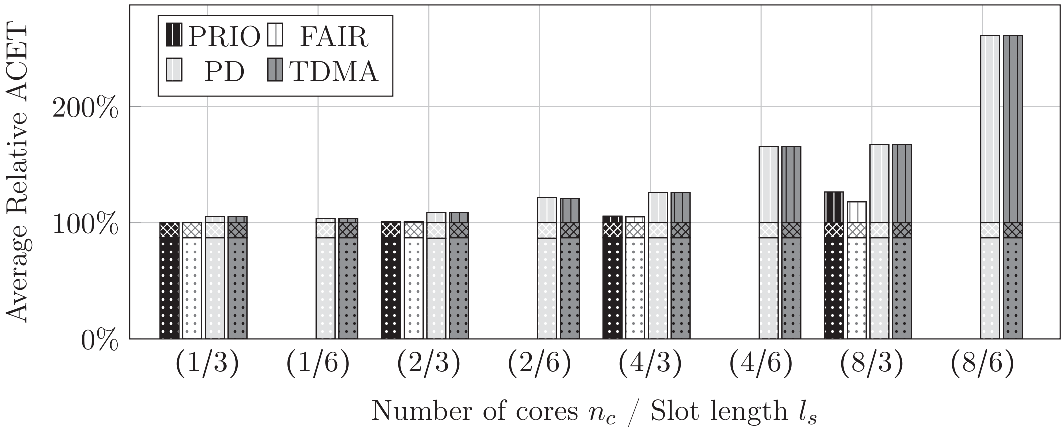


Figure 5.12: Average relative measured execution time (ACET) for different platforms (Baseline = execution time on single-core platform).

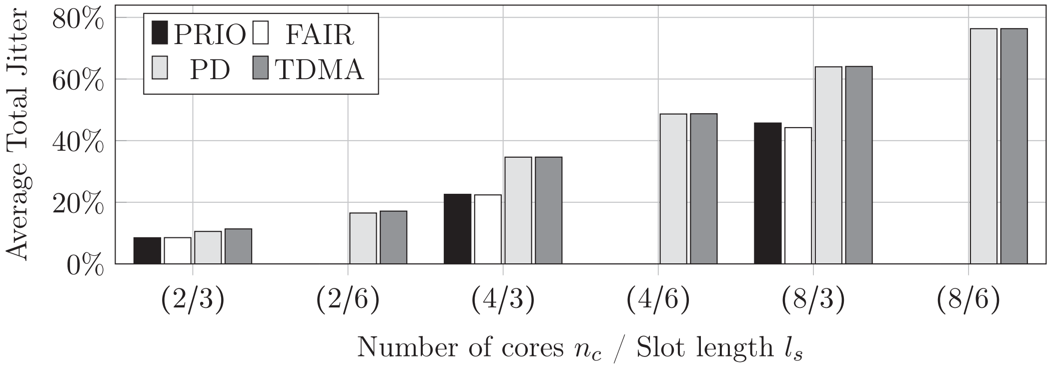


Figure 5.13: Average benchmark execution time jitter.

The ACET and utilization values are influenced by the completion time jitter of the benchmark packages, that is the length of the time interval between the first termination of a task on any core and the termination of the last task. Especially for TDMA, the jitter is problematic since it leaves slots of already terminated tasks unused. Figure 5.13 shows the jitter as a percentage of the total runtime of the benchmark package (i.e., the runtime of the longest task). It is visible that the low utilization values for TDMA are to some extent related to the rising jitter, but since this increase is itself triggered by the usage of TDMA, this is an inherent drawback of the policy. As an example, in the case of 8 cores this means, that after 21% of the total benchmark runtime the first task terminates, which leaves the remaining 79% of the slots of this task unused.

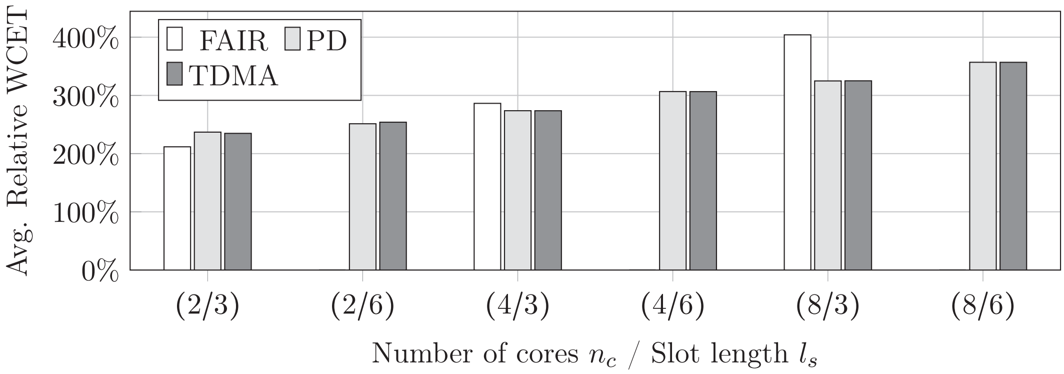


Figure 5.14: Average relative WCET when all bus accesses show the worst-case bus behavior.

WCET results

In all of the following experiments, we had to restrict the maximum analysis duration to two hours for a single benchmark, and the maximum memory consumption was implicitly limited by the fact that the analyzer was compiled in 32-bit mode, i.e., only 4 GB of main memory were available. The full loop unrolling approach from Section 5.4.4 was only able to analyze 30.9% of the benchmarks. In all other cases, it ran out of either time or memory. Therefore, we restricted our experiments to these 30.9% of the benchmarks which worked also using full unrolling. Due to this, the average WCETs obtained in this chapter are not directly comparable to those from Section 4.5, since we use a different benchmark base.

We will first examine the worst-case behavior of the arbitration policies, since this will give us one possible baseline to compare against. As in Section 4.5, we present only *relative WCET* values, i.e., the WCET divided by the ACET of the benchmark, since these relative WCETs are a bound on the overestimation of the analysis. The results for an analysis in which the worst-case bus behavior is assumed for every single access are shown in Figure 5.14. As expected, the WCETs increase linearly with the length of the schedule in the TDMA case, and linearly with the number of cores in the system in the FAIR case. WCET results for PRIO were not generated, since the task-partitioned analysis can only provide a WCET for the tasks running on the highest-priority core as detailed in Section 5.4.2. For all other tasks, the WCET using PRIO is infinite. Therefore, we excluded PRIO from the WCET comparison.

The results for a WCET analysis using the basic TDMA analysis as shown in Section 5.4.3 are presented in Figure 5.15. Here, we have not used the fact that our platform is free of timing anomalies, i.e., this is the TA-prone scenario as sketched in Section 5.4.7. The results show, that the FAIR results are almost equal to the worst-case results from Figure 5.14. We could not expect better results here, since the analysis itself is forced to make worst-case assumptions (cf. Equation 5.7).

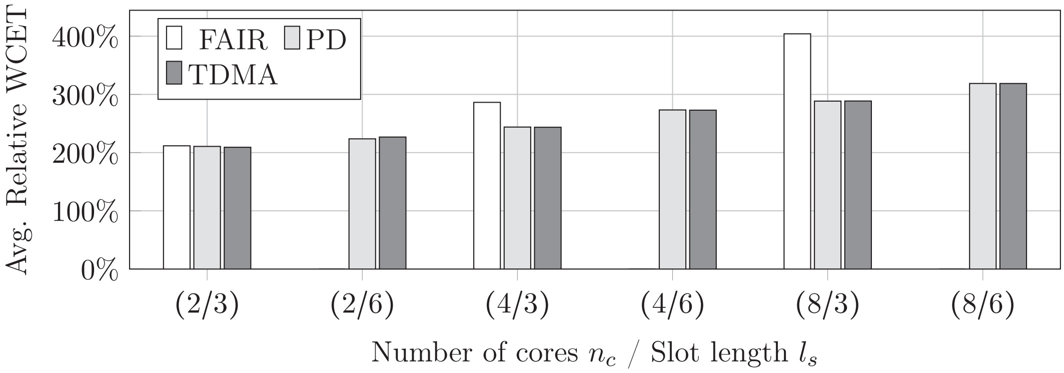


Figure 5.15: Average relative WCET for different arbitration types with the basic, TA-prone TDMA analysis.

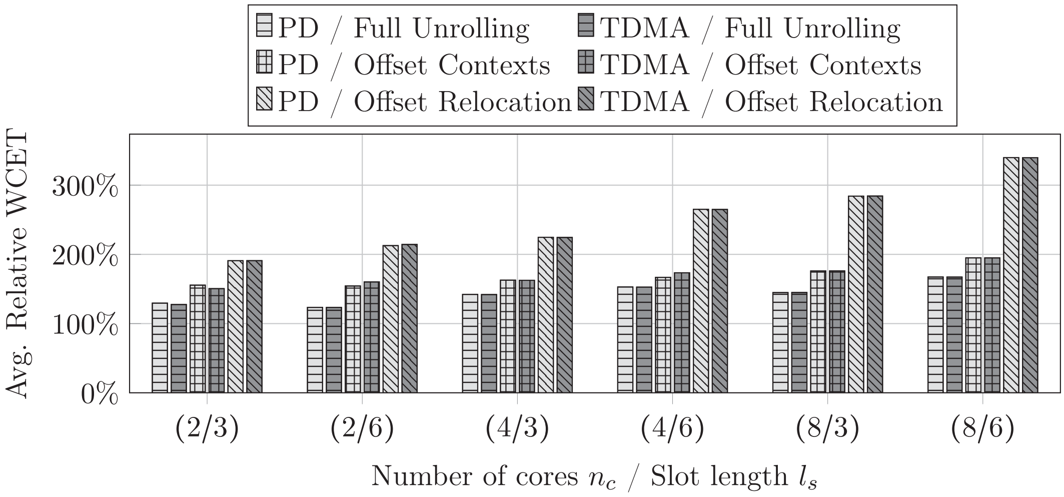


Figure 5.16: Average relative WCET for advanced TDMA analysis techniques.

For TDMA, between 10 and 20% of the total overestimation could be avoided in comparison to the worst-case behavior, but the WCET reduction is not very strong up to here. This provided the initial motivation to devise the improved TDMA analysis techniques presented in Section 5.4.4, Section 5.4.5 and Section 5.4.6.

To see, how much we can improve on this baseline, we have repeated the PD and TDMA experiments with the full loop unrolling approach (Section 5.4.4), the offset context approach (Section 5.4.5) and the offset relocation approach (Section 5.4.6). In addition, the new experiments also exploit the fact, that our architecture is TA-free. The results are presented in Figure 5.16.

As visible, all three approaches are better than the basic TDMA offset analysis shown in Figure 5.15, except for the offset relocation which is worse than the basic analysis for $n_c = 8, l_s = 6$. The best approach in terms of WCET precision is the full unrolling context approach. However, other than suggested by the average

Table 5.2: Example for a case where offset contexts yield more precise results than full unrolling.

Arbiter	TA-free	Relocation	Offset Contexts	Unrolling	WCET	Duration
TDMA	no	no	no	0	19,099	2s
TDMA	no	no	no	∞	19,099	10s
TDMA	yes	no	no	∞	18,995	13s
TDMA	yes	no	yes	0	16,226	7s
TDMA	yes	yes	no	0	19,483	2s

numbers in Figure 5.16, the offset context approach is also better than the unrolling on some instances. If a loop only contains straight-line code, the full unrolling is the most precise method to analyze it, since it will exactly capture the behavior of each iteration. If, on the other hand, a loop contains many if-branches which all contribute different result offsets, then the offset context approach may be superior. The reason is, that each offset context provides a *reset point* for the offset information (cf. lines 11 and 12 in Algorithm 6), whereas the unrolling cannot provide such reset effects. Table 5.2 shows the results for a 4-core benchmark with the tasks `select`, `mult-4-4`, `lms-float` and `fir` for which the offset contexts performs best. The full unrolling and the transition towards the TA-free analysis only yields little WCET reduction. The relocation in this case leads to a WCET which is even higher than the result with the basic analysis (first line of Table 5.2), which shows that the penalty-based relocation must be applied very cautiously. In contrast, the offset context approach reduces the WCET significantly.

Finally, as also visible from Figure 5.16, the offset relocation approach performs still better than the basic one in all cases except (8/6), but compared to the other two approaches it shows inferior precision. One reason for this behavior is, that we perform the offset relocation *only* for those loops which *may* access the bus (cf. Section 5.4.6). Unfortunately, we frequently encounter situations, in which the value analysis cannot determine the target of a memory access, which is then classified as potentially accessing *any* memory address, including those covered by the shared bus. Therefore, these imprecise value results introduce false positives into our may-access-the-bus classification which leads to costly applications of the offset relocation in loops where we have no benefit from the relocation at all. Unfortunately, we have seen in Section 5.4.7 that the offsets at a loop head of any loop which does perform bus accesses *must* be relocated to ensure the termination of the analysis. Therefore, the only way to increase the precision of the offset relocation-based analysis is to further increase the precision of the value analysis or to find some other, better approximation of the may-access-the-bus classification.

In all these experiments, the PD results were virtually equal to those for TDMA which may be due to our choice of the PD configuration and the benchmarks used.

Table 5.3: Average analysis time per benchmark for a timing-anomaly-free analysis run.

Platform	Arbiter	Relocation	Offset Contexts	Unrolling	∅ Duration
(2/3)	FAIR	no	no	0	2s
(2/3)	TDMA	no	no	∞	514s
(2/3)	TDMA	no	no	0	3s
(2/3)	TDMA	no	yes	0	30s
(2/3)	TDMA	yes	no	0	7s
(4/3)	FAIR	no	no	0	13s
(4/3)	TDMA	no	no	∞	569s
(4/3)	TDMA	no	no	0	20s
(4/3)	TDMA	no	yes	0	238s
(4/3)	TDMA	yes	no	0	12s
(8/3)	FAIR	no	no	0	21s
(8/3)	TDMA	no	no	∞	695s
(8/3)	TDMA	no	no	0	37s
(8/3)	TDMA	no	yes	0	398s
(8/3)	TDMA	yes	no	0	27s

In [KHM+13], more varied results for PD were found, and also one additional PD configuration was evaluated.

From the average analysis durations, shown in Table 5.3, we can see that the relocation approach is not dominated by the others since it has an analysis runtime which is comparable to the basic analysis' runtime. Time results for PD are not shown, since they are virtually identical to those of TDMA. The second largest runtime is observed with the offset context approach. The number of offset contexts of course directly influences the analysis duration. For each two-fold increase in schedule length, this number grows by a factor of 2^D where D is the deepest loop nesting depth in the program. Therefore, the super-linear analysis time growth observed in Table 5.3 should not be surprising.

As mentioned, the full unrolling approach ran out of time or memory in 69.1% of the experiments, whereas the offset context approach exceeded the maximum memory capacity in only 38.9% of the cases. The naive approach and the relocation had this problem in only 3.6% of the experiments. These cases where the time bound was reached *are* included in Table 5.3. Therefore, the runtime values for both the unrolling and the offset contexts can only be seen as a lower bound on the real runtime. Especially in the case of the unrolling, it can be expected that the real runtime if the process is given infinite amounts of time, is far higher.

The results shown in Table 5.3 were collected for a slot size of 3 cycles. Surprisingly, the runtime of the offset relocation analysis was only 2.7% higher for a slot size of 6, whereas the runtime for the naive analysis and the offset context analysis increased by 17% and 169% on average for slot size 6. In all cases, the fully unrolling approach was still the slowest one. The analysis time approximately doubles on average in the TA-prone case.

5.5 Unified WCET Analysis for Complex Multi-Cores

Section 5.4 introduced the concept of analyzing each task in isolation for the purpose of higher analysis speed and independent timing certification of tasks. Though these are strong arguments, we have also seen that an isolated per-task analysis can only be precise in the case of time-triggered arbitration. In particular, the important class of state-permeable arbitration policies like FAIR and PRIO (cf. Definition 20 and Table 5.1) cannot be analyzed by a partitioned analysis as presented in Section 5.4. Almost the same applies to shared caches, where only coarse-grained approximations are possible (cf. Section 5.4.1).

Therefore, this section will present a different approach to the microarchitectural analysis of multi-core systems which explicitly uses and maintains information about concurrently ongoing events. This will allow us to achieve more precise WCET estimations for FAIR arbitration and to analyze PRIO arbitration for the first time. Obviously, it also implies that the analysis effort will rise, because we need to track every relevant parallel execution scenario. The resulting analysis framework is depicted in Figure 5.17. In contrast to the partitioned framework from Figure 5.1, the microarchitectural analysis explicitly knows all tasks in the system and analyzes them together as opposed to each task in isolation. In the current state, this combined analysis only extends to the microarchitectural stage, after which the context block durations $\omega(v)$ are extracted from the combined analysis. The path analysis then works on each task CFG in isolation, as shown in Section 4.4. However, if the tasks have synchronization statements, the timing of these can only be captured by a combined path analysis. Since we focus on the microarchitectural analysis here, we ignore the synchronization aspect here, but it would be a good starting point for future work to also integrate a combined synchronization-aware path analysis.

The content of this section was previously published in [KM14].

5.5.1 Related Work

The author is not aware of any previous work that deals with a non-partitioned multi-core WCET analysis. Instead, previous work on multi-core WCET analysis, as summarized in Section 5.3.1, exclusively uses the per-task analysis approach with the mentioned drawbacks.

However, the static analysis of parallel software has a long tradition in the area of formal verification. In the following we present a short extract from the existing body of literature on this topic.

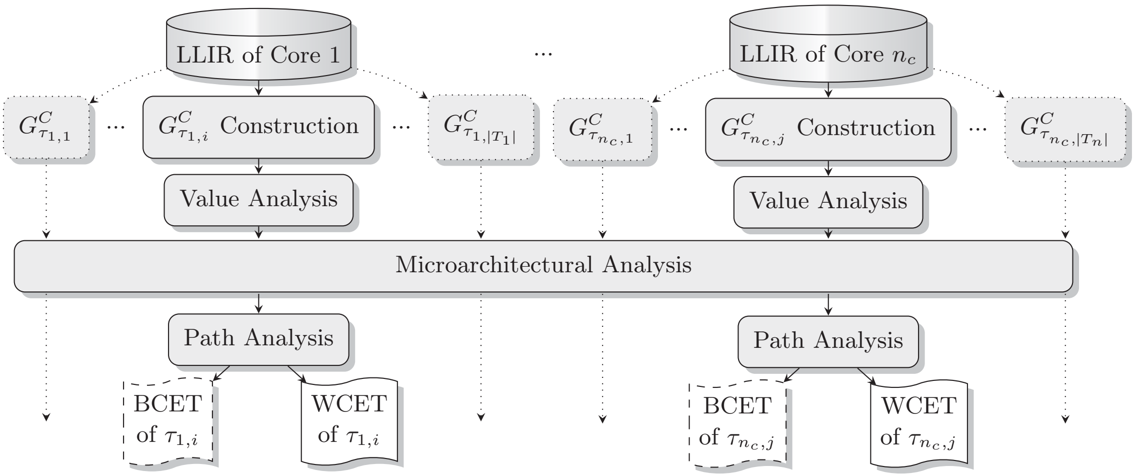


Figure 5.17: Structure of the unified multi-core WCET analysis.

Analysis of synchronization properties

A fixed-point analysis of *Communicating Sequential Processes* (CSP) was first established by Cousot and Cousot in [CC80]. It can prove properties like the absence of deadlocks or program termination for a subset of CSP programs. These concepts were later generalized to discrete state transition systems in [CC84] by the same authors.

Static analysis of the synchronization structure of concurrent programs was first considered by Taylor [Tay83a]. He presents an algorithm which can approximate which parts of a synchronized program may run in parallel to each other, similar to what we have done in Algorithm 5. The underlying problem is called the *May-Happen-In-Parallel* (MHP) problem. The difference is, that he is using synchronization to bound the possible parallel execution scenarios, whereas we will use timing information to do the same. Taylor also introduced the notion of a *Parallel Execution Graph* (PEG), which is also used in the analysis presented in this section, though at a far more fine-grained level. The MHP problem has been the topic of many works, including an approximation of the MHP relation for Java programs [NAC99] which runs in cubic time, and MHP computations using Java programs with barriers [KY06].

The exact solution to the MHP problem has been shown to be NP-hard [Tay83b] even when timing is not taken into account. Even worse, Ramalingam has shown in [Ram00] that any static analysis which is exact with respect to the call behavior and synchronization behavior of the tasks is undecidable.

In the seminal work of Valmari [Val89], an efficient algorithm is developed to explore all relevant interleavings of transitions in a generalized petri net. It is shown that many interleavings can be discarded in the analysis, since they lead to the same terminal states. Therefore, it is sufficient to consider a more restricted *stubborn set* of transitions. Unfortunately, this method is by construction only applicable in analyses which examine properties based on the reachability of termination states like, e.g., detection of deadlocks.

Chow et al. [CI92] combine stubborn sets, virtual coarsening (i.e., program slicing) and abstract interpretation to form a framework for the analysis of C-like shared memory parallel programs. Unfortunately, only concepts are given, without implementation or feasibility studies.

Analysis of parallel program semantics

Whereas the previous analyses approximate synchronization-related questions like MHP or deadlock-freedom, there are also approaches which try to create classical data-flow analyses for parallel systems, e.g., parallel liveness analysis or parallel value analysis. Again, we can only present fragments of this field of research due to the amount of literature published on it. To precisely capture the effects of parallel programs, like in the work of Taylor [Tay83a], a *Parallel Execution Graph* (PEG) is needed. As an example, this is demonstrated for the case of *Message Passing Interface* (MPI)-Analysis in [GKS+11].

Since the construction of a PEG which captures all relevant concurrent interleavings is computationally expensive, there are also attempts to use a summary-based technique similar to what we have seen in the handling of shared caches in Section 5.4.1. The basic idea is always to first compute a single-core result and then "patch" the results to account for possible parallel modifications.

For the *reaching definitions* bit-vector problem on parallel programs, this idea was first applied by Grunwald and Srinivasan [GS93]. *Bit-vector problems* are a subclass of distributive data-flow frameworks (cf. Definition 5), which are particularly easy to solve. The extension to arbitrary bit-vector problems was completed by Knoop, Steffen and Vollmer [Vol95; KSV96]. They show, that the MOP and MFP solutions (cf. Section 2.1) are identical for these types of problems when using the summary-based approach, i.e., that this approach produces precise solutions for bit-vector data-flow problems on parallel programs. Unfortunately, the microarchitectural analysis is *not* a bit-vector problem.

A similar approach is the use of a traditional sequential analysis whose results are filtered or widened by a *race detection engine* as presented by Chugh et al. [CVJ+08]. Other than in [KSV96], no guarantees on the quality of the results are given due to the usage of non-bit-vector domains.

The task scheduler was neglected in all of the works presented up to here. To overcome this, Mine [Min12] provides a scheduler-aware, path-aware abstract interpretation semantics of real-time C (without recursion and dynamic memory allocation). The analysis can detect arithmetic overflows, null pointer errors and similar run-time errors and is tested on a huge avionics benchmark. Similar to him, we also take the scheduler into account and have an explicitly parallel semantics. In addition, we use the generated WCET values to further prune the search space.

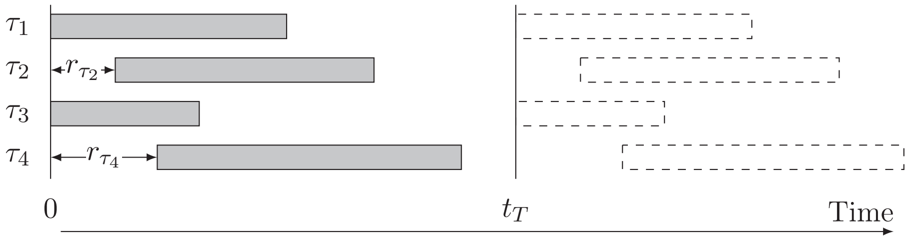


Figure 5.18: The basic task scheduling model for the unified multi-core analysis.

Finally, a recent publication from Mittermayr and Blieberger [MB12] examines the computation of feasible synchronization-aware parallel interleavings. Their approach focuses on path analysis and is thus complementary to the one presented in this section.

5.5.2 Task Model

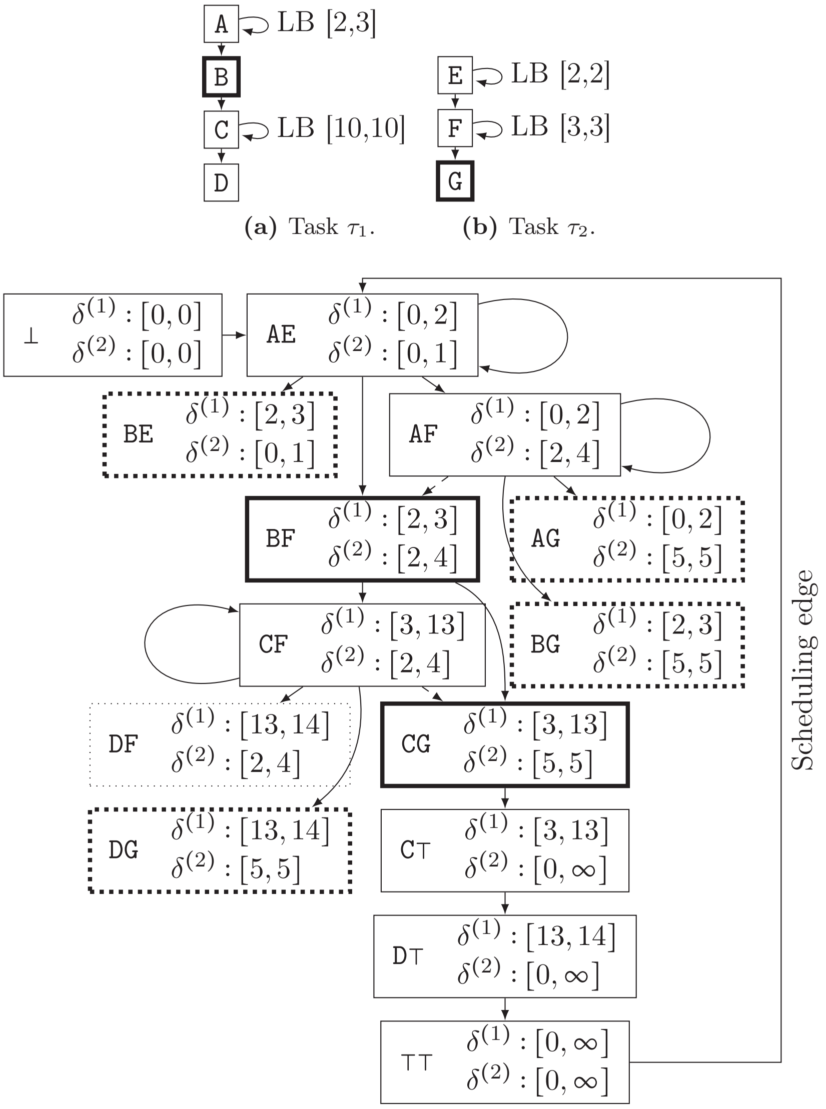
To be able to bound the number of possible parallel execution scenarios, each task $\tau \in T_c$ (i.e., mapped to core c) must be strictly periodic with a period t_τ as exemplified in Figure 5.18. Periods are given as a time length according to Definition 21. This is not a major restriction, since tasks in hard real-time systems are often strictly periodic as, e.g., in the OSEK Time-Triggered Operating System [OSE01].

In the following sections, we will need a common reference point in time for all running tasks. Therefore, we first require that all tasks τ have the same period $t_\tau = t_T$ and that each task is executed non-preemptively on a separate core. However, each task τ may have a different release time r_τ within the common period. We will discuss how to lift the restriction to a common period in Section 5.5.8.

5.5.3 Motivating Example

Before starting with the formal specifications, we briefly sketch the intuition behind the analysis procedure. Our goal will be to efficiently explore all feasible interleavings of multiple tasks running in parallel. As an example, consider the execution of the tasks τ_1 and τ_2 as given by the context graphs in Figure 5.19a and Figure 5.19b under the assumption that both tasks start concurrently at time 0. For this assumption, we can find all valid parallel execution scenarios from the *Parallel Execution Graph* (PEG) shown in Figure 5.19c.

The construction of this graph starts with node $\perp$ which represents the state of the system before the execution begins. Then, we add edges from $\perp$ to the nodes corresponding to the possible starting points for a parallel execution, in this case only the node `AE` (the δ-values will be explained below). From these start nodes, we iteratively simulate cycle steps of the system. To keep our example PEG from Figure 5.19c sufficiently small, we assume that every context block will take one cycle


A
LB [2,3]
B
C
LB [10,10]
D
E
LB [2,2]
F
LB [3,3]
G
⊥ δ(1) : [0,0] δ(2) : [0,0]
AE δ(1) : [0,2] δ(2) : [0,1]
BE δ(1) : [2,3] δ(2) : [0,1]
AF δ(1) : [0,2] δ(2) : [2,4]
BF δ(1) : [2,3] δ(2) : [2,4]
AG δ(1) : [0,2] δ(2) : [5,5]
CF δ(1) : [3,13] δ(2) : [2,4]
BG δ(1) : [2,3] δ(2) : [5,5]
DF δ(1) : [13,14] δ(2) : [2,4]
CG δ(1) : [3,13] δ(2) : [5,5]
DG δ(1) : [13,14] δ(2) : [5,5]
C⊤ δ(1) : [3,13] δ(2) : [0,∞]
D⊤ δ(1) : [13,14] δ(2) : [0,∞]
⊤⊤ δ(1) : [0,∞] δ(2) : [0,∞]
Scheduling edge

(a) Task τ_1. **(b)** Task τ_2.

(c) The final parallel execution graph.

Figure 5.19: Parallel execution graph creation example for two tasks τ_1 and τ_2, starting synchronously at time 0.

to complete. We also assume that blocks B and G hold a shared resource access, i.e., if both blocks try to execute concurrently, one of them has to wait for one cycle until the shared resource is free again. The aim of our analysis is to determine the block durations $\omega(v)$ for each block (cf. Definition 19). In our simplified example, we already know that $\omega(v) = 1$ for $v \in \{\mathtt{A}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}\}$, whereas $\omega(v) \in \{[1,1],[1,2],[2,2]\}$ for $v \in \{\mathtt{B}, \mathtt{G}\}$.

Therefore, our initial block AE is terminated after the first cycle and the execution must continue in one of the nodes AE, BE, BF and AF. To generate these successors, we simply follow all combinations of successor blocks in the tasks' context graphs. The loop bounds are not used here. If we continued the graph construction in this manner, we would end up with a full product graph of the context graphs. However, we will see in the following that a full product graph is not always needed. When every core has reached the end of its task, indicated by the "⊤" sign in Figure 5.19c, we add a back-edge from ⊤⊤ to AE to account for the repeated execution of the tasks in the cyclic schedule. The final PEG contains every possible parallel execution scenario for each context block in each task. Thus, we can derive $\omega(v)$ of a context block v from the PEG by measuring the length of all traces in the PEG which model the execution of v. The length of each PEG block is given by the number of cycle steps that were performed to analyze this block.

As visible, the PEG in Figure 5.19c is *not* a full product graph of the graphs from Figure 5.19a and Figure 5.19b. The construction of the graph has been stopped at nodes BE, AG, BG, DF and DG. To explain why this was done, and why it is correct, we need the δ-values and the loop bounds. We define $\delta^{(i)}$ as an interval containing all points in time, measured from the beginning of the common period p_T, at which a node may be entered on core i. Initially, we set $\delta^{(1)} = \delta^{(2)} = [0, 0]$ for node AE, since core 1 (2) enters node A (E) at time 0. From here on, every time we visit a PEG node v in the analysis, we recompute its δ intervals with the help of the path analysis which computes the length of the shortest and longest paths to the context blocks in v.

As an example, when we visit node AE the second time, we have already seen that both blocks A and E complete within one cycle. Therefore, since A can be executed at most three times and E at most two times (see Figure 5.19a and Figure 5.19b), the path analysis can infer that any execution of block A must begin in the time frame $\delta^{(1)} = [0, 2]$ and similarly any execution of block E must begin within $\delta^{(2)} = [0, 1]$. Thus, the path analysis always operates only on the context graphs of the tasks, *not* on the PEG. The PEG is only used to compute the $\omega(v)$ values that are used by the path analysis.

The path analysis for node BE yields $\delta^{(1)} = [2, 3]$ (due to the loop at A which must complete before B) and $\delta^{(2)} = [0, 1]$. Here, we can see the application of the computed δ-values: We can exclude this node from the PEG and thus from the analysis. Using the δ-values we know that blocks B and E cannot be executed concurrently because their execution time windows do not overlap. This condition is called the *Block Exclusion Criterion* (BEC) and all blocks for which it holds can be removed from the PEG. In Figure 5.19c, these blocks are marked by a dotted border. Therefore, we can infer from the final PEG that BG is an invalid node, i.e., that $\omega(\texttt{B}) = \omega(\texttt{G}) = 1$.

Following this basic idea, the next sections will show how to generate a PEG and how to use timing information to prune parts of it for increased WCET precision and analysis speed. When lifting the simplifications made in this example, there are many points that we will have to clarify, some of the most important being how to deal with the case when blocks do not terminate concurrently and how to incorporate system state information into this analysis scheme. We will cover these aspects in the next sections.

5.5.4 Prerequisites

To precisely define our analysis procedure, we will need some terminology which is introduced in the following.

Definition 25. *Given a task $\tau \in T_c$, a* Task Execution Position *(TEP) $\psi_\tau \in \Psi_\tau$ is a tuple (v, i, c, d), where $v \in V_\tau^C$ is a context block, $i \in v$ is an instruction within that basic block and c is the number of cycles that were already spent on the processing of this instruction. Finally, d is the number of cycles that the task must wait until its execution will begin. The set of all TEPs of task τ is called Ψ_τ.*

Definition 26. *A* System Execution Position *(SEP) ψ on n cores is an n-tuple with $\psi \in \Psi = \bigtimes_{i=1}^{n} \Psi_{\tau_i} \cup \{\top\}$, τ_i being a task mapped to core i. The special token $\top$ indicates that the respective core is currently running idle. The set of all SEPs is called Ψ.*

The motivation for this definition is, that other than in our introductory example from Figure 5.19, real basic blocks will contain more than one instruction[3] each of which may take multiple cycles to complete. Still, we need to be able to split the execution of each basic block into chunks which may be as small as a single CPU cycle, as we will see in the following. We will use SEPs to specify the point at which the execution is resumed in a PEG block. Therefore SEPs correspond to the block labels from Figure 5.19c (e.g., `AE`, `BE`, `AF`, etc).

Definition 27. Abstract Parallel System State*s (APSSs) $\Sigma \in \mathbb{X} = 2^{(Q_P)^{n_c} \times Q_E}$ are a generalization of abstract microarchitectural states $m \in \mathbb{M} = 2^{Q_P \times Q_E}$ (cf. Definition 17) which may contain more than one pipeline state.*

The environment state Q_E contains both the state of the memory hierarchy elements that are private to the cores as well as the state of shared memory hierarchy elements. We give more detail on how to form proper APSSs in Section 5.5.6.

Similarly to Definition 16, we can define the parallel execution of a set of programs based on concrete parallel system states and generalize this execution to abstract states along the lines of Equation 4.17. We require the existence of a monotonic *cycle step function* $\xi_\mathbb{X} : \mathbb{X} \times \Psi \times 2^{\{1,\ldots,n_c\}} \to (\{0,1\}^{n_c} \times \mathbb{X})$. The invocation of $\xi_\mathbb{X}(\Sigma, \psi, \alpha)$ must simulate all possible state transfers that may happen when a single clock cycle is executed at position ψ in system state Σ. However, only the cores in the set $\alpha \subseteq \{1, \ldots, |T|\}$ may perform a cycle step, to be able to account for different release times. For any *instruction completion vector* $c \in \{0,1\}^n$ which may occur in this cycle, it must specify the result state, where c defines for each core,

[3] In the example, we have not even differentiated between basic blocks and instructions.

whether it has retired its current instruction (1) or not (0) in analogy to Definition 16. The "current instruction" is always given by the "program counter" register value.

Definition 28. *A* Parallel Execution Graph $G_P = (V_P, E_P)$ *is a directed graph with node set* $V_P \subseteq \Psi \cup \{\bot\}$ *and edge set* $E_P \subseteq V_P \times V_P$*.* $\bot$ *is a special PEG node which is exclusively used to model the situation that the execution of the parallel system has not yet started. For any PEG, we define*

- *a* block time window *mapping* $\delta : V_P \to (\mathbb{I}_{time})^{n_c}$*,*
- *an* edge state mapping $\lambda : E_P \to \mathbb{X}$*, and*
- *a* block length mapping $\omega_P : V_P \to \mathbb{N}$.

$\mathbb{I}_{time} = \{[x,y] \subset 2^{\mathbb{N}}_0 | x \leq y\}$ *is the set of all discrete execution time intervals, measured in cycles from the last point where all cores were synchronized.*

The edge set $E_P = E_P^{cf} \cup E_P^{sched}$ *is partitioned into a set of control-flow edges* E_P^{cf} *and a set of scheduling edges* E_P^{sched}*.*

The time window function will be used to rule out infeasible SEPs as indicated in Figure 5.19c. The edge state function is employed for the propagation of the possible hardware states from one PEG node to the other and the block length function specifies how many cycles were spent on the execution of a PEG node. The three mappings are not defined a priori. They will be computed by the algorithms presented in the following. The partitioning of the edges is needed, since also in our example from Figure 5.19c we have two types of edges. The edge $(\texttt{TT}, \texttt{AE})$ is a scheduling edge which means that an unspecified amount of time may pass until the transition from `TT` to `AE` is completed. All other edges are control-flow edges which have the semantics of an *immediate* transition, i.e., no time passes when taking a control-flow edge.

Note, that the PEG block runtime ω_P must not be confused with the context block runtime ω as defined in Definition 19. To disambiguate the two symbols, we refer to the latter as ω_C in the following.

In Section 5.5.5, we will first examine how to construct a PEG for a given task set and with a given abstract cycle step function $\xi_{\mathbb{X}}$. We also show how to extract the ω_C-values for each context block from the final PEG (cf. Definition 19). In Section 5.5.6, we will then take a brief look at how to properly model the abstract parallel system states $\mathbb{X}$ and their cycle step $\xi_{\mathbb{X}}$.

5.5.5 Parallel Execution Graph Construction

The outline of the main analysis is shown in Algorithm 7. Here and in the following, we use $()^{(i)}$ to access the i-th element of a tuple.

It starts with an initialization of the initial context block runtimes ω_C in line 2 and of the work-list Q in line 3. According to the system schedule, the initial SEP consists of the begin of the start block of each task ($v_{\tau_i}^{\text{start}}$) with a delay of r_i cycles.

Algorithm 7 PEG-driven parallelism analysis.

```
 1: function ParallelismAnalysis($\Sigma_{\text{start}}$, $G^C_{\tau_1}$, ..., $G^C_{\tau_n}$)
 2:     $\forall \tau : \forall v \in V^C_\tau : \omega_C(v) = \emptyset$          ▷ Initialize all context block runtimes to $\emptyset$
 3:     $Q \leftarrow (v^{\text{start}}_{\tau_1}, 0, 0, r_1) \times \cdots \times (v^{\text{start}}_{\tau_n}, 0, 0, r_n)$          ▷ Initialize work queue
 4:     $G_P \leftarrow (Q \cup \{\bot\}, \emptyset)$
 5:     $\delta(\bot) \leftarrow [0,0]^{n_c}, \omega_P(\bot) \leftarrow \infty$
 6:     $\forall \psi \in Q : \delta(\psi) \leftarrow \{\emptyset\}^{n_c}, \lambda((\bot, \psi)) \leftarrow \Sigma_{\text{start}}, \omega_P(\psi) \leftarrow \infty$
 7:     while $Q \neq \emptyset$ do                                      ▷ Main loop
 8:         $\psi$ = PopFront($Q$)                                        ▷ Analyze next block
 9:         $\omega_C \leftarrow$ GatherNewBBTraces($\psi, G_P, \omega_P, \omega_C$)          ▷ Update $\omega_C$.
10:         for $i \in \{1, \dots, n_c\}$ do                   ▷ Update $\delta$-window for all cores
11:             $\delta(\psi)^{(i)} \leftarrow \bigcup_{(\psi', \psi) \in E^{\text{cf}}_P} \delta(\psi')^{(i)} + \omega_P(\psi')$
12:             if IsLoopHeadOrExit($\psi^{(i)}$) then
13:                 $\delta(\psi)^{(i)} = r_i +$ PathAnalysis($\psi^{(i)}, G^C_{\tau_i}, \omega_C$)
14:         if $\forall_{i \in \{1,\dots,n_c\}} \delta(\psi)^{(i)} \neq \emptyset \wedge \bigcap_{i=1}^{n_c} \delta(\psi)^{(i)} = \emptyset$ then       ▷ If BEC holds ...
15:             continue                                  ▷ ... skip the current block $\psi$ ...
16:         else                                                   ▷ ... else analyze it.
17:             $\lambda_{\text{prev}} \leftarrow \lambda$, $G_{P,\text{prev}} \leftarrow G_P$
18:             $(G_P, \lambda, \omega_P) \leftarrow$ AnalyzeBlock($\psi, G_P, \lambda, \omega_P$)
19:             if $\lambda_{\text{prev}} \neq \lambda \vee G_{P,\text{prev}} \neq G_P$ then  ▷ If graph or states were altered ...
20:                 $\forall (\psi, \psi') \in E_P$ : PushBack($Q, \psi'$)   ▷ ... propagate the changes.
21:                 if $E_{P,\text{prev}} \neq E_P$ then                    ▷ If edges were added ...
22:                     $\forall \psi \rightsquigarrow \psi'$ : PushBack($Q, \psi'$)          ▷ ... propagate $\delta$-changes
23:     return $\omega_C$
```

As in the following, every new SEP get an initially empty execution time window δ and an infinite runtime ω_P. We also create a virtual edge $(\bot, \psi)$ pointing to the initial SEP, which is assigned the initial APSS $\Sigma_{\text{start}} \in \mathbb{X}$. The start block $\bot$ itself has a runtime of zero cycles and executes in the start window $[0,0]$ to mark that the schedule starts here. Then, we process items from the queue Q until it gets empty (line 7). In the main loop, we extract the first block ψ from the queue. After the new SEP ψ is extracted from Q, we first check whether ψ models the end of a context block v on any core in the call to GatherNewBBTraces in line 9. For any such context block v, its runtime $\omega_C(v)$ is updated in GatherNewBBTraces as shown in Algorithm 8.

In line 11 of Algorithm 7, we infer the block time window for all task positions $\psi^{(i)} \in \psi$ from the windows and runtimes of its control-flow predecessors. If ψ is part of a sequential block chain, the δ-update in line 11 is sufficient. On the other hand, if ψ is a loop head (like `A` in Figure 5.19a) or a loop exit[4] (like `B` in Figure 5.19b), then we have to take the loop bounds into account to determine the block time window,

[4] A successor of a loop head which is not part of the loop.

Algorithm 8 Update of basic block runtimes.

```
function GatherNewBBTraces($\psi, G_P, \omega_P, \omega_C$)
    for $i \in \{1, \dots, n_c\}, (\psi', \psi) \in E_P$ do
        if $\psi^{(i)(1)} \neq \psi'^{(i)(1)}$ then          ▷ If $\psi$ is context block start on core $k$, ...
            $v_{\text{pred}} = \psi'^{(i)(1)}$     ▷ ... collect the length of all paths to starts of $v_{\text{pred}}$.
            $\omega_C(v_{\text{pred}}) \leftarrow \omega_C(v_{\text{pred}}) \cup$ TraceToStarts($v_{\text{pred}}, \psi', i, G_P, \omega_P$)
    return $\omega_C$
function TraceToStarts($v_\tau, \psi, i, G_P, \omega_P$)
    if $\psi^{(i)} = (v_\tau, i_0, 0, 0)$ then                      ▷ If $\psi^{(i)}$ is a begin of $v_\tau$, ...
        return $\omega_P(\psi)$                                          ▷ ... finish this trace.
    else                                              ▷ Else continue with the recursion.
        return $\bigcup_{(\psi', \psi) \in E_P} \{\omega_P(\psi) +$ TraceToStarts($v_\tau, \psi', i, G_P, \omega_P$)$\}$
```

like we have done in the computation of $\delta^{(1)}$ in, e.g., AE and BE in Figure 5.19c. This is done in line 13, where the existing path analysis of our framework is used to compute the shortest and the longest path from $v_{\tau_i}^{\text{start}}$ to $\psi^{(i)}$. The path analysis may fail, because not all loop blocks were yet analyzed. In this case, an empty set is returned, which also sets $\delta(\psi)^{(i)} = \varnothing$.

The δ values are used in line 14, where we try to apply the *block exclusion criterion* by intersecting all block time windows. However, this test can only be applied if the time windows for each task could already be determined, i.e., if they are not empty. If the *intersection* is empty, ψ cannot be reached from its current predecessors and we skip its analysis in line 15. This is exactly what we have done with BE in Figure 5.19c. Still, we may need to analyze ψ in the future when it becomes accessible via new edges. Then, we will re-check whether our exclusion criterion still holds. Thus, this skipping is effectively either postponing or avoiding the graph growth at ψ.

If the exclusion criterion does not hold (line 16), we analyze the *parallel execution block* (PEB) beginning at node ψ (line 18). This analysis will determine a block runtime $\omega_P(\psi)$, an output APPS for all out-edges of ψ and will possibly alter G_P. If the output states or the graph are changed, we push the successors of ψ into the work-list at line 20. By doing this, all changes to the block time windows δ, edge states λ and block runtimes ω_P will be propagated through the graph. Finally, if we have added edges to the PEG, we also push all blocks ψ' which are reachable from ψ into Q (line 22), to ensure that a new attempt to compute $\delta(\psi')$ is started, if ψ' is a loop head or exit. The algorithm terminates when no more edges are added and all edge states have converged.

All in all, Algorithm 7 is a standard data-flow analysis work-list algorithm, with the difference that we are dynamically expanding (line 18) the underlying graph G_P. However, we will show in the proof of correctness, that the convergence of the PEG is still guaranteed. When ParallelismAnalysis has finished, all reachable blocks of all tasks will have been visited in one or more parallel execution blocks, i.e., ω_C will contain the possible execution time for each reachable context block of each task.

For the path analysis $\textsc{PathAnalysis}(\psi^{(i)}, G^C_{\tau_i}, G_P, \omega_C)$ we are using an adapted IPET, as presented in Section 4.4, with some modifications to account for the facts that

1. we do not yet know the final context block durations $\omega_C(v)$, and
2. we are not searching for the longest (shortest) path through the *whole* program but only for the longest (shortest) path to the current context block $\psi^{(i)}$.

To deal with problem 1, PathAnalysis simply uses the preliminary values of ω_C as determined by Algorithm 8. If any block $v \in G^C_{\tau_i}$ with $v \rightsquigarrow \psi^{(i)}$ is not yet covered in G_P, then the initial value $\omega_C(u) = \varnothing$ is still present. The path analysis will then also return $\varnothing$ for the path length to $\psi^{(i)}$.

Problem 2 requires some minor changes to the IPET, too. In the original IPET, we added edges from all terminal blocks $v \in \delta^A_\top(\nu_\top(f^\perp_\tau))$ to the virtual sink v^-. Since we want to determine the longest (shortest) path to $\psi^{(i)}$ now, we no longer add these edges. Instead we add edges to v^- from all $v \in \delta^-(\psi^{(i)})$, i.e., all predecessors of the current task execution position. We call all of these predecessors *explicit sinks* of the resulting IPET. Each of the explicit sinks can be used to terminate the flow through the IPET by directing its flow towards v^-. Thus, each of the explicit sinks can also be used to exit from a function call sequence or one or multiple loops in which $\psi^{(i)}$ is nested. Therefore, Equation 4.45 must be adapted to account for the possibility that affected functions are left via the explicit sink edge and not via the return edge. Similarly, the lower loop bound in Equation 4.47 must be dropped for all loops which are not post-dominated by *all* explicit sinks, since the minimum iteration count only applies if the loop is completely executed before $\psi^{(i)}$ is reached. The same problems arises for flow restrictions, but unfortunately we cannot determine whether a given flow restriction stays valid in the longest-path-to-$\psi^{(i)}$ problem. Therefore, our modified path analysis rejects tasks with flow restrictions. These are not needed in most real-time benchmarks, and it is only a restriction of our *path* analysis, not of the parallelism analysis itself.

The repeated building and solving of the modified IPETs is not an ideal solution, since basically we are just re-computing shortest and longest paths to all nodes of the context graph under changing node weights. Advanced single-source all-sinks analyses [KFM13] would be much better suited to solve this problem more efficiently. Since the IPET was already available, we nevertheless used it as a preliminary solution to avoid the implementation of a new type of path analysis.

Single-Block Analysis

To complete the view on the analysis, Algorithm 9 shows the function AnalyzeBlock which is invoked in Algorithm 7. First, the incoming APSSs are joined in line 2. The current system execution position ψ_{run} is initialized to ψ_{in} (remember that $V_P \subseteq \Psi$) and the block duration $\omega_P(\psi_{\text{in}})$ is set to zero. Then, we simulate the effect of successive system cycle steps on ψ_{run} and Σ_{run}, until on any core, either

a) the end of a basic block is reached or
b) the successor SEP is ambiguous.

The latter happens, when it is uncertain in APSS Σ_{run} whether the current instruction of at least one core will complete or not. In this case, we track all completion combinations in separate successor blocks.

The first step in each cycle is to invoke the APSS cycle step function $\xi_{\mathbb{X}}$, which is done in line 5, but only for those cores with zero delay cycles (set α). The APSS cycle step function $\xi_{\mathbb{X}}$ returns a mapping $\kappa \subseteq \{0,1\}^{n_c} \times \mathbb{X}$, i.e., it associates instruction completion vectors to successor APSSs. Line 6 checks the two block termination conditions *a)* and *b)* mentioned above. The helper function $\phi_c^\alpha : \Psi \to \Psi$ generates the successor SEP for a given SEP ψ, instruction completion vector c and active core set α. We define it as

$$\phi_c^\alpha(\psi) = \bigtimes_{\substack{k\in\{1,\dots,n_c\}:\\ \psi^{(k)}=(v,i,c,d)}} \begin{cases} (v,i,c,d-1) & \text{if } k \notin \alpha \\ (v,i,c+1,0) & \text{if } k \in \alpha, c^{(k)} = 0 \\ \phi'(v,i) & \text{if } k \in \alpha, c^{(k)} = 1 \end{cases} \tag{5.27}$$

where $\phi'(v,i)$ is the set of all task execution positions "after" instruction i, i.e., either

- the next instruction in v, if there is any, or
- the first instruction of all blocks w with $(v,w) \in E^C_{\tau_k}$, if such edges do exist, or
- the terminal symbol $\top$, if $\psi^{(k)} \neq \top$, or
- all initial PEG nodes if $\psi^{(k)} = \top$.

If neither a basic block end is reached, nor the successor SEP is ambiguous, we take over the results of the cycle step as our new working SEP ψ_{run} and APSS Σ_{run} in line 7 and increment the cycle counter for this block in line 8. Here, $\psi_{\text{run}}^{(i)(1)}$ is the basic block executed by core i, $\kappa^{(1)(1)}$ is the first instruction completion vector and $\kappa^{(1)(2)}$ is its associated successor APSS.

If the block end is detected, we terminate the current block as shown from line 9 on. It will be one invariant of our analysis that the length of a block can only stay the same or be reduced in successive analyses of the same block. Therefore, we only check in line 10, whether the block has been shortened. This may happen due to a newly joined-in APSS, that triggers an earlier ambiguous successor SEP. In this case, we remove all previous out-edges of the current block v (line 11). In any case, we add for each instruction completion vector c an out-edge to $\phi_c^\alpha(\Sigma_{\text{run}})$ which gets annotated with the respective out-state Σ_c (lines 13–16). In the end, the modified graph, edge states and block lengths are returned in line 18.

Algorithm 9 PEG block analysis.

```
function AnalyzeBlock($\psi_{\text{in}}, G_P, \lambda, \omega_P$)
    $\Sigma_{\text{run}} \leftarrow \bigsqcup_{\forall e=(\psi',\psi_{\text{in}})\in E_P} \lambda(e)$    ▷ Join incoming states
    $\psi_{\text{run}} \leftarrow \psi_{\text{in}}$, $\omega_{P,\text{prev}} \leftarrow \omega_P$, $\omega_P(\psi_{\text{in}}) \leftarrow 0$
    while true do
        $\kappa \leftarrow \xi_{\mathbb{X}}(\Sigma_{\text{run}}, \psi_{\text{run}}, \alpha = \{i|\psi_{\text{run}}^{(i)} = (\cdot,\cdot,\cdot,0)\})$    ▷ Simulate next cycle
        if $|\kappa| = 1 \wedge \nexists i : (\phi^{\alpha}_{\kappa^{(1)(1)}}(\psi_{\text{run}}))^{(i)(1)} \neq \psi_{\text{run}}^{(i)(1)}$ then    ▷ Split/Block end?
            $\Sigma_{\text{run}} \leftarrow \kappa^{(1)(2)}, \psi_{\text{run}} \leftarrow \phi^{\alpha}_{\kappa^{(1)(1)}}(\psi_{\text{run}})$    ▷ If not, prepare next cycle
            $\omega_P(\psi_{\text{in}}) \leftarrow \omega_P(\psi_{\text{in}}) + 1$
        else    ▷ Else terminate the current block
            if $\omega_P(\psi_{\text{in}}) < \omega_{P,\text{prev}}(\psi_{\text{in}})$ then    ▷ If the block shrank, ...
                $E_P \leftarrow E_P \setminus \{(\psi_{\text{in}}, \psi') \in E_P\}$    ▷ ... remove old edges
            for $(c \rightarrow \Sigma_c) \in \kappa$ do    ▷ Add new successors and out-states
                $V_P \leftarrow V_P \cup \{\psi_{\text{new}} = \phi^{\alpha}_c(\Sigma_{\text{run}})\}$
                $\delta(\psi_{\text{new}}) \leftarrow \{\varnothing\}^{n_c}, \omega_P(\psi_{\text{new}}) \leftarrow \infty$
                $E_P \leftarrow E_P \cup \{e_{\text{new}} = (\psi_{\text{in}}, \psi_{\text{new}})\}$
                $\lambda(e_{\text{new}}) \leftarrow \Sigma_c$
            break
    return $(G_P, \lambda, \omega_P)$    ▷ Return all modifications
```

With Algorithm 9 we completed the macroscopic side of the analysis. In the next section, we will examine the microscopic perspective, namely how to efficiently represent abstract parallel system states.

5.5.6 Parallel System States

As stated in Definition 27, an APSS is an abstraction of the microarchitectural state of a parallel machine. Therefore, we can use all of the models that were introduced for modeling pipelines (Section 4.3.1), caches (Section 4.3.2) and buses (Section 5.4.3). However, other than in the task-partitioned case (Section 5.4), we do not need to account for possible parallel interference by either using cache interference summaries or time-triggered bus schedules. Instead, we can now explicitly analyze concurrently ongoing actions, since the PEG encodes which program parts are currently executed in parallel. This allows us to also analyze the PRIO arbitration, which was not possible in the task-partitioned case, and to derive tighter WCET estimates for FAIR arbitration and shared caches.

The main difference compared to the single-core case is, that an APSS Σ contains an abstract state for n_c cores instead of for only one core. Therefore, each element $\sigma \in \Sigma$ is a tuple $\sigma \in (Q_P)^{n_c} \times Q_E$. The rationale behind Σ being a set of tuples is – as in the single-core case – that we need this mechanism to track different, alternative microarchitectural behaviors, e.g., in the case of an access to an unknown memory address.

In every cycle step, i.e., every invocation of $\xi_{\mathbb{X}}$, we perform the cycle step on each contained sub-state tuple $\sigma \in \Sigma$ with $\xi_\sigma : \sigma \times \psi \times 2^{\{1,\dots,n\}} \to (\{0,1\}^n \times \mathbb{X})$. ξ_σ then simply applies a cycle step to each contained pipeline and memory hierarchy element state. As we have already seen in the single-core case, the abstract finite state machines behind these states are non-deterministic, therefore multiple successors may be generated. This is reflected in the choice of $\mathbb{X}$ as the target domain of ξ_σ. Finally, each successor state σ' is labeled with the instruction completion information that was emitted by the pipeline models during the transition to σ'. These sub-results are collected for all $\sigma \in \Sigma$, and those successor states with the same completion vector are grouped together. This merged result is then returned by $\xi_{\mathbb{X}}$.

Up to here, we have not yet exploited the explicit encoding of parallelism in the PEG. This is only done if one or more cores access a shared resource. In this case, the requests arrive at the shared bus which is responsible for arbitrating them. The parallelism-aware abstract bus model is almost the same as the one shown in Figure 5.2, with the important difference that

- the input is not a single access r but a *set* of accesses R, and
- the transitions from "Arbitrate" to the "Blocked" and "Forward" states are only enabled for every $r \in arb(q^B, R)$.

The function $arb(q^B, R)$ must decide which of the access requests in R *may* be granted, based on the current bus state q^B. If the information in q^B is imprecise, $arb(q^B, R) = R$ is possible, and in the case of TDMA also $arb(q^B, R) = \varnothing$ is possible if all requests will definitely not be granted in the current cycle. If $|arb(q^B, R)| > 1$, then every grant possibility must be explored in a separate successor state.

Arbitration functions

For a TDMA bus, the state information q^B is a TDMA offset set as presented in Section 5.4.3. For TDMA, the *arb*-function is formed based on the *delay* as

$$arb_{\text{TDMA}}(q^B, R) = \{r \in R \mid 0 \in delay_{\text{TDMA}}(q^B, r)\} \tag{5.28}$$

A real advantage can be gained only for FAIR and PRIO arbitration. In contrast to TDMA, both are *work-conserving* arbitration methods, i.e., they do not generate idle cycles as long as there are pending requests. Therefore, the $delay_{\text{PRIO}}$ and $delay_{\text{FAIR}}$ are always equal to zero in the parallelism-aware analysis. The delay for those accesses which are blocked by another one is implicitly generated, since

- there will only be a single winner in an arbitration cycle, and
- contending requests which were not granted the bus must be re-issued until they are finally granted.

For FAIR, q^B over-approximates the cores which last accessed the bus, i.e.,

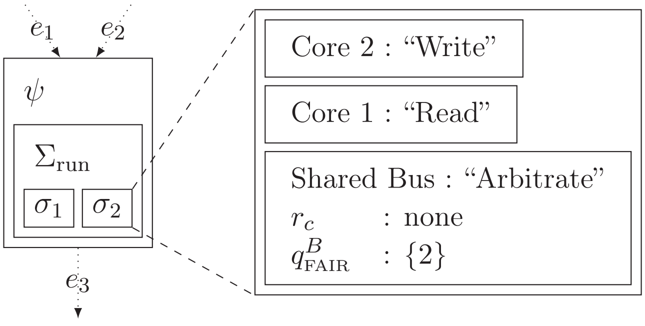


Figure 5.20: An example PEG block ψ with attached APSS Σ and details on the contained state tuple σ_2.

$$B_{\text{FAIR}} = 2^{\{1,\dots,n_c\}} \tag{5.29}$$

$$arb_{\text{FAIR}}(q^B, R) = \{r \in R \mid \exists c_p \in q^B : \forall r_2 \in R : c(r) - c_p \leq_{\text{mod}\, n_c} c(r_2) - c_p\} \tag{5.30}$$

In Equation 5.30, $c(r)$ yields the number of the core which issued request r. Thus, $arb_{\text{FAIR}}(q^B, R)$ returns all those requests which belong to a core that *may* be the next one in the round-robin schedule. The "last-access" information in q^B is only updated in the "Forward" states from Figure 5.2. For these states, we define

$$update_{\text{FAIR}}(q^B) = \{c(r_c)\} \tag{5.31}$$

where r_c is the currently granted access as defined in Figure 5.2.

The arbitration function for PRIO is even simpler, since we do not need to maintain any state information here. Instead, we can always perform the arbitration based on the priorities $p(r)$ of the requests r.

$$B_{\text{PRIO}} = \{-\} \tag{5.32}$$

$$arb_{\text{PRIO}}(-, R) = \{r \in R \mid \forall r_2 \in R : p(r) \geq p(r_2)\} \tag{5.33}$$

An example for these states is illustrated in Figure 5.20, where a PEG block ψ is shown with incoming and outgoing edges e_1, e_2 and e_3. The state Σ_{run} for this block (see Algorithm 9) holds two sub-states, of which σ_2 is presented in more detail. In this sub-state, the two cores in this example are currently performing a memory read and a memory write operation. The bus state q^B_{FAIR} shows that the last access has definitely been carried out by core 2. Assuming that both accesses hit the shared bus in this cycle, we know based on q^B that $arb_{\text{FAIR}}(\{2\}, \{r_1, r_2\}) = \{r_1\}$ with $c(r_1) = 1$ and $c(r_2) = 2$.

Since the PEG already carries the burden of constructing all possible interleaving scenarios, we can formulate the arbitration analysis in a rather simple manner here.

By construction, this has not been possible for the standard per-core WCET analysis approach.

Shared caches are immediately analyzable with this framework and the original cache domain from Section 4.3.2. Every possible order of accesses is implicitly explored by the PEG construction and the arbitration analysis. Therefore, no summary-based approach as in Section 5.4.1 is needed.

5.5.7 Correctness

In the following, we use G_P^i, λ^i, ω_P^i and δ^i to denote the PEG and the values of the three functions *after* i-th iteration of the main loop of Algorithm 7. Also, we denote the PEG node ψ that is analyzed in iteration i as ψ^i. The special iteration number 0 is used to denote the state *before* the first iteration of the main loop. We first show, that with rising analysis iteration count, for each PEG node the block runtime will only shrink, the incoming APSS will only get more imprecise and the execution time intervals for each task execution position will only become wider.

Lemma 2. *For any iteration j of the main loop of the parallelism analysis (line 7 in Algorithm 7), any iteration $i \leq j$ and any SEP $\psi \in G_P^i$, the following invariants hold:*

1. $\forall \psi' \in V_P^i : \psi' \rightsquigarrow_{G_P^i} \psi \implies \psi' \rightsquigarrow_{G_P^j} \psi$,
2. $\forall \psi' \in V_P^i : \lambda^i((\psi', \psi)) \sqsubseteq \lambda^j((\psi', \psi))$,
3. $\omega_P^i(\psi) \geq \omega_P^j(\psi)$, *and*
4. $\forall k \in \{1, \dots, n\} : \delta^i(\psi)^{(k)} \subseteq \delta^j(\psi)^{(k)}$.

Condition 1 codifies that any existing path in the PEG of iteration i is retained in the future versions of the PEG. Therefore, we call this condition the *structural condition* in the following. The conditions 2, 3, and 4 claim that the development of the three adjunct properties is monotonic and are therefore called *property conditions*.

Proof. We show the lemma by induction over j. For brevity of notation we denote the incoming APSS at block ψ as $\lambda_{\text{in}}(\psi)$, i.e.,

$$\lambda_{\text{in}}^j(\psi) = \bigsqcup_{(\psi', \psi) \in E_P^{j-1}} \lambda^{j-1}((\psi', \psi)) \tag{5.34}$$

Induction Base $(j = 1)$: In this case, we know the initial values of $\omega_P^0(\psi^1) = \infty$, $\lambda_{\text{in}}^0(\psi^1) = \Sigma_{\text{start}}$ and $\delta^0(\psi^1) = \{\varnothing\}^{n_c}$ from line 5 of Algorithm 7. Thus, condition 3 trivially holds, and $\lambda^0(\psi^1)$ will always contain the value Σ_{start} since it is contributed via the static edge $(\bot, \psi^1)$, which satisfies condition 2. Since E_P is initially empty, it can only grow in the first iteration, which implies condition 1. The $\delta^1(\psi^1)$-value is either $\varnothing$ or $[0, 0]$ depending on whether ψ^1 is a loop head or not. Therefore, also condition 4 holds.

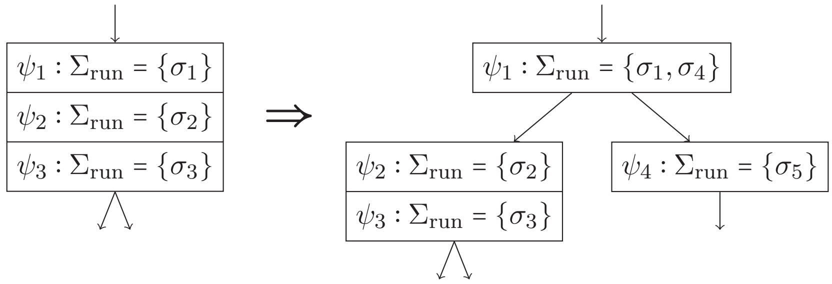

$\psi_1 : \Sigma_{\text{run}} = \{\sigma_1\}$
$\psi_2 : \Sigma_{\text{run}} = \{\sigma_2\}$
$\psi_3 : \Sigma_{\text{run}} = \{\sigma_3\}$
$\Rightarrow$
$\psi_1 : \Sigma_{\text{run}} = \{\sigma_1, \sigma_4\}$
$\psi_2 : \Sigma_{\text{run}} = \{\sigma_2\}$
$\psi_3 : \Sigma_{\text{run}} = \{\sigma_3\}$
$\psi_4 : \Sigma_{\text{run}} = \{\sigma_5\}$

Figure 5.21: Example for block shortening during the PEG construction.

Induction Step $(j-1 \to j)$: The structural condition and the property conditions depend on each other. We will need the induction hypothesis of the property conditions to prove the induction step of the structural condition and vice versa.

Concerning condition 1, we know from the induction hypothesis that $\psi' \rightsquigarrow_{G_P^{j-1}} \psi$ holds. We must then show, that it also holds in G_P^j. Iteration j can only alter G_P in a limited way, namely by removing or adding edges in lines 11 and 13 of Algorithm 9. The incoming APSS $\lambda_{\text{in}}^j(\psi)$ are given through Σ_{run} in line 2 of Algorithm 9. With the induction hypothesis of condition 2, it follows that

$$\lambda_{\text{in}}^j(\psi) = \bigsqcup_{(\psi',\psi)\in E_P^{j-1}} \lambda^{j-1}((\psi',\psi)) \sqsupseteq \bigsqcup_{(\psi',\psi)\in E_P^{j-1}} \lambda^i((\psi',\psi)) \sqsupseteq \lambda_{\text{in}}^i(\psi) \tag{5.35}$$

Due to the monotonicity of the cycle step function $\xi_{\mathbb{X}}$, $\lambda_{\text{in}}^j(\psi) \sqsupseteq \lambda_{\text{in}}^i(\psi)$ implies that

$$\xi_{\mathbb{X}}(\lambda_{\text{in}}^j(\psi)) \sqsupseteq \xi_{\mathbb{X}}(\lambda_{\text{in}}^i(\psi)) \tag{5.36}$$

$$\xi_{\mathbb{X}}(\xi_{\mathbb{X}}(\lambda_{\text{in}}^j(\psi))) \sqsupseteq \xi_{\mathbb{X}}(\xi_{\mathbb{X}}(\lambda_{\text{in}}^i(\psi))) \tag{5.37}$$

$$\dots \tag{5.38}$$

Therefore, every successor of ψ which was reachable in iteration i must also be reachable in iteration j. An example is given in Figure 5.21, where the result of iteration $j-1$ is shown on the left side and the result of iteration j is shown in the right side, with $\lambda_{\text{in}}^{j-1}(\psi) = \{\sigma_1\}$ and $\lambda_{\text{in}}^j(\psi) = \{\sigma_1, \sigma_4\}$, respectively. Thus, even if the block length shrinks, as shown in the example for ψ_1, the application of the transfer function $\xi_{\mathbb{X}}$ will still generate at least one transition to the original successor SEPs (ψ_2 in the example). Therefore, the edge removal in line 11 of Algorithm 9 cannot destroy any previously existing path, which completes the induction step for condition 1.

For the induction step of condition 2 we must show that

$$\forall e \in E_P^i : \lambda^i(e) \sqsubseteq \lambda^j(e) \tag{5.39}$$

under the hypothesis that

$$\forall e \in E_P^i : \lambda^i(e) \sqsubseteq \lambda^{j-1}(e) \tag{5.40}$$

For an edge $e = (\psi_2, \psi_3)$, $\lambda^j(e)$ is computed in Algorithm 9 through one or more applications of $\xi_{\mathbb{X}}$ on the incoming microarchitectural state $\lambda^j_{\text{in}}(\psi_2)$ as shown in Equation 5.41. We use $\xi^+_{\mathbb{X}}$ to denote one or more applications of $\xi_{\mathbb{X}}$. Since $\xi_{\mathbb{X}}$ is monotonic, Equation 5.42 follows with the help of Equation 5.40. Equation 5.43 then follows from the induction hypothesis of condition 1, which completes the induction step for condition 2.

$$\begin{aligned}
\lambda^j((\psi_2, \psi_3)) &= \xi^+_{\mathbb{X}}\left(\bigsqcup_{(\psi_1,\psi_2)\in E^{j-1}_P} \lambda^{j-1}((\psi_1,\psi_2)) \right) && (5.41)\\
&\sqsupseteq \xi^+_{\mathbb{X}}\left(\bigsqcup_{(\psi_1,\psi_2)\in E^{j-1}_P} \lambda^{i-1}((\psi_1,\psi_2)) \right) && (5.42)\\
&\sqsupseteq \xi^+_{\mathbb{X}}\left(\bigsqcup_{(\psi_1,\psi_2)\in E^{i}_P} \lambda^{i-1}((\psi_1,\psi_2)) \right) && (5.43)\\
&= \lambda^i((\psi_2, \psi_3)) && (5.44)
\end{aligned}$$

The monotonic growth of $\lambda(e)$ and thus also of $\lambda_{\text{in}}(\psi)$ directly causes the monotonic decrease of $\omega_P(\psi)$. To see why this is the case, it is important to bear in mind, that $\lambda_{\text{in}}(\psi)$ encodes the possible initial hardware states when executing SEP ψ. If iteration i led to a runtime of $\omega^i_P(\psi)$ cycles, then there must be a $\sigma_i \in \lambda^i_{\text{in}}(\psi)$ which caused this runtime through a basic block end or ambiguous instruction completion. With the induction hypothesis of condition 2, we know that there must be a $\sigma_j \in \lambda^j_{\text{in}}(\psi)$ with $\sigma_i \sqsubseteq \sigma_j$. Therefore, σ_j will also cover the execution paths derived from σ_i, which implies that the basic block end or ambiguous instruction completion which caused the runtime $\omega^i_P(\psi)$ is still reachable in σ_j. Due to $\sigma_i \sqsubseteq \sigma_j$, there may also be *new* execution paths for σ_j which were not possible with σ_i. These may trigger an even *earlier* occurrence of a basic block end or ambiguous instruction completion, therefore $\omega^j_P(\psi) \leq \omega^i_P(\psi)$.

Finally, we have to prove the induction step of condition 4. The δ-values are computed by the path analysis based on the context block runtimes as determined by Algorithm 8. By construction, $\omega^i_C(v_\tau) \subseteq \omega^j_C(v_\tau)$, since we never remove elements from ω_C at all. Additional traces for v_τ may exist in G^j_P which leads to the possibility of $\omega^i_C(v_\tau)$ being not equal to $\omega^j_C(v_\tau)$. With the monotonic growth of the context block durations $\omega_C(v_\tau)$, it is obvious that also the path lengths in δ can only grow, which proves the last condition. □

With the help of Lemma 2 we can then show that Algorithm 7 terminates. The main loop only continues as long as either the PEG or the data-flow information in λ changes. The addition of new blocks and edges is limited, since there are only finitely many different SEPs. According to Lemma 2, edges are also never really removed but only replaced. Therefore, in the worst-case, the PEG becomes a full cycle-level product graph of the different context graphs. For the data-flow values

$\lambda(\psi) \in \mathbb{X}$ we can then apply the usual argument that the lattice $\mathbb{X}$ has finite-height, thus, in the worst-case, the λ-values converge in a finite number of steps towards $\top \in \mathbb{X}$.

Theorem 4. *The context block runtimes ω_C as returned by Algorithm 7 are safe over-approximations of the concrete block runtimes in any possible parallel execution scenario.*

Proof. Any possible concrete, periodic task set execution, can be modeled as an infinite sequence $S = (\psi_0, \psi_1, \psi_2, \dots)$ of SEPs where each ψ_i models that effect after a new cycle step. For each $\psi_i \in S$ there is a corresponding concrete hardware state $\hat{\Sigma}_i$. Due to the construction of the PEG, ψ_0 must be one of the initial PEG blocks and by prerequisite $\hat{\Sigma}_0$ must be contained in $\gamma(\Sigma_{\text{start}})$, i.e., the concretization of the abstract start state. Since the PEG is constructed according to the SEP transitions dictated by the cycle step function $\xi_{\mathbb{X}}$, and this cycle step function is a safe abstraction of the concrete cycle step, the sequence S must be an infinite path through the PEG if and only if the block exclusion criterion did not apply for all analysis iterations of any block $\psi_i \in S$.

If we assume that $\psi_i \in S$ is the first SEP in S where the BEC was applied in every analysis iteration of ψ_i, then we can easily show that this leads to a contradiction. Since the BEC was not applied in all analysis iterations for each of the blocks in $S_{i-1} = (\psi_0, \psi_1, \dots, \psi_{i-1})$, these SEPs must be contained in the PEG. Through the correctness of $\xi_{\mathbb{X}}$ and $\hat{\Sigma}_0 \in \gamma(\Sigma_{\text{start}})$ we also know, that $\hat{\Sigma}_j \in \gamma(\lambda_{\text{in}}(\psi_j))$ for all $j \in \{0, \dots, i-1\}$, i.e., there must be an edge from ψ_{i-1} to ψ_i in the PEG. Likewise, after S_{i-1} is covered in the PEG, the block runtimes ω_P will cover the concrete runtimes of this path and therefore also ω_C will cover the runtimes of the context graph nodes in this path. Therefore, since ψ_i is actually reachable in the concrete execution S, the δ-intersection then cannot be empty at ψ_i, which implies that the BEC cannot be applied at ψ_i.

Therefore, any concrete parallel execution, and thus any concrete context block runtime, is covered in the PEG. □

5.5.8 Extensions

Topologically sorted work queue. To increase the effectiveness of the BEC we have to make sure that each PEG block fulfills the condition $\forall_{k \in \{1,\dots,n_c\}} \delta^i(\psi)^{(k)} \neq \varnothing$ as early as possible. Therefore, we perform a topological sort of the tasks' context graphs which induces an order $\leq_{\text{topo}}: \Psi_\tau \times \Psi_\tau \rightarrow \{\texttt{true}, \texttt{false}\}$ on the task execution positions $\psi_\tau \in \Psi_\tau$ of the respective core. From this, we can derive a topological order of system execution positions as

$$\psi_1 \leq_{\text{topo}} \psi_2 \Leftrightarrow \forall_{i \in \{1,\dots,n_c\}} \psi_1^{(i)} \leq_{\text{topo}} \psi_2^{(i)} \tag{5.45}$$

This order is used to sort the work-list before extracting a new SEP in line 8 of Algorithm 7.

Exploiting timing-anomaly-free architectures. If the underlying architecture is guaranteed to be free of *timing anomalies* (cf. Section 2.2.8), then in each block analysis (Algorithm 9, line 5) we can skip all instruction completion vectors $c_1 \in \kappa$ which are dominated by another vector $c_2 \in \kappa$, i.e., $c_1 \prec_c c_2 \Leftrightarrow \forall i \in \{1, \ldots, n\} : c_2^{(i)} \Rightarrow c_1^{(i)}$. The dominated vectors correspond to an earlier termination of an instruction. Since every local worst-case action is always also the global worst-case action in a timing-anomaly-free architecture, we can assume that they are never part of the worst-case path. This can drastically reduce the state space and the PEG size. Unfortunately, it also renders invalid Lemma 2, since some paths in the PEG will actually be removed now. This may lead to non-termination of the analysis, similar to what we have shown in Section 5.4.7.

Explicit synchronization. In task sets with explicit synchronization points, we have to consider these points in the path analysis as shown in [PP13]. In addition, we can also use them to prune the PEG as we have done in Section 5.5.5, since a task which is waiting for synchronization cannot progress until a partner has arrived to complete the rendez-vous. This idea has already been used in [Tay83a] and similar to there, it can be used on top of the timing information to further prune the PEG.

Non-uniform periods The extension of our framework to task sets with non-uniform periods is also possible. With non-uniform task periods, we can still compute the global hyperperiod, i.e., the smallest common multiple of all task periods and build a PEG for this hyperperiod. In this case, the task execution position must also contain the current position in the core's cyclic schedule. The biggest problem is then, to perform the cycle step for a SEP ψ with $\exists i : \psi^{(i)} = \top$ but $\psi \neq \top^{n_c}$. For any such core i, we need to determine whether the next task execution position for i may be still $\top$ or the begin of the next task instance J_{next} on that core. To determine this, we need information about the current position in the system schedule.

For this purpose we can employ the δ-values. However, since we compute these values based on the local CFG structure we will have $\delta(\psi)^{(i)} = [0, \infty]$ if $\psi^{(i)} = \top$. Due to $\psi \neq \top^{n_c}$, we have at least one core j on which a task is still running. Therefore, we can determine the current time window as $\delta_{\text{now}} = \bigcap_{j \in \{1,\ldots,n_c\}} \delta(\psi)^{(j)}$ just as in the block exclusion criterion. If the next task instance starts at time t_{next}, we know that $\top$ is a possible successor if $\delta_{\text{now}} \setminus \{t \mid t > t_{\text{next}}\} \neq \emptyset$ and J_{next} is a possible successor if $t_{\text{next}} \in \delta_{\text{now}}$.

Still, this yields a lot of possible spawn points for the next task instance. The problem can be further curtailed if synchronization structures are taken into account or better approximations for $\delta(\psi)^{(i)}$ with $\psi^{(i)} = \top$ can be found.

In contrast, SEPs with $\psi = \top^{n_c}$ are not problematic. All successors of these nodes are reached via scheduling edges, not normal control-flow edges. Therefore, we can simply create successors for these SEPs corresponding to the task instances which spawn next according to the global schedule.

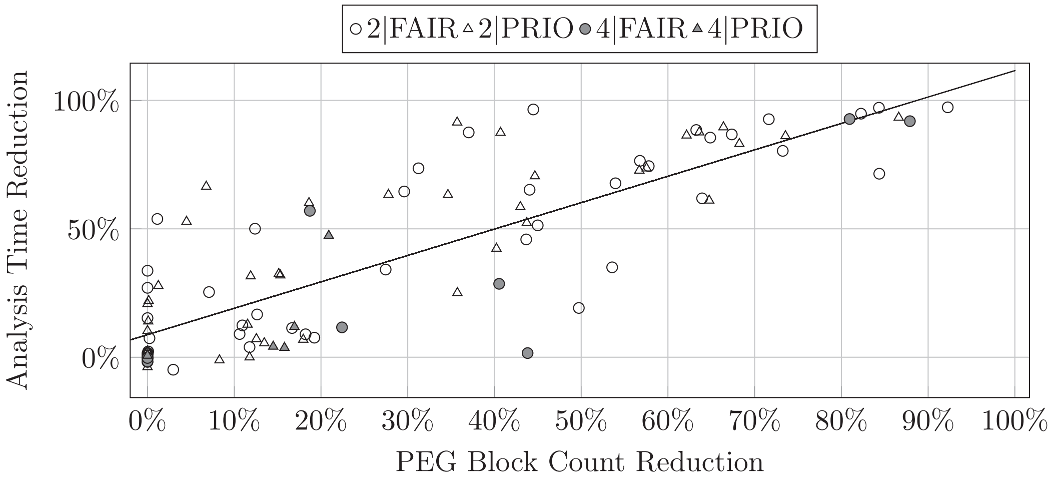


Figure 5.22: Efficiency of the block exclusion criterion on example benchmarks for varying number of cores and arbitration policies. The solid line is a linear regression of the data points.

Multiple-issue processors. We can easily account for multiple-issue cores which can complete multiple instructions per cycle. In this case, the instruction completion vectors $c \in \{0,1\}^n$ are replaced by instruction completion count vectors $c \in \{0,\dots,c_{\max}\}^n$, where $c_{\max}$ is the maximum number of instructions that can be completed in one cycle. All algorithms must be adapted to this change, which results in a possibly enlarged PEG.

5.5.9 Evaluation

We ran our evaluations on single-core tasks from the MRTC and DSPStone real-time benchmark suites. Due to the restrictions of our IPET-based path analysis (cf. page 130) we excluded benchmarks with an irreducible loop structure which require flow restrictions to bound the WCET. Out of these single-core tasks, we formed 35 task sets of size 2 and 6 task sets of size 4. All tasks were assigned a release time of 0. As we will see in the following, the parallelism analysis proved to be very time-consuming, therefore bigger task sets could not be analyzed in the given time limit of two hours per analysis run. Like before, we compiled the benchmarks with optimization level O0, i.e., without optimizations. We analyzed the system topology from Section 3.4 with 2 or 4 cores, depending on the task set. In the evaluation, we first focus on analyzing state-permeable bus arbitration methods (PRIO and FAIR) which were not analyzable (PRIO) or not precisely analyzable (FAIR) without the presented parallelism analysis. The bus which is arbitrated by these methods is the shared memory bus introduced in Section 3.4.

In Figure 5.22, the results of our block exclusion criterion (BEC) from Algorithm 7, line 14 are shown. Each mark represents one analysis run on one task set.

Table 5.4: Average analysis time and PEG sizes.

Schedule	Analysis Type	∅ Duration	∅ PEBs
FAIR	C|N	1s	0
FAIR	P|O|N	389s	963
FAIR	P|B|N	161s	728
FAIR	C|T	10s	0
FAIR	P|O|T	693s	5,455
FAIR	P|B|T	509s	4,046
PRIO	P|O|N	503s	983
PRIO	P|B|N	71s	728
PRIO	P|O|T	776s	5,298
PRIO	P|B|T	322s	3,961

The circle marks indicate runs where the shared bus was configured for FAIR arbitration, the triangles correspond to fixed priority-based arbitration and the squares correspond to TDMA. Non-filled (filled) marks are analysis runs with the 2-core (4-core) system. The x-axis value is the number of PEG blocks that are generated during the analysis, when the BEC is used compared to the case when it is not used (100%). On the y-axis, the required analysis time is shown, also compared to the case that the BEC was not used (100%). From the data points and the solid regression curve, it is visible that the analysis time scales roughly linearly with the number of PEG blocks, which was expected, since the runtime of the main loop in Algorithm 7 depends on the total number of blocks. The variations stem from the convergence behavior of the individual benchmarks, i.e., how often loops have to be visited until the attached APSSs converge. More importantly, we can see from Figure 5.22 that the BEC is effective, since the average of *all* data points in Figure 5.22 corresponds to a reduction of the PEG block count and the analysis time by 35.6% and 49.7%, respectively.

To limit the evaluation runtime we set up a deadline of two hours for each individual analysis run. Our current implementation completed the analysis of the 35 smallest dual-core and the 6 smallest four-core benchmarks in that time frame. It could not complete bigger task sets (8 cores) or sets with bigger task CFGs. Therefore, we limited our evaluation to the aforementioned benchmarks only.

The average resulting analysis time for the benchmarks is presented in Table 5.4. The column “Analysis Type” shows which type of WCET analysis was tested. We compare the classical multi-core WCET analysis as presented in Section 5.4 (abbr. “C”) to our new parallelism analysis with (abbr. “P|B”) and without (abbr. “P|O”) usage of the block exclusion criterion. The last element of the “Analysis Type” column shows whether the absence of timing-anomalies on our platform was exploited by the analysis (abbr. “N”) or not (abbr. “T”). As already seen in Figure 5.22, “P|B” is always superior to “P|O” but both are slower than the classical approach “C” by a

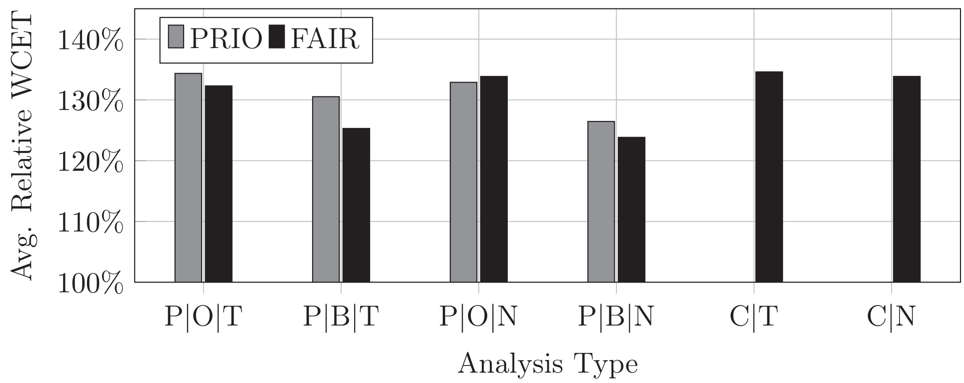
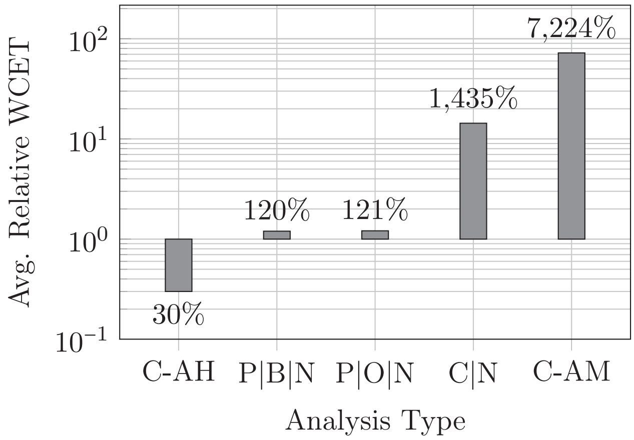


Figure 5.23: Relative WCET results.

factor of 106 or 229, respectively. This is a result of the more complex system state and of the thousands of parallel interleavings that have to be explored, whereas the classical analysis only operates on the CFG of a single task and the state of a single core. For a WCET analysis these runtimes are still acceptable, though. Even mature tools like aiT need 12 hours per task in a 256-task system [SPH+07]. As presented in Section 5.5.8, the exploitation of timing-anomaly-freedom can be used to drastically reduce the PEG size, which is visible in Table 5.4 in column "∅ PEBs", which holds the average number of PEG blocks for this analysis scenario. The configurations where absence of timing anomalies was assumed ("N") produce far lower PEG sizes and analysis times than their counterparts ("T"). As already pointed out in Section 5.5.8, the "N"-type analysis is not guaranteed to terminate in all cases. However, in the analyzed set of benchmarks we did not observe a case of non-termination.

The benefits we get from the parallelism analysis ("P"-configurations) at the price of increased analysis times are that we can *analyze the PRIO arbitration for the first time* and that we can *reduce the arbitration delay estimations for FAIR arbitration.*

Details on both aspects are presented in Figure 5.23. In analogy to Section 5.4.8, it shows the average relative WCET, i.e., the average quotient of WCET and ACET, for different analysis configurations from Table 5.4. The "C"-configurations show the results for the classical WCET analysis framework, which can only assume the maximum possible delay for every access in state-permeable arbitration policies. The plain parallelism-based analysis ("P|O") is able to outperform this approach by 2.3% in the TA-prone analysis ("T"). Only if the timing-based block exclusion criterion is used ("B"), we observe average reductions of up to 10.0% in the case of the TA-free analysis ("P|B|N" compared to "C|N"). At this point it is important to note that the majority of the experiments was done for dual-core benchmarks and the worst-case delay for FAIR arbitration grows with the number of cores (cf. Equation 5.7). Therefore, the parallelism analysis can be expected to yield higher WCET decreases on four-core and eight-core systems. Finally, Figure 5.23 shows



(a) WCET results.

Analysis Type	⌀ Duration
C-AH	0.8s
P|B|N	1,489.9s
P|O|N	2,090.2s
C|N	2.0s
C-AM	1.9s

(b) Analysis duration results.

Figure 5.24: Results for different analysis methods of shared caches averaged over 36 dual-core task sets.

that the PEG-based WCET analysis for a system with PRIO arbitration yields results that are comparable to those for FAIR arbitration.

We have also evaluated the performance of the parallelism analysis with respect to shared caches. Obviously, shared caches can only be analyzed together with a shared bus which connects them to the cores. Since we want to exclude the effect of shared bus analysis in the following experiments, we have set it into an "always assume best-case" mode. In this mode, the analysis assumes that the bus is immediately granted on each request of the analyzed task. This allows us to definitely attribute the changes in WCET results to the shared cache analysis. To make the cache analysis reasonably precise, we only activated the L1 and L2 instruction caches, moved the task code into the cached RAM (cf. Figure 3.4), and unrolled the first iteration of each loop. Following the methodology from Section 4.5, the L1 cache size was set to 50% of the core's tasks code size and the L2 cache size was set to 50% of the overall code size.

The average results on 36 dual-core task sets are shown in Figure 5.24. Similar to Section 4.5, we compare the results against the "always-hit" (C-AH) and "always-miss" (C-AM) scenario, which provide lower and upper bounds on the possible results of any shared cache analysis. First of all, we see from Figure 5.24a that the always-miss configuration produces a *far* higher pessimism than in the single-core case, where it led to an average relative WCET of 2318%. This is plausible, because in Section 4.5 we examined a L1 cache only. Of course, every cache read miss includes a cache line refill. Since the bus that connects the L1 cache to the higher memory hierarchy levels has limited width, we need multiple bus requests to fill the cache line. In our system configuration from Table 3.1 we need 32 B/4 B = 8 shared bus and shared cache accesses to fill the L1 cache line. The C-AM configuration must now assume that *every* L1 access is a miss and that *all* of the 8 line refill accesses are again L2 misses which trigger L2 cache line refills. Since each L2 cache line refill takes $2 + 16 * 3 = 50$ cycles, the total duration of a memory access rises from 1 cycle (L1 hit) to $1 + 8 * 50 = 401$ cycles. This shows the large potential for pessimism in any shared cache analysis.

The summary-based shared cache from Section 5.4.1 which is shown in Figure 5.24 as "C|N" yields a high average relative WCET of 1435%. This is significantly worse than the single-core results, which showed an average relative WCET of 645% in Section 4.5. The main drawback for the summary-based approach is that there are always memory accesses for which the value analysis fails to determine an access range bound, i.e., these accesses may go *anywhere* including the task's code section. A single access of this type in a task τ is sufficient to raise the cardinality of the interference map $tags(\tau, s)$ for every cache set s far above the associativity $\hat{a}$ since τ may then access all possible tags. Therefore, a single access of this type in task τ also degrades the classification of every shared cache access in $\tau' \in T \setminus \{\tau\}$ to $\{\mathtt{HIT}, \mathtt{MISS}\}$ according to Equation 5.1. Obviously, every additionally introduced consideration of a `MISS` case impairs the WCET as mentioned in the discussion of the C-AM configuration.

The parallelism-aware analyses ("P|B|N" and "P|O|N" in Figure 5.24) are *not* making any such summary-based approximations, but simply use the original abstract LRU cache states as described in Section 4.3.2. The parallelism analysis framework already takes care of exploring every possible access order. In addition, the shared bus modeling resolves truly concurrent accesses by either determining a guaranteed winner or by splitting the microarchitectural state. This increased effort translates into an average relative WCET of only 120% if the block exclusion criterion is used (configuration "P|B|N"). The WCET difference to the case in which the BEC is not used ("P|O|N") is marginal here, but as visible from Figure 5.24b the analysis duration is reduced by 28.7% when the BEC is used. However, similar to our previous experiments, there is a vast increase in analysis duration compared to the task-partitioned approach. On the other hand, in this scenario the parallelism-aware analysis can really make use of its superior knowledge about concurrent events and outperform the summary-based approach by a *factor* of 11.96 on average.

5.6 Summary

This chapter has introduced two distinct approaches towards the modeling of shared resources in the WCET analysis of multi-core systems. Both are based on non-deterministic finite-state-machines which are also used in the single-core analysis to model abstract hardware states (cf. Section 4.3).

For a *task-partitioned* WCET analysis which analyzes the tasks in isolation, only time-triggered arbitration schemes could be analyzed with good precision as shown in Section 5.4. For these schemes, we have presented cyclic data-flow contexts, called *offset contexts*, and the *offset relocation* technique to speed up the analysis and still gain close-to-optimal results. We have also seen that shared caches and *state-permeable* arbitration policies can only be analyzed with the help of coarse-grained worst-case assumptions in a task-partitioned analysis.

To overcome this problem, we also investigated a *unified* multi-core WCET analysis in Section 5.5, which explores the possible interleavings of a parallel, periodic task set. It was shown that this approach can handle state-permeable arbitration policies with very good precision, being up to 10% more precise on average than a task-partitioned analysis of fair round-robin arbitration and 11.96 times more precise than the best preexisting shared cache analysis. In addition, this is the first approach which enables the analysis of fixed-priority arbitration. The combinatorial explosion which is caused by the enormous search space of parallel configurations could be limited to some extent by applying a new, timing-based *block-exclusion criterion* to exclude infeasible interleavings. Experiments have shown that the BEC can halve the analysis duration and decrease the resulting WCET by up to 10%. We also pointed out, that synchronization statements in the program can be used in addition to further reduce the search space of parallel configurations.

Chapter 6

Multi-Core WCET Optimization

Contents

In the previous chapter, we have discussed the implications of different types of arbitration methods on the achievable analysis precision. In particular, the analysis of time-triggered arbitration methods was found to be comparatively fast and precise in Section 5.4. However, the WCET and ACET performance of systems running time-triggered arbitration methods is highly dependent on

- the parameterization of the schedules and
- the structure of the examined tasks.

The analyses work best for tasks where the distribution of accesses in the time domain matches the time-triggered schedule of the resource. Therefore, two novel optimizations were developed that address the two points mentioned above. These were first published by the author of this thesis in [KMB14]. The first is an evolutionary optimization of the shared bus schedule parameters, whereas the second is a multi-core WCET-aware instruction scheduling which re-structures the tasks to increase their performance on a given time-predictable multi-core platform. Both optimizations aim for a decrease of the WCET and ACET of the given tasks, which in turn leads to improved schedulability and increased resource utilization.

6.1 Multi-Objective Evolutionary Schedule Optimization

In this section, we present a multi-objective evolutionary search algorithm which automatically determines a range of well-suited schedules for a task set. It enables

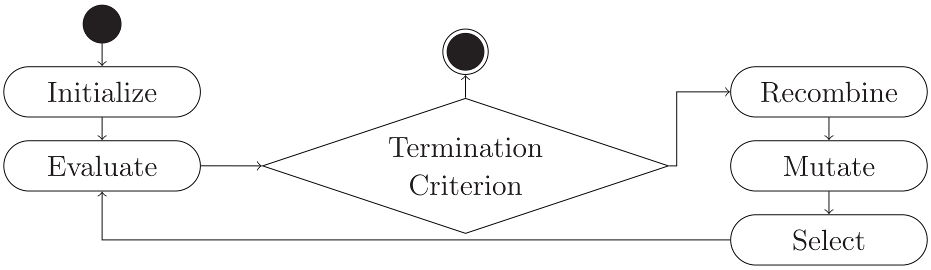


Figure 6.1: The structure of the evolutionary bus schedule optimization.

users to choose a solution which balances WCET, ACET and utilization according to their needs, and thus avoids a manual search for an optimal schedule, since this is a hard and error-prone task.

6.1.1 Related Work

The optimization of bus schedules has been the topic of a range of previous publications, but the vast majority either is restricted to TDMA schedules or uses ad-hoc WCET computations instead of an analyzer following the design principles presented in Section 2.2.

The optimization in [RNE+11] and [AEP+08] by the same authors is based on the evolutionary optimization technique *simulated annealing*. It integrates system-wide task scheduling with optimization, but on the other hand, it is restricted to TDMA schedules, whereas we also consider more flexible schedule variants. TDMA slot length allocation is also done in [WT06], but the employed WCET analysis framework is less precise and it is again restricted to TDMA. Concerning the employed evolutionary variation operators, we use an approach similar to [HE05], but [HE05] is restricted to TDMA and considers the optimization at a far more coarse-grained level, i.e., the scheduling of tasks as a whole. Finally, [YKS11] also examines bus schedule optimization, but only for the special case of *Harmonic Round-Robin* schedules and for additive WCET models.

To the best of our knowledge, previous work has not addressed the optimization of real-time bus schedules including TDMA *and* more flexible methods. We will see in the following that especially the consideration of schedule types other than TDMA is important to achieve the highest WCET and ACET gains.

6.1.2 Evolutionary Algorithm

The structure of the optimization is depicted in Figure 6.1. It starts with a set of initial schedules, which are called *individuals* in the context of evolutionary optimization. For all individuals, the WCET, ACET and bus utilization values are determined. Then, promising individuals are recombined and mutated with a certain probability. After these steps, the optimizer selects those individuals that form

t	n_l	$\vec{p} = (p_0, \dots, p_{63})$	$\vec{l} = (l_0, \dots, l_{63})$	$\vec{o} = (o_0, \dots, o_{63})$

Figure 6.2: The evolutionary algorithm's genome.

the next generation and the optimization continues with them. The steps are repeated until a user-definable termination criterion is met, e.g., until a predefined result quality is achieved or until no further increase was observed over a predefined time frame.

In our schedule optimization, individuals are represented by the genome shown in Figure 6.2. It contains the scheduling policy t (one of FAIR, PRIO, TDMA or PD), the number of slots n_l and the vectors of priorities, slot lengths and slot owners ($\vec{p}$, $\vec{l}$ and $\vec{o}$). For an efficient recombination and mutation, the genome needs a fixed length. Therefore, each vector is limited to 64 entries, which limits the solution space to 64 slots. We will examine systems with up to 8 cores, and our initial experiments have shown that good solutions almost exclusively use a minimum number of slots. This is also supported by the fact that the last $T^B_{\max} - 1$ cycles of each slot are "wasted" since no access can be scheduled here (cf. Figure 5.3). Therefore, it can be expected that the limitation to 64 slots does not degrade the solution quality.

Initialization

The optimization process starts with a set of schedule candidates also called the *population*. It contains a FAIR schedule, a uniform PRIO schedule, a uniform TDMA schedule and a uniform PD schedule. Here, "uniform" means that all cores get one slot ($n_l = n_c$) and all slot lengths are set to the minimum allowed size ($\forall i : l_i = T^B_{\max}$), since from our experience this reduces the bus arbitration delay on average. Slot priorities are distributed such that the cores get a priority equal to their core ID ($\forall i : p_i = i$).

The rest of the population is filled up with candidates for which the parameters are randomly chosen according to a uniform distribution. The randomness is needed to appropriately cover the search space and is a standard approach in evolutionary optimization [Wei07].

Recombination and Mutation

Similar to [HE05], we use *arithmetic operators* which do not treat the genome as a bit string and flip individual bits, but which perform arithmetic operations on the contained parameter values. This is done to limit the degree of randomness in the optimization, since otherwise flipping a high-order bit of a parameter might cause the optimization to unguidedly "jump around" in the parameter space.

The recombination works piecewise on two genomes, with a multi-point crossover. That is, during the recombination of A and B, for each segment $\sigma \in \{t, n_l, \vec{p}, \vec{l}, \vec{o}\}$ from Figure 6.2 a recombination point $r \in \{0, \dots, l_\sigma\}$ is determined randomly with uniform distribution, where l_σ is the length of σ. The new segment σ^C for the resulting individual C is then given as the concatenation of the substrings $\sigma^A_{[0:r)}$ and $\sigma^B_{[r:l_\sigma)}$. l_σ always denotes the *effective* length of the segment, e.g., we may have up to 64 slots, but if A and B only use 7 slots at maximum, then $l_{\vec{\sigma}} = 7$.

The mutation is also only applied for parameters within the effective lengths and mutates each segment's values with probability 0.3. To restrict the step size, we use δ-mutation, where a new value v_{new} is randomly chosen from $[v_{old} - \delta, v_{old} + \delta]$. For the number of slots s the value of δ is 5, for the slot lengths $l \in \vec{l}$ we chose $\delta = 30$.

Finally, we perform a randomized *genome repair* step, which mutates the individual until each core has a unique priority and each core is the owner of at least one slot. The first is a requirement of our platform, whereas the latter is needed to avoid core starvation and thus infinite WCET values. The intention behind all of these design decisions is to increase the chances that we find good solutions early, since the objective evaluation and thus each new generation is costly in our scenario as we have seen in Section 5.4.8.

6.1.3 Evaluation

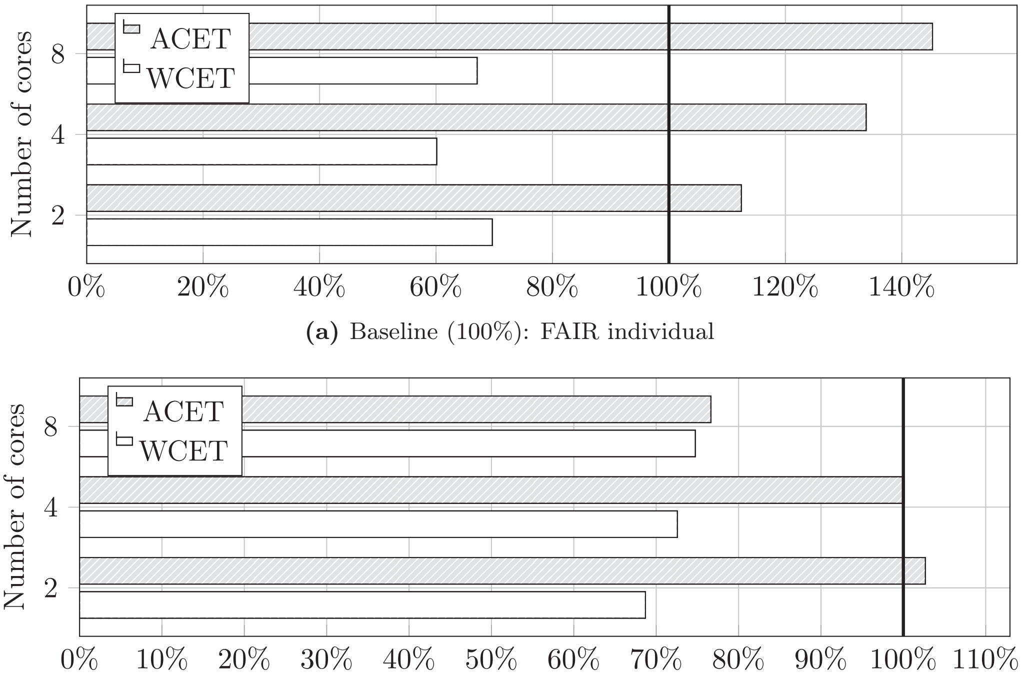
We implemented the evolutionary optimization with the PISA framework [BLT+03], using the SPEA2 selector [ZLT02], which is used to determine individuals for mutation, recombination and generation survival. SPEA2 tries to keep individuals in the population that are not *pareto-dominated* by others, i.e., for which no other individual exists which is better in all objective values. In addition, SPEA2 increases the *diversity* of the generated solutions by maintaining a solution density value for each individual which indicates how different it is with respect to all other individuals.

For the WCET analysis, we used the task-partitioned approach with the offset relocation technique as presented in Section 5.4.6 due to its superior analysis duration. The ACET and utilization values were again determined with the CoMET simulator [Syn14].

In total, we used 110 tasks from the UTDSP, MRTC, MiBench and MediaBench suites [LCS92; Mäl05; GRE+01; LPM97]. The tasks were compiled with optimization level O1 which includes only the most basic compiler optimizations. As described in Section 5.4.8 and Section 5.5.9, the tasks were grouped by their ACET and only their input and output is read from and written to the shared memory. Thus, only the I/O operations issued by the tasks are subject to bus arbitration, the tasks' code and local data are stored in the scratchpads of the cores.

We used a generation size of 20 individuals and a minimum number of 20 generations. After the 20th generation, optimization is continued if the current generation is at least 0.05% better in any objective than the previous one. This was added to provide confidence that we do not abort the optimization prematurely, but we encountered no cases where the 21st generation was actually reached.

We present relative results in the following, where the first meaningful baseline is the FAIR individual as this represents the current practice in many real-world



(a) Baseline (100%): FAIR individual

(b) Baseline (100%): Uniform TDMA individual

Figure 6.3: Average results for the best-WCET individuals.

systems. Figure 6.3a shows the geometrical mean of the relative WCET (ACET) of the final-generation individuals with the best WCET, relative to the WCET (ACET) value of the FAIR individual from the first generation. Since the FAIR individual has no parameters and thus never evolves, it does not matter from which generation it is taken. It can be seen that the reduction in WCET of up to 39% in the case with 4 cores is accompanied by an increase in ACET. This is plausible, because most of the best-WCET individuals are using TDMA or PD (see Figure 6.5), which have better WCET, but worse ACET performance (cf. Section 5.4.8).

As the second baseline, we chose the uniform TDMA schedule with minimum slot length, which usually produces good WCET values. Here, the question is whether the optimization can still improve upon this baseline. As can be seen in Figure 6.3b, we can still observe WCET improvements of 31% (2 cores) to 25% (8 cores) without significant loss of ACET performance. Also note that Figure 6.3 contains the results for the individual with the *best WCET*. Thus, if we want to balance ACET and WCET, the evolutionary approach also delivers matching solutions, some of which are presented in the following.

To indicate the distribution of the results among the benchmarks, Figure 6.4 shows the detailed WCET results for all benchmarks in the 2-core configuration. Each segment in the figure represents the best-WCET individual for one benchmark, which is identified by its benchmark ID, shown on the x axis. All WCETs are relative to the WCET of the uniform-TDMA individual, which was also taken as

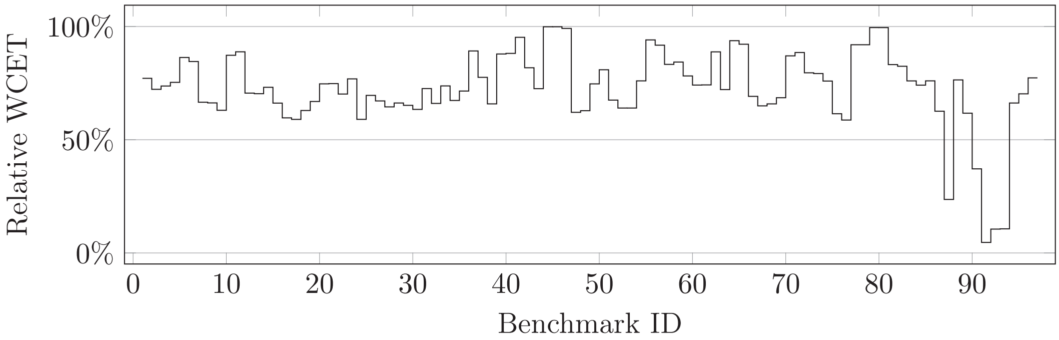


Figure 6.4: Detailed results for the best-WCET individuals on the 2-core platform (Baseline: Uniform TDMA individual).

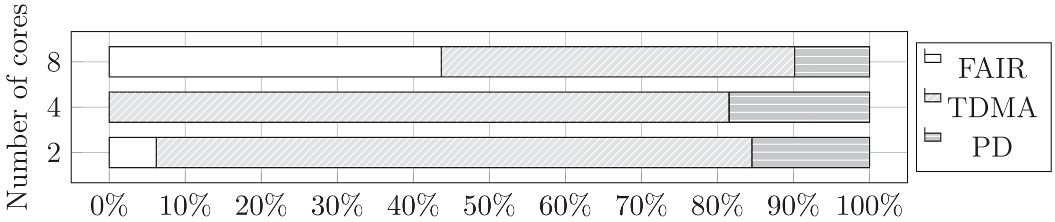


Figure 6.5: Distribution of different schedule types among best-WCET individuals.

the comparison base in Figure 6.3b. It is visible that the average WCET reduction is achieved by a uniform distribution of WCET reductions. The few benchmarks which experience WCET reductions larger than 50% are unbalanced examples, where one task with the largest WCET needs much more bus bandwidth than the others, and thus its runtime can be drastically decreased by assigning more or longer slots to it.

The best-WCET individuals constitute of TDMA schedules with adapted slot lengths, of even more customized PD schedules and of some FAIR schedules. The distribution of the schedule types is depicted in Figure 6.5. Note that in the worst case, a FAIR access may have to wait for at most one access from all other cores. A TDMA access that is issued too late in the issuer's slot to finish inside that slot may have to wait for the rest of the slot *plus* the slots of all other cores. Due to this, for tasks on which the TDMA WCET analysis fails to produce precise results, FAIR can be better than TDMA. Apparently, this mostly happens for the platform with 8 cores. This platform requires a longer TDMA schedule to still provide at least one slot per core and it seems that the WCET analysis gets more imprecise with growing schedule length. Nevertheless, it works well for most examples, making TDMA the predominant schedule type among the best-WCET individuals.

The fact that the optimization performs better for systems with fewer cores can also be explained when examining the baseline utilization of the shared bus as depicted in Figure 5.11. For 2 cores, the current benchmark set has an average bus

Mode	s	WCET	ACET	Utilization	$\vec{l}$	$\vec{o}$	$\vec{p}$
FAIR	-	95076900	5752270	0.6596	-	-	-
PD	8	85379400	7085960	0.5496	$(3, 3, 3, 3,$ $3, 3, 3, 3)$	$(2, 1, 7, 5,$ $4, 0, 6, 3)$	$(7, 3, 1, 4,$ $5, 6, 8, 2)$
TDMA	8	61227900	10725600	0.3870	$(10, 3, 3, 3,$ $3, 3, 3, 3)$	$(2, 3, 1, 0,$ $4, 5, 6, 7)$	-
TDMA	8	85379400	9383380	0.4421	$(3, 3, 3, 3,$ $3, 3, 3, 3)$	$(0, 1, 2, 3,$ $4, 5, 6, 7)$	-

Table 6.1: Details on the pareto-optimal individuals from Figure 6.6.

load of 21%, which rises to 41% for 4 cores and 64% for 8 cores, measured under FAIR scheduling. Thus, all attempts to increase the utilization are ultimately limited by the amount of unused bus time, which is decreasing as the number of cores increases.

The development of the individuals during a single optimization run is illustrated in Figure 6.6 which shows the WCET, ACET and utilization for *all* individuals that were evaluated in the course of the optimization of an 8-core benchmark containing mixed multimedia and control tasks (`codecs-dcodhuff`, `lmsfir-32-64`, `fft-256`, `selection-sort`, `edge-detect`, `latnrm-32-64`, `adpcm-decoder` and `adpcm-encoder` from Appendix A). WCET and ACET are shown on the x and y axes, whereas the color of the marks indicates their utilization, as shown in the color bar under the Figure. The axes are scaled logarithmically to accommodate the spread of the results.

The PD individuals all show a good ACET performance, but vary in their WCET by more than one order of magnitude depending on the configuration. In contrast, the TDMA individuals stringently have a worse ACET performance which is compensated by a bigger span of WCET values – they provide both the best and the worst WCET values. The utilization is directly proportional to the ACET, which confirms our expectation that higher utilization implies lower average bus access delays.

The pareto-optimal points are represented by the blank symbols on the left side of the figure. They are also listed in detail in Table 6.1 together with their slot length, owner and priority vectors $\vec{l}$, $\vec{o}$ and $\vec{p}$. As can be seen, FAIR produces the best utilization and ACET values (the triangle in Figure 6.6) and TDMA has the best WCET value (the blank squares in Figure 6.6). In between we find TDMA and PD configurations, where, in this case, PD provides a significantly enhanced ACET without loss of precision at the WCET side (blank circle in Figure 6.6). This distribution of results is typical and could be observed for most benchmarks.

The evolutionary optimization of a single task set takes 3 to 4 hours on average, depending on the number of analyzed cores. Taking into account that even the evaluation of a single configuration takes minutes on average, and that almost all of the time is spent on the WCET analysis (59%) and the CoMET simulation (36%), this is still reasonable for, e.g., nightly builds of a software. Also, the optimization

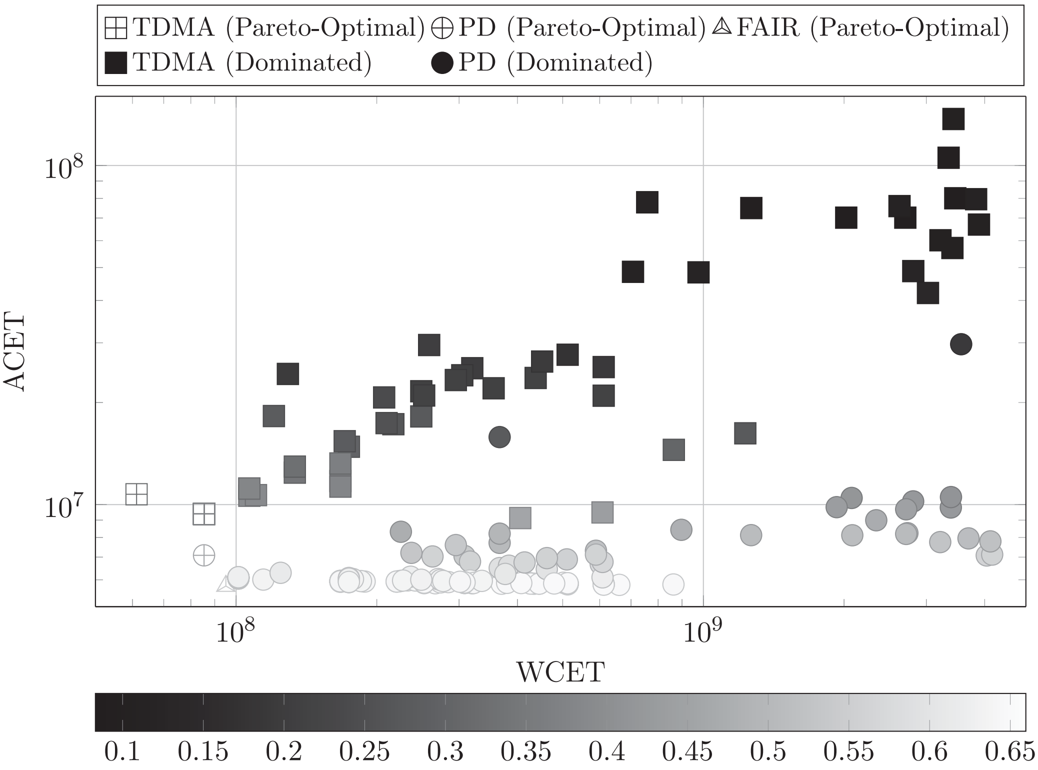


Figure 6.6: Exemplary population with marked Pareto-Front for a benchmark with 8 cores.

itself is trivially parallelizable, which we have not done here. The WCET analysis and simulation runtime will scale linearly with the number of cores, but the total runtime of the optimization until "good" solutions are found might grow faster than linear since more parameters will have to be explored.

All in all, we have seen that FAIR arbitration is strong on producing good ACET values, and that it can even outperform more predictable arbitration schemes especially when the minimum schedule length increases as, e.g., for systems with rising core numbers. TDMA has proven to be the best choice for WCET. Still, for TDMA as well as for PD, an optimization of the schedule parameters is highly desirable and may lead to WCET improvements of more than 30%. PD can be used to balance ACET and WCET which again is easier to do in an automated way.

6.2 WCET-driven Multi-Core Instruction Scheduling

In Section 6.1, we have examined the possibilities of adjusting the bus schedule parameters to the given task set. But there is another degree of freedom, namely to reorder the instructions inside the tasks to match the bus schedule. Since both optimizations are interdependent, we perform the instruction reordering for every single individual that is generated by the algorithm from Section 6.1. Thus, we may also find solutions in which the bus schedule only excels when combined with a custom instruction schedule. Alternatively, the instruction reordering can also be invoked separately for a given, user-defined schedule.

We build upon a classical *list scheduler* [ALS+07] here, which partitions the task into non-overlapping, sequential *regions* and schedules each of those separately. A region $(v_1, \ldots, v_n)$ is therefore defined as a path through the CFG of the task (cf. Definition 2), which is a sequence of adjacent basic blocks v_i. The instructions of the regions are re-ordered by our scheduler. The scheduler maintains a list of dependencies of the instructions *on* the path, and a set of instructions which are *ready* for execution, i.e., whose dependencies have been fulfilled. Our task is to assign a *priority* to them, and the scheduler will then select one of the instructions with highest priority and append it to the result order. This process continues in the same manner until all instructions of the region have been scheduled. If a region contained more than one basic block, instructions may have moved into a different block after the optimization. To restore the original semantics of the task, it is necessary to add *compensation code* in this case [Fis81] after the scheduling of the region. Since this only occurs if basic block boundaries are crossed, compensation code is never needed if the regions only contain a single basic block. The whole procedure is a compile-time optimization, there is no runtime scheduling involved.

In contrast to the evolutionary optimization of the schedule, the instruction scheduling is *not* a multi-objective optimization. It exclusively focuses on the optimization of the WCET. To achieve this, it will use the results of the WCET analysis to be able to assess the effect of code changes on the WCET. This is a crucial step, since unconditionally applying an optimization without such an assessment may even lead to degraded results [ZCS03].

6.2.1 Related Work

The majority of previous publications on WCET-aware instruction scheduling is focused on optimizing the WCET of a *single-core system* [ZKW+05; HZX12]. As an exception, [SPC+10] discusses several access models for time-predictable multi-cores on an abstract level, but requires manual restructuring of the tasks. In contrast, the instruction scheduler presented in this thesis can be used to automatically implement these models on a microarchitectural scale.

A higher-level approach to WCET-aware scheduling is taken in [DZ10], where a cache-aware task scheduling algorithm is presented together with a greedy minimization of the shared cache interference. This work is orthogonal to ours, since we are concerned with low-level instruction scheduling, whereas [DZ10] focuses on operating-system-level task scheduling.

To the best of the author's knowledge, previous work has not addressed the scheduling of instructions according to the requirements of a time-predictable multi-core platform.

6.2.2 Scheduling Heuristics

In the following, we present two novel priority assignment heuristics that are tailored towards the optimization of the WCET of tasks running on time-predictable multi-cores. Since we want to exploit the information generated during the WCET analysis, we first need to establish a connection between the region to schedule and the WCET analysis results.

For any context block $v^C \in V_\tau^C$, we have gathered the set of initial microarchitectural states $q^M \in q^{\text{in}}_{v^C}$ through the data-flow analysis framework presented in Section 4.3. Each of these microarchitectural states is a tuple which, among others, contains a bus state q^B. According to Definition 21, q^B is an offset set for time-triggered schedules. Additionally, each q^M can be used to determine a runtime interval for the execution of v^C with initial state q^M using the abstract execution function given in Equation 4.17. The union of the runtime intervals for all q^M is called $\omega(v^C)$ according to Definition 19.

To make these results usable for the optimization, we first need to map the WCET results, which were obtained based on the context graph, back to the original CFG of the task. Context graph nodes were generated from the CFG with the help of three distinct construction steps:

- **Context copies**. According to Definition 14, a context graph G_τ^C holds one or more copies of every node $v^A \in G_c^A$, i.e., to represent the execution of v^A in different contexts. We require a function *ctxts* : $V_c^A \to 2^{V_\tau^C}$ which maps each analysis graph node to its context copies.
- **Sequential splits**. These are produced by the analysis graph construction in Algorithm 2. As visible in Figure 4.3, every basic block v is partitioned into a sequence of analysis blocks. The analysis blocks which model the *entry* into v are given by $\delta_\perp^A(v)$.
- **Alternative copies**. Like the sequential splits, these are generated by Algorithm 2, since every block with a predicate other than "AL"/"always" is represented by two analysis blocks (cf. Figure 4.3b). The potential alternative copies of the start block are contained in $\delta_\perp^A(v)$. Actually, they are the reason why $|\delta_\perp^A(v)| = 2$ is possible.

Therefore, we can map the results of any data-flow analysis, performed on the context graph, back to the CFG nodes $v \in V_c^f$ as follows:

$$q_v^{\text{in}} = \bigsqcup_{q^A \in \delta_\perp^A(v)} \; \bigsqcup_{v^C \in ctxts(v^A)} q^{\text{in}}_{v^C} \tag{6.1}$$

Then, q_v^{in} represents the data-flow information at the start of CFG node v. In the following, we will use the mapped-back results of the value analysis ($q_v^{\mathbb{V},\text{in}}$) and the microarchitectural analysis ($q_v^{\mathbb{M},\text{in}}$).

For the optimization of task τ running on core c, the *slot length heuristic* (SL) first determines the length l_c^{max} of the longest slot which is assigned to c. During the scheduling of a region, it maintains a counter l_c^{cur} which is set to 0 at the start of the region. With the help of the maximum bus access duration $T_B^{\max}$, the priority[1] of instruction $i \in I$ is given by the priority function p_{SL} as

$$p_{\mathrm{SL}}(i) = \begin{cases} 1 & \text{if } bac(i) \wedge (l_c^{cur} + T_B^{\max}) \leq l_c^{max} \\ 0 & \text{else} \end{cases} \tag{6.2}$$

where $bac : I \rightarrow \{true, false\}$ determines whether an instruction will possibly access the shared bus. After an instruction i with $bac(i) = true$ was scheduled, l_c^{cur} is incremented by t^{max}, otherwise $l_c^{cur} = 0$. The intention is to bundle bus-accesses to packages which fit into the slots of the core. The *bac* function can be implemented by inspecting the mapped-back value analysis results for the current region. If the address range of any memory-accessing instruction i covers the addresses that are served by the shared bus, then $bac(i) = true$, else $bac(i) = false$.

The second heuristic, called *offset heuristic* (OF), uses the results of the WCET analysis more directly. For each region $(v_1, \ldots, v_k)$, it first determines the incoming microarchitectural state $q_{v_1}^{\mathbb{M},\mathrm{in}}$ as presented in Equation 6.1. After a new instruction i was scheduled, the transfer function of i is invoked, i.e., cycle steps of the abstract states are performed until i is "committed". This results in a new incoming microarchitectural state for the next instruction to schedule. Therefore, we can maintain a $q_{\mathrm{cur}}^{\mathbb{M}}$, which is the current microarchitectural state *before* the execution of the next instruction. From $q_{\mathrm{cur}}^{\mathbb{M}}$ we can derive the union of the possible current TDMA states $q_{\mathrm{cur}}^{B} = \bigcup_{q_i^B \in q_i^M, q_i^M \in q_{\mathrm{cur}}^{\mathbb{M}}} q_i^B$. With this information, we can determine whether i is guaranteed to be granted the bus or not, and define the priority function p_{OF} as

$$p_{\mathrm{OF}}(i) = \begin{cases} 2 & \text{if } bac(i) \wedge q_{\mathrm{cur}}^{B} \subseteq \gamma(c) \\ 1 & \text{if } \neg bac(i) \\ 0 & \text{else} \end{cases} \tag{6.3}$$

where $\gamma(c)$ is the grant window of core c as defined in Equation 5.11. The idea is to force the immediate scheduling of a bus-accessing instruction when we know for sure that the access will be granted (case 1), and to delay it if possible when the access will not be granted (case 3). All instructions which do not require the bus use a default priority (case 2).

As an example, consider Figure 6.7 which shows a task's control-flow graph in the upper half. For this example, we assume a system with 2 cores, where the presented task is executed on core 0. The TDMA bus schedule consists of 4 slots, whose length and owner cores are depicted in the bottom of the figure. Below the graph, the set q_{cur}^{B} for the first instruction of block `L4` is shown, which is a subset of the full TDMA offset span, marked in gray. Thus, the analysis has determined for this example that the load instruction `ldr` at the head of block `L4`, which accesses the bus, will always start its execution from one of the offsets contained in the

[1] Higher values indicate higher priority.

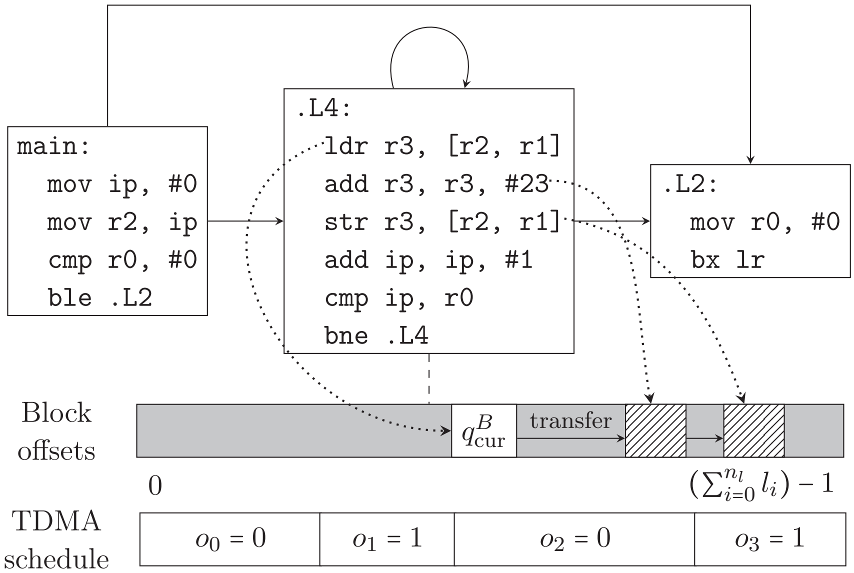


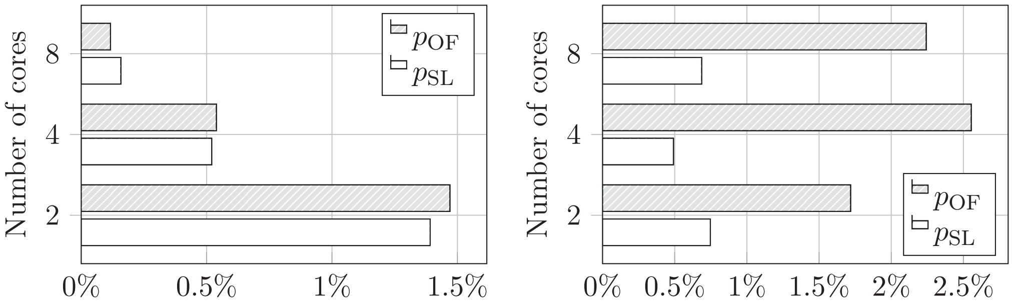
Figure 6.7: An example for offset results as computed during the multi-core WCET analysis.

white rectangle marked in the schedule. Considering the schedule, we know that this area is contained in slot 2 which is owned by core 0, and thus the access will be granted immediately. Therefore, $p_{\text{OF}}(\texttt{ldr}) = 2$. After the `ldr` instruction was scheduled, the analysis will compute new offsets which reflect the positions in the schedule at which the execution of the next instruction will start. The striped areas in Figure 6.7 represent q^B_{cur} for the second and third instruction of `L4`. The dotted arrows indicate which offset information belongs to which instruction. In this case, the optimization can decide that after the execution of the first `add`, the following `str` has to wait for the bus in slot 3 and thus can prefer to schedule the second `add` and `cmp` first.

6.2.3 Evaluation

To evaluate the effectiveness of the heuristics, we tested the scheduler on the same benchmarks that were used in Section 6.1.3. We first evaluated scheduling at the basic block level, thus “regions” are “basic blocks” in the following. The basic blocks consisted of 1 to 661 instructions (average: 5.25), and 11.42% of those were accessing the bus on average. In total, the benchmarks contained 82, 133 instructions.

In Figure 6.8a, the average results for the scheduling are shown for a uniform TDMA schedule, where each core has one slot of length 3 cycles. Both heuristics perform equally well in this setting. This may be due to the fact that in this setting, only one bus access fits into each TDMA slot since the maximum bus access duration is also 3 cycles. Therefore, the schedule is short and it is sufficient to keep the bus accesses isolated, which both heuristics are capable of.



(a) Average WCET reductions for uniform TDMA with slot length of 3 cycles.

(b) Average WCET reductions for uniform TDMA with slot length of 12 cycles.

Figure 6.8: Average results per platform for scheduling with the slot length heuristic (p_{SL}) and offset heuristic (p_{OF}).

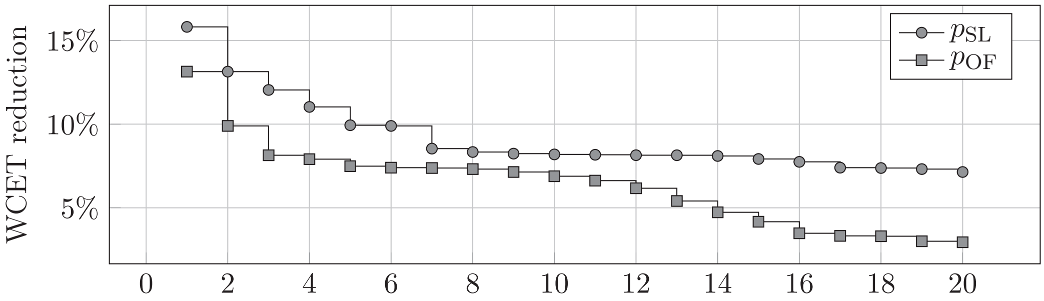


Figure 6.9: Relative WCET results for the best 20 benchmarks from Figure 6.8a per scheduling method.

To test the sensitivity of the optimization w.r.t. to different schedule configurations, we also tested it on a uniform TDMA schedule with a slot length of 12 cycles for which the results are shown in Figure 6.8b. Here, the WCET reduction achieved by p_{OF} is up to 4 times higher than the reduction for p_{SL}. The drawback is, that the absolute WCET values for the 12-cycle configuration are 25% (2 cores) to 158% (8 cores) worse than those for the 3-cycle one, as already pointed out in Section 5.4.8. It is a general observation in our experiments, that bigger TDMA slots lead to worse WCET and utilization values, which defeats the value of the increased optimization potential in these configurations.

Since the scheduler works on the microarchitectural level, it cannot be expected to have as much impact as the macroscopic schedule parameter optimization presented in Section 6.1. To illustrate the results for the individual benchmarks, Figure 6.9 lists the 20 highest WCET reductions from the results shown in Figure 6.8a. In this range, we observe an increased average bus utilization of 14% and 4.4 instructions per region. The WCET reductions for the individual benchmarks range from 13.2% to 2.5% (p_{OF}) or 15.8% to 7.2% (p_{SL}), respectively. Therefore, though the results are lower on average, we still have many benchmarks for which the scheduler achieves significant gains with both heuristics.

The compilation times have tripled compared to the compilation without the WCET-aware scheduling, but again, this is mostly due to the runtime of the WCET analyses. Also, to be precise, we would need to recompute the WCET and thus the microarchitectural states after each scheduled region. However, this would lead to a vast increase of the optimization runtime. Therefore, we performed the whole scheduling with the microarchitectural results for the original task. This leads to some degree of imprecision, but otherwise the compilation time can be expected to scale linearly with the number of regions (basic blocks, in this case) which is not acceptable.

We also extended the optimization to work on trace regions [Fis81] and superblock regions [HMC+93]. Both are well-known methods to increase the scheduling flexibility, but they come at the cost of inserting compensation code which can adversely affect the WCET. In our experiments, the average WCET obtained with both trace and superblock scheduling varies between 99.7% and 100.3% of the WCET that was achieved using pure basic block scheduling. In addition, the minimum achieved WCET was consistently larger than the basic-block scheduling WCET by up to 15%. Therefore, the additional overhead of creating traces and superblocks does not really pay off in this scheduling scenario. This also follows from the observation that the I/O operations which access the shared bus often have data dependencies to their neighbors and thus can rarely make use of the increased trace and superblock region size.

Due to the relatively small impact of the instruction scheduling compared to the evolutionary schedule optimization (cf. Section 6.1) and the high computational demand of the latter, we tested the combination of both on selected benchmarks only. The instruction scheduler was invoked for every generated individual to also find solutions which are only accessible through a combination of bus and instruction scheduling. Unfortunately, this optimization combination did not yield any further WCET decrease beyond the decrease that is caused by the evolutionary schedule optimization alone. Therefore, the averages given in Figure 6.8a are most likely also an upper bound on the additional WCET decrease that is achievable *after* the schedule itself was optimized. This suggests that optimizing the bus schedule first and performing the instruction scheduling afterwards is sufficient in practice, leading to a combined *average* WCET reduction of more than 30%.

6.3 Summary

This chapter has presented the first WCET-aware multi-core bus schedule optimization which takes into account fair, TDMA and priority-division schedules. The results show that it can reduce the WCET of real-world benchmarks by more than 30% on average, and how ACET, WCET and bus utilization evolve under different parameterizations of the three schedule types. In addition, we have seen that TDMA is *not* always the best choice for minimizing the WCET, which was a basic assumption in previous work. This macroscopic approach was complemented with a new type of instruction scheduling heuristic tailored towards multi-core WCET reduction, which can further reduce the WCET by up to 15.8%. In summary, both optimizations significantly increase the precision of the estimated WCETs and thus the usability of multi-core WCET analysis.

Chapter 7

Conclusion and Future Work

7.1 Summary

This thesis has presented two self-contained, holistic approaches towards the WCET analysis of tasks running on embedded multi-core systems which strongly improve upon previously published work. For both state-permeable and non-state-permeable resources, we have developed new approaches which deliver precise WCET results and an analysis duration which is lower than the one achieved by previously known approaches. Last but not least, we have presented two optimizations which are able to decrease the WCET of tasks running on embedded multi-core systems with shared resources.

We have started with a comparison of different approaches to WCET analysis in Chapter 2. There, we have shown that only the mathematically sound *static* WCET analysis, based on *abstract interpretation*, has the potential to safely capture every possible execution behavior of a program. Therefore, the rest of the thesis builds upon this branch of WCET analysis. The need for an analysis which covers all interacting components of a hardware system was motivated with the definition of *compositionality*, which is hardly ever present in today's computing systems. We also introduced the concept of *timing anomalies* and how it can be used to ease the static WCET analysis. In Chapter 3, we have briefly sketched the infrastructure which was used to implement the analyses in the research compiler WCC.

The most well-known form of a static, single-core WCET analysis pipeline was presented in Chapter 4. We have discussed how call and iteration contexts are disambiguated in WCET analysis, how a value analysis that supports predicated execution can be performed, how to properly model hardware states in WCET analysis using non-deterministic finite-state-machines, and we finally have seen how to perform a path analysis that determines the WCET from runtimes of individual context blocks. The results indicate that the implementation is as precise as the commercial analyzer aiT for non-cached code. For cached code, the WCC implementation is 2.26 times worse than aiT but still 3.56 times better than pessimistic worst-case assumptions, which suffices for our following multi-core comparisons.

In Chapter 5, we have initially identified shared caches and shared buses as the main challenge in the static timing analysis of multi-core systems. Depending on their parameterization we have classified these according to their *state-permeability* and *bounded access delay*. This classification determines whether the analysis of such systems is possible through a *task-partitioned* WCET analysis, or whether it requires a more time-consuming *unified* WCET analysis which considers all parallel tasks together.

For the case of task-partitioned analyses, we have introduced a new abstract microarchitectural domain, called *TDMA offset sets*, for shared, time-triggered buses in Section 5.4, which allows us to employ the single-core analysis framework from Chapter 4 for the WCET analysis of multi-cores. We have shown that the achievable precision for TDMA offset sets strongly depends on the context management. The trivial approach of fully unrolling all loops was identified as being infeasible due to the resulting analysis duration. To overcome this, *cyclic data-flow contexts*, called *offset contexts*, were introduced into the microarchitectural analysis. These are able to outperform the fully unrolling approach in some cases while on average, they are only 18.4% worse and require less than 35.0% of the analysis runtime. Finally, *offset relocation* was proposed as the fastest option, which only needs less than 2.4% of the full unrolling's analysis time but also generates WCET values which are only 3.3% better than a "naive" offset analysis. The full unrolling, being the most precise approach, outperforms the pessimistic worst-case bound by up to 56% on average.

State-permeable and unbounded-delay resources like shared caches and priority-driven arbitration could *not* be analyzed with sufficient precision by the task-partitioned framework. Currently, there is little reason to believe that the precision of this feature combination can be improved at all, since any task-partitioned analysis is forced to use summary-based or worst-case-assumption-based techniques to analyze state-permeable resources. Therefore, the task-partitioned analysis, as presented in Section 5.4, is most suitable for systems with time-triggered bus arbitration and partitioned or locked caches or scratchpads.

Precise WCET results for shared, state-permeable resources can be determined with the help of a *unified, parallelism-aware* WCET analysis as presented in Section 5.5. It explores the possible parallel interleavings of a concurrent, strictly periodic task set at the cycle level. To limit the combinatorial explosion which is incurred by any such exploration, we have introduced a new, timing-based *block exclusion criterion* which can be used to identify invalid execution scenarios and thus to restrict the search space. Our experiments have shown that the parallelism-aware analysis reduces the WCET overestimation by a *factor* of 11.96 when analyzing shared caches compared to the results from Section 5.4. Likewise, the overestimation in the case of fair round-robin arbitration could be decreased by up to 10% and a microarchitectural analysis of fixed-priority-driven arbitration could be performed for the first time. The increase in precision and scope comes at the price of an analysis time which is increased by a factor of 229 compared to the task-partitioned analysis. The block exclusion criterion can be used to limit this increase, since it eliminates 35.6% of the analyzed parallel interleavings on average which translates to a reduction in analysis time by 49.7% and a decrease of the WCET overestimation by up to 10%.

Finally, we have examined in how far the determined multi-core WCETs can be *optimized* in Chapter 6. To this end, an evolutionary optimization of both the schedule type and parameters was devised which searches for configurations which are optimal with respect to ACET, WCET and utilization. Since the schedule must be adapted to the shared resource usage of the benchmark, there can be no generally ideal schedule. Compared with the default configurations, the optimization could achieve an average WCET decrease of more than 30%. Moreover, we devised an instruction scheduling which exploits the fine-grained microarchitectural results from the WCET analysis. In the experiments, this scheduling could additionally decrease the WCET by up to 15.8%.

7.2 Future Work

Task-partitioned WCET Analysis Our treatment of offset contexts in Section 5.4.5 is potentially not optimal if the iterations of a loop can be partitioned into sequential groups with similar loop body behavior inside the group and differing behavior among the groups. In this case, it may be worthwhile to consider using one *layer of offset contexts* for each such group, i.e., one horizontal layer is introduced for each sequential group in Figure 5.6. However, the number of offset contexts will then quickly reach the number of contexts in the full unrolling case. Therefore, sophisticated algorithms would be needed to ensure that this layering is only used when profit can actually be gained. Making the microarchitectural analysis *path-aware* by adding path history information to its domain (cf. Section 2.1) is also one interesting possibility, which could not be explored in this thesis.

Unified WCET Analysis There are multiple opportunities to further improve the scalability and precision of the *unified WCET analysis* as presented in Section 5.5. The most promising idea is to provide synchronization data to the unified WCET analysis to be able to further prune the search space by exploiting rendezvous synchronization behavior. With this extension, the analysis is expected to scale up as long as the tasks to analyze are tightly synchronized. Another important extension would be to deal with non-uniform periods in a more advanced way than proposed in Section 5.5.8.

The unified WCET analysis is also one of the first approaches towards the integration of schedulability and WCET analysis, since it implicitly adheres to the periodic, non-preemptive schedule. Therefore, it can be used to determine whether a task set is schedulable. The synchronization-aware version will in addition allow to determine bounds on the time that a task may wait for a given lock.

Synchronization-aware Path Analysis This work has focused on the difficulties that arise during the *microarchitectural analysis* of multi-core systems. As already pointed out in the beginning of Section 5.4, another major challenge is the integration of synchronization semantics into the *path analysis*. For synchronization statements inside of loops, the IPET approach can no longer be used. Therefore, new approaches are needed to deal with this important case. Data-flow-based path analyses [EGL11; KFM13] are one very promising candidate here.

Multi-core Value Analysis The *value analysis* is also affected by multi-core interference, if communication between concurrently executing tasks is allowed. Since the value analysis is also a data-flow analysis, the approaches that were developed in this thesis are applicable. It is possible to use a summary-based technique similar to the one presented in Section 5.4.1, or the value analysis can be integrated with the microarchitectural analysis. In the latter case, it could also profit from the new, unified WCET analysis and its associated timing-based block exclusion criterion as presented in Section 5.5. A last option is a dedicated parallelism-aware value analysis that works without timing information and possibly on sliced task CFGs.

Extended Benchmark and Hardware Feature Coverage In general, it would be interesting to apply the analyses presented in this thesis to a broader set of benchmarks, possibly including synchronization among the tasks. In particular, some of the average-case drawbacks that were attributed to TDMA in Section 5.4.8 will vanish on benchmarks, where the system is fully utilized all the time. Unfortunately, there are still no standard real-time multi-task or multi-core benchmarks available, and most general purpose benchmark suites use features like dynamic memory allocation, deep software libraries and operating system support which drastically complicate their analysis even if carried out with commercial WCET tools.

To limit the implementation effort, we have not even halfway explored all hardware features that can be modeled using the presented framework. Out-of-order processors, speculative execution, fetch and store buffers and split transactions can all be modeled in both the task-partitioned and the unified WCET analysis. Especially split transactions are interesting, since they would allow for smaller slot sizes which increases the TDMA performance. In addition, they would render the last $T^B_{\max}-1$ cycles of each slot usable which was not the case in our analysis setting used in Chapter 5. The task-partitioned modeling of TDMA offsets could also be applied to statically-allocated time-triggered interconnection networks like Aethereal and TTP and to the FlexRay bus.

Compiler Optimization Opportunities The developed multi-core WCET analyses also open up new possibilities in the domain of compiler optimizations. As already demonstrated for the example of instruction scheduling in Section 6.2, we are now able to assess the effect of code modifications in very high detail. The microarchitectural states determined in the WCET analysis can also be used to guide compiler optimizations like scratchpad and register allocation. The detailed analysis of shared caches now allows a comparison with preexisting cache locking and partitioning methods. Finally, the optimization of the task-to-core mapping becomes feasible, since we can detailedly quantify each mapping's impact on the tasks' WCETs.

List of Figures

List of Tables

List of Algorithms

Glossary

ACET *Average-Case Execution Time.* 37
AI *Abstract Interpretation.* 12
APSS *Abstract Parallel System State.* 126

BCET *Best-Case Execution Time.* 3
BEC *Block Exclusion Criterion.* 125

CDM *Code-Division Multiplexing.* 84
CFG *Control Flow Graph.* 13, 49
CRPD *Cache-Related Preemption Delay.* 30, 87
CSP *Communicating Sequential Processes.* 121

DC-UCB *Definitely Cached Useful Cache Blocks.* 31
DFA *Data-Flow Analysis.* 14
DFS *Depth-First Search.* 58

ECB *Evicting Cache Blocks.* 31
ECU *Electronic Control Units.* 1

FDM *Frequency-Division Multiplexing.* 84
FIFO *First-In First-Out.* 70

HLIR *High-Level Intermediate Representation.* 39

ICN *Interconnection Network.* 83
ILP *Integer Linear Program.* 20, 74
IPCFG *Interprocedural Control Flow Graph.* 52
IPET *Implicit Path Enumeration Technique.* 74
ISA *Instruction Set Architecture.* 43

LLIR *Low-Level Intermediate Representation.* 39
LOC *Lines Of Code.* 40, 209
LRU *Least Recently Used.* 70, 71

MFP *Minimum Fixed Point.* 15
MHP *May-Happen-In-Parallel.* 121
MOP *Meet Over All Paths.* 15
MPI *Message Passing Interface.* 122

PC *Program Counter.* 13
PD *Priority Division.* 86
PEG *Parallel Execution Graph.* 121–123

PLRU *Pseudo-LRU*. 70

RTC *Real-Time Calculus*. 32, 87
RTOS *Real-Time Operating System*. 29

SEP *System Execution Position*. 126

TA *Timing Anomaly*. 25
TDM *Time-Division Multiplexing*. 84, 85
TDMA *Time-Division Multiple Access*. 86
TEP *Task Execution Position*. 126

UCB *Useful Cache Blocks*. 30

WCC *WCET-aware C Compiler*. 7, 20, 37
WCEC *Worst-Case Execution Count*. 39, 74
WCEP *Worst-Case Execution Path*. 37
WCET *Worst-Case Execution Time*. v
WCRT *Worst-Case Response Time*. 30

Bibliography

[AAN11] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. “Precise and Efficient Parametric Path Analysis”. In: *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES ’11. Chicago, IL, USA: ACM, 2011, pp. 141–150. ISBN: 978-1-4503-0555-6. DOI: `10.1145/1967677.1967697`. URL: `http://doi.acm.org/10.1145/1967677.1967697` (Cited on page 21).

[AB09] Sebastian Altmeyer and Claire Burguiere. “A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay”. In: *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*. ECRTS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 109–118. ISBN: 978-0-7695-3724-5. DOI: `10.1109/ECRTS.2009.21`. URL: `http://dx.doi.org/10.1109/ECRTS.2009.21` (Cited on page 31).

[ABD+13] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. “Impact of Resource Sharing on Performance and Performance Prediciton: A Survey”. In: *CONCUR*. 08/2013 (Cited on page 82).

[Abs14a] AbsInt GmbH. *aiT Worst-Case Execution Time Analyzers*. `http://www.absint.com/ait`. 2014 (Cited on pages 3 sq., 7, 37, 41).

[Abs14b] AbsInt GmbH. *Astrée Run Time Error Analyzer*. `http://www.absint.com/astree/index_de.htm`. 2014 (Cited on page 2).

[ACD06] James H. Anderson, John M. Cal, and Umamaheswari C. Devi. “Real-time scheduling on multicore platforms”. In: *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp.* Chapman Hall/CRC, Boca, 2006, pp. 179–190 (Cited on page 34).

[AD90] Rajeev Alur and D. L. Dill. “Automata for Modeling Real-time Systems”. In: *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. Warwick University, England: Springer-Verlag New York, Inc., 1990, pp. 322–335. ISBN: 0-387-52826-1. URL: `http://dl.acm.org/citation.cfm?id=90397.90438` (Cited on page 20).

[AEL+11] Peter Altenbernd, Andreas Ermedahl, Björn Lisper, and Jan Gustafsson. “Automatic Generation of Timing Models for Timing Analysis of High-Level Code”. In: *Proc. 19th International Conference on Real-Time and Network Systems (RTNS2011)*. Ed. by Sébastien Faucou. The IRCCyN lab., 09/2011. URL: `http://www.es.mdh.se/publications/2134-` (Cited on page 22).

[AEL10] Björn Andersson, Arvind Easwaran, and Jinkyu Lee. “Finding an Upper Bound on the Increase in Execution Time Due to Contention on the Memory Bus in COTS-based Multicore Systems”. In: *SIGBED Rev.* 7.1 (01/2010), 4:1–4:4. ISSN: 1551-3688. DOI: `10.1145/1851166.1851172`. URL: `http://doi.acm.org/10.1145/1851166.1851172` (Cited on page 87).

[AEP+08] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. “Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip”. In: *Proceedings of the 21st International Conference on VLSI Design.* VLSID ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 103–110. ISBN: 0-7695-3083-4. DOI: `10.1109/VLSI.2008.33`. URL: `http://dx.doi.org/10.1109/VLSI.2008.33` (Cited on pages 101, 148).

[AGP03] Mathieu Avila, Maxime Glaizot, and Isabelle Puaut. “Impact of Automatic Gain Time Identification on Tree-Based Static WCET Analysis”. In: *WCET.* 2003, pp. 71–74 (Cited on page 74).

[AHL+08] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. “Parametric Timing Analysis for Complex Architectures”. In: *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.* RTCSA ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 367–376. ISBN: 978-0-7695-3349-0. DOI: `10.1109/RTCSA.2008.7`. URL: `http://dx.doi.org/10.1109/RTCSA.2008.7` (Cited on page 21).

[ALS+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd edition. Pearson Education, 2007. ISBN: 0321486811 (Cited on pages 12, 15, 39, 58, 155).

[Alt00] Altera / Eureka Technology, Inc. *PCI Bus Arbiter.* `http://www.altera.com/products/ip/iup/pci/m-eur-pci-bus-arb.html`. 2000 (Cited on page 85).

[AMR10] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. “Resilience Analysis: Tightening the CRPD Bound for Set-associative Caches”. In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems.* LCTES ’10. Stockholm, Sweden: ACM, 2010, pp. 153–162. ISBN: 978-1-60558-953-4. DOI: `10.1145/1755888.1755911`. URL: `http://doi.acm.org/10.1145/1755888.1755911` (Cited on page 31).

[ARM04] ARM Ltd. *ARM7TDMI Technical Reference Manual.* Revision: r4p1. ARM DDI 0210C. 11/2004 (Cited on pages 44, 67).

[ARM05] ARM Ltd. *ARM Architecture Reference Manual.* I. ARM DDI 0100I. 110 Fulbourn Road Cambridge, England CB1 9NJ, 07/2005 (Cited on pages 43, 48, 51, 53).

[ARM14a] ARM Ltd. *ARM Cortex-R Series.* `http://www.arm.com/products/processors/cortex-r/index.php`. 2014 (Cited on page 23).

[ARM14b] ARM Ltd. *ARM Processor Families.* `http://www.arm.com/products/processors/classic/arm7/index.php`. 2014 (Cited on page 6).

[AUT09] AUTOSAR Administration. *Specification of Multi-Core OS Architecture V1.0.0.* R4.0 Rev 1. 2009 (Cited on pages 29, 33).

[Bay08] Nimrod Bayer. *US Patent 60986659: Shared Mmemory System for a Tightly-Coupled Multiprocessor.* 11/2008 (Cited on page 84).

[BB06] Adam Betts and Guillem Bernat. “Tree-Based WCET Analysis on Instrumentation Point Graphs”. In: *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. ISORC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 558–565. ISBN: 0-7695-2561-X. DOI: `10.1109/ISORC.2006.75`. URL: `http://dx.doi.org/10.1109/ISORC.2006.75` (Cited on page 74).

[BC08] Clément Ballabriga and Hugues Casse. “Improving the First-Miss Computation in Set-Associative Instruction Caches”. In: *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. ECRTS ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 341–350. ISBN: 978-0-7695-3298-1. DOI: `10.1109/ECRTS.2008.34`. URL: `http://dx.doi.org/10.1109/ECRTS.2008.34` (Cited on page 70).

[BC11] Jean-Luc Béchennec and Franck Cassez. *Computation of WCET using Program Slicing and Real-Time Model-Checking*. Research Report. IRCCyN/C-NRS, 05/2011 (Cited on page 20).

[BCM09] Clément Ballabriga, Hugues Cassé, and Marianne De Michiel. “A Generic Framework for Blackbox Components in WCET Computation”. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET’09)*. Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2009, pp. 1–12. ISBN: 978-3-939897-14-9. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2009.2290`. URL: `http://drops.dagstuhl.de/opus/volltexte/2009/2290` (Cited on page 28).

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. “An Efficient Algorithm for Parametric WCET Calculation”. In: *Journal of Systems Architecture* 57.6 (06/2011), pp. 614–624. ISSN: 1383-7621. DOI: `10.1016/j.sysarc.2010.06.009`. URL: `http://dx.doi.org/10.1016/j.sysarc.2010.06.009` (Cited on page 21).

[BHQ+07] A.O. Balkan, M.N. Horak, Gang Qu, and Uzi Vishkin. “Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing”. In: *15th Annual IEEE Symposium on High-Performance Interconnects, 2007*. HOTI 2007. 2007/08///2007, pp. 21–28. DOI: `10.1109/HOTI.2007.11` (Cited on page 84).

[BHV11] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. “Refinement-based CFG Reconstruction from Unstructured Programs”. In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI’11. Austin, TX, USA: Springer-Verlag, 2011, pp. 54–69. ISBN: 978-3-642-18274-7. URL: `http://dl.acm.org/citation.cfm?id=1946284.1946290` (Cited on page 49).

[BJM11] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. “Multi-Core Cache Hierarchies”. In: *Synthesis Lectures on Computer Architecture*. Ed. by University of Wisconsin Mark D. Hill. Vol. Lecture #17. Morgan & Claypool, 2011. ISBN: 9781598297546. DOI: `10.2200/S00365ED1V01Y201105CAC017` (Cited on page 82).

[BKJ+01] Jr. Bell R.H., Chang Yong Kang, L. John, and E.E. Swartzlander. "CDMA as a multiprocessor interconnect strategy". In: *Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers, 2001.* Vol. 2. 11/2001, 1246–1250 vol.2. DOI: `10.1109/ACSSC.2001.987690` (Cited on page 84).

[BL08] Stefan Bygde and Björn Lisper. "Towards an Automatic Parametric WCET Analysis". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08).* Ed. by Raimund Kirner. Vol. 8. OpenAccess Series in Informatics (OASIcs). also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2008. ISBN: 978-3-939897-10-1. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2008.1659`. URL: `http://drops.dagstuhl.de/opus/volltexte/2008/1659` (Cited on page 21).

[Bli02] Johann Blieberger. "Data-Flow Frameworks for Worst-Case Execution Time Analysis". In: *Real-Time Systems* 22.3 (05/2002), pp. 183–227. ISSN: 0922-6443. DOI: `10.1023/A:1014535317056`. URL: `http://dx.doi.org/10.1023/A:1014535317056` (Cited on page 21).

[BLL+11] Dai Bui, Edward Lee, Isaac Liu, Hiren Patel, and Jan Reineke. "Temporal Isolation on Multiprocessing Architectures". In: *Proceedings of the 48th Design Automation Conference.* DAC '11. San Diego, California: ACM, 2011, pp. 274–279. ISBN: 978-1-4503-0636-2. DOI: `10.1145/2024724.2024787`. URL: `http://doi.acm.org/10.1145/2024724.2024787` (Cited on page 89).

[BLT+03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. "PISA: A Platform and Programming Language Independent Interface for Search Algorithms". In: *Proceedings of the 2nd International Conference on Evolutionary Multi-criterion Optimization.* EMO'03. Faro, Portugal: Springer-Verlag, 2003, pp. 494–508. ISBN: 3-540-01869-7. URL: `http://dl.acm.org/citation.cfm?id=1760102.1760144` (Cited on page 150).

[BM11] Balasubramanya Bhat and Frank Mueller. "Making DRAM Refresh Predictable". In: *Real-Time Systems* 47.5 (09/2011), pp. 430–453. ISSN: 0922-6443. DOI: `10.1007/s11241-011-9129-6`. URL: `http://dx.doi.org/10.1007/s11241-011-9129-6` (Cited on pages 23, 45, 82).

[BMV12] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. "A Time-composable Operating System". In: *WCET.* 2012, pp. 69–80 (Cited on page 30).

[Bor13] Hendrik Borghorst. "Schedulingverfahren zur WCET-Reduktion in eingebetteten Multicore-Systemen". Master's Thesis. TU Dortmund, 2013 (Cited on page 10).

[Bor96] Hans Borjesson. "Incorporating Worst Case Execution Time in a Commercial C-compiler". Undergraduate Thesis. Department of Computer Systems, Uppsala University, 1996 (Cited on page 38).

[BRA09] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. "Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions." In: *WCET.* 2009 (Cited on pages 31, 70).

[BSI+08] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, and Aarti Gupta. "SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement". English. In: *Static Analysis*. Ed. by María Alpuente and Germán Vidal. Vol. 5079. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 238–254. ISBN: 978-3-540-69163-1. URL: `http://dx.doi.org/10.1007/978-3-540-69166-2%5C_16` (Cited on page 18).

[Buc00] William Buchanan. *Computer Busses*. Newton, MA, USA: Butterworth-Heinemann, 2000. ISBN: 0340740760 (Cited on page 85).

[BW13] David Blaza and Alex Wolfe. *UBM Tech 2013 Embedded Market Study*. Design West. San Jose, CA, 2013 (Cited on page 2).

[BY04] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". English. In: *Lectures on Concurrency and Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 87–124. ISBN: 978-3-540-22261-3. DOI: `10.1007/978-3-540-27755-2\_3`. URL: `http://dx.doi.org/10.1007/978-3-540-27755-2%5C_3` (Cited on page 20).

[BZT+11] Sven Bünte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. "Improving the Confidence in Measurement-Based Timing Analysis". In: *ISORC*. 2011, pp. 144–151 (Cited on page 22).

[CBR13] Sudipta Chattopadhyay, Abhijeet Banerjee, and Abhik Roychoudhury. "Precise Micro-architectural Modeling for WCET Analysis via AI+SAT". In: *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. RTAS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 87–96. ISBN: 978-1-4799-0186-9. DOI: `10.1109/RTAS.2013.6531082`. URL: `http://dx.doi.org/10.1109/RTAS.2013.6531082` (Cited on page 74).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: `10.1145/512950.512973`. URL: `http://doi.acm.org/10.1145/512950.512973` (Cited on page 12).

[CC80] Patrick Cousot and Radhia Cousot. "Semantic analysis of communicating sequential processes". English. In: *Automata, Languages and Programming*. Ed. by Jaco de Bakker and Jan van Leeuwen. Vol. 85. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1980, pp. 119–133. ISBN: 978-3-540-10003-4. URL: `http://dx.doi.org/10.1007/3-540-10003-2%5C_65` (Cited on page 121).

[CC84] Patrick Cousot and Radhia Cousot. "Invariance Proof Methods and Analysis Techniques For Parallel Programs". In: *Automatic Program Construction Techniques*. Ed. by A.W. Biermann, G. Guiho, and Y. Kodratoff. Macmillan, New York, United States, 1984. Chap. 12, pp. 243–271 (Cited on page 121).

[CCK+13] Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and H. Falk. "Real-time partitioned scheduling on multi-core systems with local and global memories". In: *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. 01/2013, pp. 467–472. DOI: `10.1109/ASPDAC.2013.6509640` (Cited on page 34).

[CCM97] Sérgio Campos, Edmund Clarke, and Marius Minea. "The verus tool: A quantitative approach to the formal verification of real-time systems". English. In: *Computer Aided Verification*. Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 452–455. ISBN: 978-3-540-63166-8. DOI: `10.1007/3-540-63166-6\_46`. URL: `http://dx.doi.org/10.1007/3-540-63166-6%5C_46` (Cited on page 20).

[CCR+14] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. "A Unified WCET Analysis Framework for Multicore Platforms". In: *ACM Transactions on Embedded Computing Systems* 13.4s (04/2014), 124:1–124:29. ISSN: 1539-9087. DOI: `10.1145/2584654`. URL: `http://doi.acm.org/10.1145/2584654` (Cited on page 87).

[CEN+13] Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel. "Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs". In: *Proceedings of the Sixth International Workshop on Multi-/Many-core Computing Systems (MuCoCoS 2013)*. MuCoCoS 2013. Edinburgh, Scotland, UK, 09/2013 (Cited on page 33).

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. "Predictability Considerations in the Design of Multi-Core Embedded Systems". In: *Ingénieurs de l'Automobile* 807 (09/2010), pp. 36–42. ISSN: 0020-1200 (Cited on pages 45, 89).

[CI92] Jyh-Herng Chow and Williams Ludwell Harrison III. "A General Framework for Analyzing Shared-Memory Parallel Programs." In: *ICPP (2)*. 1992, pp. 192–199 (Cited on page 122).

[CKR+12] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Heiko Falk, and Peter Marwedel. "A Unified WCET Analysis Framework for Multi-Core Platforms". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Beijing, China, 04/2012, pp. 99–108 (Cited on pages 33, 88).

[CM07] Christoph Cullmann and Florian Martin. "Data-Flow Based Detection of Loop Bounds". In: *WCET*. 2007 (Cited on page 74).

[CMR+05] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. "Exploiting branch constraints without exhaustive path enumeration". In: *In 5th International Workshop on Worst-Case Execution Time Analysis (WCET*. 2005 (Cited on page 74).

[CN98] Marek Chrobak and John Noga. "LRU is Better Than FIFO". In: *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '98. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1998, pp. 78–81. ISBN: 0-89871-410-9. URL: `http://dl.acm.org/citation.cfm?id=314613.314655` (Cited on page 70).

[Cor10] IBM Corporation. *User's Manual for CPLEX 12.2.* 2010 (Cited on page 1).

[Cou01] Patrick Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges". In: *Informatics - 10 Years Back. 10 Years Ahead.* London, UK, UK: Springer-Verlag, 2001, pp. 138–156. ISBN: 3-540-41635-8. URL: `http://dl.acm.org/citation.cfm?id=647348.724445` (Cited on pages 16, 63).

[CP01] Antoine Colin and Isabelle Puaut. "A Modular & Retargetable Framework for Tree-Based WCET Analysis". In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems.* ECRTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 37–. URL: `http://dl.acm.org/citation.cfm?id=871910.871918` (Cited on page 74).

[CQV+13] Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. "PROARTIS: Probabilistically Analyzable Real-Time Systems". In: *ACM Trans. Embed. Comput. Syst.* 12.2s (05/2013), 94:1–94:26. ISSN: 1539-9087. DOI: `10.1145/2465787.2465796`. URL: `http://doi.acm.org/10.1145/2465787.2465796` (Cited on page 22).

[CR09] Sudipta Chattopadhyay and Abhik Roychoudhury. "Unified Cache Modeling for WCET Analysis and Layout Optimizations". In: *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium.* RTSS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 47–56. ISBN: 978-0-7695-3875-4. DOI: `10.1109/RTSS.2009.20`. URL: `http://dx.doi.org/10.1109/RTSS.2009.20` (Cited on page 70).

[CRM10] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. "Modeling Shared Cache and Bus in Multi-cores for Timing Analysis". In: *Proceedings of the 13th International Workshop on Software &#38; Compilers for Embedded Systems.* SCOPES '10. St. Goar, Germany: ACM, 2010, 6:1–6:10. ISBN: 978-1-4503-0084-1. DOI: `10.1145/1811212.1811220`. URL: `http://doi.acm.org/10.1145/1811212.1811220` (Cited on pages 10, 88, 105).

[CSB+10] H. Cassé, P. Sainrat, C. Ballabriga, and M. de Michiel. "Experimentation of WCET Computation on Both Ends of Automotive Processor Range". In: *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness &#38; Safety.* CARS '10. Valencia, Spain: ACM, 2010, pp. 67–70. ISBN: 978-1-60558-915-2. DOI: `10.1145/1772643.1772663`. URL: `http://doi.acm.org/10.1145/1772643.1772663` (Cited on page 23).

[CT94] Chia-Mei Chen and Satish K. Tripathi. *Multiprocessor Priority Ceiling Based Protocols.* Tech. rep. College Park, MD, USA, 1994 (Cited on page 33).

[CVJ+08] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. "Dataflow Analysis for Concurrent Programs Using Datarace Detection". In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 316–326. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375620`. URL: `http://doi.acm.org/10.1145/1375581.1375620` (Cited on page 122).

[DAN+11] Dakshina Dasari, Bjorn Andersson, Vincent Nelis, Stefan M. Petters, Arvind Easwaran, and Jinkyu Lee. "Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus". In: *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications.* TRUSTCOM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1075. ISBN: 978-0-7695-4600-1. DOI: 10.1109/TrustCom.2011.146. URL: http://dx.doi.org/10.1109/TrustCom.2011.146 (Cited on page 87).

[DBB+07] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised". In: *Real-Time Systems* 35.3 (04/2007), pp. 239–272. ISSN: 0922-6443. DOI: 10.1007/s11241-007-9012-7. URL: http://dx.doi.org/10.1007/s11241-007-9012-7 (Cited on page 35).

[DDY06] Dinakar Dhurjati, Manuvir Das, and Yue Yang. "Path-Sensitive Dataflow Analysis with Iterative Refinement". In: *Proceedings of the 13th International Conference on Static Analysis.* SAS'06. Seoul, Korea: Springer-Verlag, 2006, pp. 425–442. ISBN: 3-540-37756-5, 978-3-540-37756-6. DOI: 10.1007/11823230_27. URL: http://dx.doi.org/10.1007/11823230%5C_27 (Cited on page 18).

[DMS08] Damian Dechev, Rabi N. Mahapatra, and Bjarne Stroustrup. "Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems." In: 2008, pp. 375–393 (Cited on page 38).

[DZ10] Yiqiang Ding and Wei Zhang. "Improving the Static Real-time Scheduling on Multicore Processors by Reducing Worst-case Inter-thread Cache Interferences". In: *Proceedings of the 48th Annual Southeast Regional Conference.* ACM SE '10. Oxford, Mississippi: ACM, 2010, 108:1–108:4. ISBN: 978-1-4503-0064-3. DOI: 10.1145/1900008.1900148. URL: http://doi.acm.org/10.1145/1900008.1900148 (Cited on page 155).

[EBS+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture.* ISCA '11. San Jose, California, USA: ACM, 2011, pp. 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108. URL: http://doi.acm.org/10.1145/2000064.2000108 (Cited on page 6).

[EGL11] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. "Deriving WCET bounds by abstract execution". In: *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011:)* 2011 (Cited on pages 74, 166).

[Ele12] ElektronikPraxis. "ElektronikPraxis". In: (02/2012), p. 15 (Cited on pages 2, 44).

[EPB+06] J. Eisinger, I Polian, B. Becker, and A Metzner. "Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis". In: *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE.* 04/2006, pp. 13–18. DOI: 10.1109/DDECS.2006.1649563 (Cited on page 26).

[ES08] Michael Engel and Olaf Spinczyk. "System-on-chip Integration of Embedded Automotive Controllers". In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. Glasgow, Scotland: ACM, 2008, pp. 29–34. ISBN: 978-1-60558-126-2. DOI: 10.1145/1435458.1435464. URL: http://doi.acm.org/10.1145/1435458.1435464 (Cited on page 1).

[Evi14] Evidence. *ERIKA Enterprise - Open Source RTOS OSEK/VDX Kernel*. http://erika.tuxfamily.org/drupal. 2014 (Cited on page 40).

[FDG+09] Alberto Ferrari, Marco Di Natale, Giacomo Gentile, Giovanni Reggiani, and Paolo Gai. "Time and Memory Tradeoffs in the Implementation of AUTOSAR Components". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 864–869. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.org/citation.cfm?id=1874620.1874830 (Cited on page 29).

[FH04] Christian Ferdinand and Reinhold Heckmann. "aiT: Worst-Case Execution Time Prediction by Static Program Analysis". English. In: *Building the Information Society*. Ed. by Renè Jacquart. Vol. 156. IFIP International Federation for Information Processing. Springer US, 2004, pp. 377–383. ISBN: 978-1-4020-8156-9. DOI: 10.1007/978-1-4020-8157-6_29. URL: http://dx.doi.org/10.1007/978-1-4020-8157-6%5C_29 (Cited on page 20).

[Fis81] J. A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". In: *IEEE Trans. Comput.* 30.7 (07/1981), pp. 478–490. ISSN: 0018-9340. DOI: 10.1109/TC.1981.1675827. URL: http://dx.doi.org/10.1109/TC.1981.1675827 (Cited on pages 155, 160).

[FKP+07] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. "Task automata: Schedulability, decidability and undecidability". In: *Information and Computation* 205.8 (2007), pp. 1149–1172 (Cited on page 20).

[FL10] Heiko Falk and Paul Lokuciejewski. "A compiler framework for the reduction of worst-case execution times". In: *Journal on Real-Time Systems* 46.2 (10/2010). DOI 10.1007/s11241-010-9101-x, pp. 251–300 (Cited on pages 9, 20, 37, 39).

[FLT06] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. "Design of a WCET-Aware C Compiler". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Dresden/Germany, 07/2006 (Cited on page 37).

[Fre09] Freescale Semiconductor, Inc. *Embedded Multicore: An Introduction, Rev. 0*. www.freescale.com. Document Number: EMBMCRM. 2009 (Cited on pages 5, 32, 45).

[FW86] Philip J. Fleming and John J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Commun. ACM* 29.3 (03/1986), pp. 218–221. ISSN: 0001-0782. DOI: 10.1145/5666.5673. URL: http://doi.acm.org/10.1145/5666.5673 (Cited on page 78).

[FW99] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-TimeSystems". In: *Real-Time Systems* 17.2-3 (12/1999), pp. 131–181. ISSN: 0922-6443. DOI: 10.1023/A:1008186323068. URL: http://dx.doi.org/10.1023/A:1008186323068 (Cited on page 70).

[GAE+09] Jan Gustafsson, Peter Altenbernd, Andreas Ermedahl, and Björn Lisper. "Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development". In: *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*. SEUS '09. Newport Beach, CA: Springer-Verlag, 2009, pp. 308–319. ISBN: 978-3-642-10264-6. DOI: `10.1007/978-3-642-10265-3\_28`. URL: `http://dx.doi.org/10.1007/978-3-642-10265-3%5C_28` (Cited on page 22).

[GE07] Jan Gustafsson and Andreas Ermedahl. "Experiences from Applying WCET Analysis in Industrial Settings". In: *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. ISORC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 382–392. ISBN: 0-7695-2765-5. DOI: `10.1109/ISORC.2007.36`. URL: `http://dx.doi.org/10.1109/ISORC.2007.36` (Cited on pages 21, 23 sq.).

[Geb10] Gernot Gebhard. "Timing Anomalies Reloaded". In: *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Ed. by Björn Lisper. Austrian Computer Society, 07/2010, pp. 5–15 (Cited on page 26).

[GEL+10] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. "Towards WCET Analysis of Multicore Architectures Using UPPAAL". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2010, pp. 101–112. ISBN: 978-3-939897-21-7. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2010.101`. URL: `http://drops.dagstuhl.de/opus/volltexte/2010/2830` (Cited on pages 20, 33, 88).

[GGL12] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. "Toward Static Timing Analysis of Parallel Software". In: *12th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012, pp. 38–47. ISBN: 978-3-939897-41-5. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2012.38`. URL: `http://drops.dagstuhl.de/opus/volltexte/2012/3555` (Cited on page 88).

[GH10] Kees Goossens and Andreas Hansson. "The Aethereal Network on Chip After Ten Years: Goals, Evolution, Lessons, and Future". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 306–311. ISBN: 978-1-4503-0002-5. DOI: `10.1145/1837274.1837353`. URL: `http://doi.acm.org/10.1145/1837274.1837353` (Cited on page 86).

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (Cited on page 1).

[GHK+11] Peter Gliwa, Jens Harnisch, Ursula Kelling, and Christoph Ficek. "From Single-Core to Multi-Core Platforms - Systematic Migration of Hard Real-Time Software in AUTOSAR". In: *Embedded World 28*. 2011, pp. 979–992 (Cited on page 33).

[GKS+11] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. "Formal Analysis of MPI-based Parallel Programs". In: *Commun. ACM* 54.12 (12/2011), pp. 82–91. ISSN: 0001-0782. DOI: `10.1145/2043174.2043194`. URL: `http://doi.acm.org/10.1145/2043174.2043194` (Cited on page 122).

[GLM11] Antonio González, Fernando Latorre, and Grigorios Magklis. "Processor Microarchitecture: An Implementation Perspective". In: *Synthesis Lectures on Computer Architecture*. Ed. by University of Wisconsin Mark D. Hill. Vol. Lecture #12. Morgan & Claypool, 2011. ISBN: 9781608454532. DOI: `10.2200/S00309ED1V01Y201011CAC012` (Cited on page 63).

[GR10] Daniel Grund and Jan Reineke. "Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection". In: *Proceedings of the 2010 22Nd Euromicro Conference on Real-Time Systems*. ECRTS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 155–164. ISBN: 978-0-7695-4111-2. DOI: `10.1109/ECRTS.2010.8`. URL: `http://dx.doi.org/10.1109/ECRTS.2010.8` (Cited on page 70).

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: `10.1109/WWC.2001.15`. URL: `http://dx.doi.org/10.1109/WWC.2001.15` (Cited on pages 76, 150).

[Gro08] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Newton, MA, USA: Newnes, 2008. ISBN: 075068397X, 9780750683975 (Cited on page 44).

[GS93] Dirk Grunwald and Harini Srinivasan. "Data Flow Equations for Explicitly Parallel Programs". In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '93. San Diego, California, USA: ACM, 1993, pp. 159–168. ISBN: 0-89791-589-5. DOI: `10.1145/155332.155349`. URL: `http://doi.acm.org/10.1145/155332.155349` (Cited on page 122).

[Gün13] Christian Günter. "Unterstützung modularer WCET-Analyse durch annotierte Binärobjekte". Bachelor's Thesis. TU Dortmund, 2013 (Cited on pages 9, 48).

[GZ10] Satya Mohan Raju Gudidevuni and Wei Zhang. "A Time-predictable Dual-core Prototype on FPGA". In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE '10. Oxford, Mississippi: ACM, 2010, 7:1–7:4. ISBN: 978-1-4503-0064-3. DOI: `10.1145/1900008.1900020`. URL: `http://doi.acm.org/10.1145/1900008.1900020` (Cited on page 90).

[Har13] Tim Harde. "Vergleichende Studie von Arbitrierungsverfahren für Kommunikationsstrukturen in eingebetteten Multicoresystemen". Bachelor's Thesis. TU Dortmund, 2013 (Cited on page 9).

[HBH+11] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. "CAMA: A Predictable Cache-Aware Memory Allocator". In: *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on.* 07/2011, pp. 23–32. DOI: `10.1109/ECRTS.2011.11` (Cited on page 71).

[HE05] Arne Hamann and Rolf Ernst. "TDMA Time Slot and Turn Optimization with Evolutionary Search Techniques". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1.* DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 312–317. ISBN: 0-7695-2288-2. DOI: `10.1109/DATE.2005.299`. URL: `http://dx.doi.org/10.1109/DATE.2005.299` (Cited on pages 148 sq.).

[HG11] Andreas Hansson and Kees Goossens. *On-Chip Interconnect with aelite: Composable and Predictable Systems.* Embedded Systems. Springer New York, 2011. ISBN: 978-1-4419-6496-0 (Cited on page 86).

[HG12] Sebastian Hahn and Daniel Grund. "Relational Cache Analysis for Static Timing Analysis". In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS '12).* Pisa, Italy, 07/2012, pp. 102–111. ISBN: 978-1-4673-2032-0. DOI: `10.1109/ECRTS.2012.14` (Cited on page 71).

[HGB+08] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. " WCET 2008 – Report from the Tool Challenge 2008 – 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08).* Ed. by Raimund Kirner. Vol. 8. OpenAccess Series in Informatics (OASIcs). Prague / Czech Republic: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 09/2008. ISBN: 978-3-939897-10-1 (Cited on page 24).

[HGG+01] González Harbour, Gutiérrez García, Palencia Gutiérrez, and Drake Moyano. "Mast: Modeling and analysis suite for real time applications". In: *13th Euromicro Conference on Real-Time Systems.* IEEE. 2001, pp. 125–134 (Cited on page 32).

[HHJ+05] R. Henia, A Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. "System level performance analysis - the SymTA/S approach". In: *IEEE Proceedings on Computers and Digital Techniques* 152.2 (03/2005), pp. 148–166. ISSN: 1350-2387. DOI: `10.1049/ip-cdt:20045088` (Cited on page 32).

[HKB+14] Chen-Wei Huang, Timon Kelter, Bjoern Boenninghoff, Jan Kleinsorge, Michael Engel, Peter Marwedel, and Shiao-Li Tsao. "Static WCET Analysis of the H.264/AVC Decoder Exploiting Coding Information". In: *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* IEEE. Chongqing, China, 08/2014 (Cited on pages 21, 24).

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation". In: *J. Supercomput.* 7.1-2 (05/1993), pp. 229–248. ISSN: 0920-8542. DOI: `10.1007/BF01205185`. URL: `http://dx.doi.org/10.1007/BF01205185` (Cited on page 160).

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: A Path Sensitive Static Analyser". In: *Electron. Notes Theor. Comput. Sci.* 289 (12/2012), pp. 15–25. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2012.11.003`. URL: `http://dx.doi.org/10.1016/j.entcs.2012.11.003` (Cited on pages 18, 53).

[Höf12] Kai Höfig. "Failure-Dependent Timing Analysis - A New Methodology for Probabilistic Worst-Case Execution Time Analysis". English. In: *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*. Ed. by JensB. Schmitt. Vol. 7201. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 61–75. ISBN: 978-3-642-28539-4. DOI: `10.1007/978-3-642-28540-0\_5`. URL: `http://dx.doi.org/10.1007/978-3-642-28540-0%5C_5` (Cited on page 22).

[HP08] Damien Hardy and Isabelle Puaut. "WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches". In: *Proceedings of the 2008 Real-Time Systems Symposium*. RTSS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 456–466. ISBN: 978-0-7695-3477-0. DOI: `10.1109/RTSS.2008.10`. URL: `http://dx.doi.org/10.1109/RTSS.2008.10` (Cited on page 73).

[HP09] Damien Hardy and Isabelle Puaut. "Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches". In: *17th International Conference on Real-Time and Network Systems*. Ed. by Laurent George and Maryline Chetto andMikael Sjodin. Paris, France, 2009, pp. 45–54. URL: `http://hal.inria.fr/inria-00441959` (Cited on page 87).

[HPP11] Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. "Towards an open timing analysis platform". In: *11th International Workshop on Worst-Case Execution Time Analysis*. 07/2011 (Cited on pages 20, 38).

[HPP12] Benedikt Huber, Daniel Prokesch, and Peter Puschner. "A Formal Framework for Precise Parametric WCET Formulas". In: *12th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012, pp. 91–102. ISBN: 978-3-939897-41-5. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2012.91`. URL: `http://drops.dagstuhl.de/opus/volltexte/2012/3560` (Cited on page 21).

[HPS12] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. "Worst-case Execution Time Analysis-driven Object Cache Design". In: *Concurrency and Computation: Practice & Experience* 24.8 (06/2012), pp. 753–771. ISSN: 1532-0626. DOI: `10.1002/cpe.1763`. URL: `http://dx.doi.org/10.1002/cpe.1763` (Cited on page 23).

[HRW13] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. "Towards Compositionality in Execution Time Analysis – Definition and Challenges". In: *CRTS*. 12/2013 (Cited on pages 27 sq.).

[HS09] Benedikt Huber and Martin Schoeberl. "Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis". In: *WCET*. 2009 (Cited on page 20).

[HT07] Wolfgang Haid and Lothar Thiele. "Complex Task Activation Schemes in System Level Performance Analysis". In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '07. Salzburg, Austria: ACM, 2007, pp. 173–178. ISBN: 978-1-59593-824-4. DOI: 10.1145/1289816.1289860. URL: http://doi.acm.org/10.1145/1289816.1289860 (Cited on page 34).

[HT98] Maria Handjieva and Stanislav Tzolovski. "Refining Static Analyses by Trace-Based Partitioning Using Control Flow". English. In: *Static Analysis*. Ed. by Giorgio Levi. Vol. 1503. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 200–214. ISBN: 978-3-540-65014-0. URL: http://dx.doi.org/10.1007/3-540-49727-7%5C_12 (Cited on page 17).

[HZX12] Yazhi Huang, Mengying Zhao, and Chun Jason Xue. "WCET-aware Re-Scheduling Register Allocation for Real-Time Embedded Systems with Clustered VLIW Architecture". In: *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. 2012 (Cited on page 155).

[Inf08] Infineon Technologies AG. *TriCore 1, Instruction Set V1.3 & V1.3.1 Architecture*. User's Manual V1.3.8. 01/2008 (Cited on page 43).

[Inf09] Infineon Technologies AG. *TC1767 32-Bit Single-Chip Microcontroller*. User's Manual V1.1. 05/2009 (Cited on pages 44 sqq.).

[Inf14] Infineon Technologies AG. *AURIX microcontroller*. 2014 (Cited on page 23).

[Jac09] Bruce Jacob. "The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It". In: *Synthesis Lectures on Computer Architecture*. Ed. by University of Wisconsin Mark D. Hill. Vol. Lecture #7. Morgan & Claypool, 2009. ISBN: 9781598295887. DOI: 10.2200/S00201ED1V01Y200907CAC007 (Cited on page 82).

[JMR10] J. Schneider, M. Bohn, and R. Rößger. "Migration of Automotive Real-Time Software to Multicore Systems: First Steps towards an Automated Solution". In: *Proceedings Work-In-Progress Session of the 22th Euromicro Conference on Real-Time Systems*. ECRTS'10. Brussels, Belgium, 07/2010, pp. 37–40 (Cited on page 33).

[Joh14] R. Colin Johnson. "IBM Puts Brain On-a-Chip". In: *EE Times* (08/2014) (Cited on page 5).

[JP86] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29.5 (1986), pp. 390–395. DOI: 10.1093/comjnl/29.5.390. eprint: http://comjnl.oxfordjournals.org/content/29/5/390.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/29/5/390.abstract (Cited on page 30).

[KAL14] KALRAY Corporation. *MPPA 256 - Many-core Processors*. http://www.kalray.eu. 2014 (Cited on page 23).

[KB03] Hermann Kopetz and G. Bauer. "The time-triggered architecture". In: *Proceedings of the IEEE* 91.1 (01/2003), pp. 112–126. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805821 (Cited on pages 34, 85).

[KFM+11] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. "Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds". In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*. Porto, Portugal, 07/2011, pp. 3–12 (Cited on pages 10, 88, 96, 101, 104, 113).

[KFM+14] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. "Static Analysis of Multi-Core TDMA Resource Arbitration Delays". English. In: *Real-Time Systems* 50.2 (03/2014), pp. 185–229. ISSN: 0922-6443. DOI: 10.1007/s11241-013-9189-x. URL: http://dx.doi.org/10.1007/s11241-013-9189-x (Cited on pages 10, 33, 87 sq., 96, 104 sq., 113).

[KFM11] Jan C. Kleinsorge, Heiko Falk, and Peter Marwedel. "A Synergetic Approach To Accurate Analysis Of Cache-Related Preemption Delay". In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*. Taipei, Taiwan, 10/2011, pp. 329–338 (Cited on page 31).

[KFM13] Jan Kleinsorge, Heiko Falk, and Peter Marwedel. "Simple Analysis of Partial Worst-case Execution Paths on General Control Flow Graphs". In: *Proceedings of the International Conference on Embedded Software (EMSOFT 2013)*. EMSOFT 2013. Montreal, Canada, 10/2013 (Cited on pages 74, 130, 166).

[KHM+13] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. "Evaluation of Resource Arbitration Methods for Multi-Core Real-Time Systems". In: *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Claire Maiza. Paris, France, 07/2013 (Cited on pages 10, 96, 113, 119).

[Kil73] Gary A. Kildall. "A Unified Approach to Global Program Optimization". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945 (Cited on page 12).

[Kir12] Raimund Kirner. "The WCET Analysis Tool Calcwcet167". In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Volume Part II*. ISoLA'12. Heraklion, Crete, Greece: Springer-Verlag, 2012, pp. 158–172. ISBN: 978-3-642-34031-4. DOI: 10.1007/978-3-642-34032-1_17. URL: http://dx.doi.org/10.1007/978-3-642-34032-1%5C_17 (Cited on pages 20, 38).

[KKP+11] 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 and Systems Modeling* 10.3 (07/2011), pp. 411–437. ISSN: 1619-1366. DOI: 10.1007/s10270-010-0161-0. URL: http://dx.doi.org/10.1007/s10270-010-0161-0 (Cited on pages 19, 42, 76).

[KKP09] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. "Precise Worst-Case Execution Time Analysis for Processors with Timing Anomalies". In: *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*. ECRTS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 119–128. ISBN: 978-0-7695-3724-5. DOI: 10.1109/ECRTS.2009.8. URL: http://dx.doi.org/10.1109/ECRTS.2009.8 (Cited on page 26).

[KM14] Timon Kelter and Peter Marwedel. "Parallelism Analysis: Precise WCET Values for Complex Multi-Core Systems". In: *Third International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)*. Ed. by Cyrille Artho and Peter Ölveczky. Luxembourg: Springer, 11/2014 (Cited on pages 10, 120).

[KMB14] Timon Kelter, Peter Marwedel, and Hendrik Borghorst. "WCET-aware Scheduling Optimizations for Multi-Core Real-Time Systems". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, Greece, 07/2014 (Cited on pages 10, 147).

[KP01] R. Kirner and P. Puschner. "Transformation of path information for WCET analysis during compilation". In: *Real-Time Systems, 13th Euromicro Conference on, 2001.* 2001, pp. 29–36. DOI: 10.1109/EMRTS.2001.933993 (Cited on page 38).

[KPP10] Raimund Kirner, Peter Puschner, and Adrian Prantl. "Transforming Flow Information During Code Optimization for Timing Analysis". In: *Real-Time Systems* 45.1-2 (06/2010), pp. 72–105. ISSN: 0922-6443. DOI: 10.1007/s11241-010-9091-8. URL: http://dx.doi.org/10.1007/s11241-010-9091-8 (Cited on page 40).

[KS92] Jens Knoop and Bernhard Steffen. "The interprocedural coincidence theorem". English. In: *Compiler Construction*. Ed. by Uwe Kastens and Peter Pfahler. Vol. 641. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 125–140. ISBN: 978-3-540-55984-9. DOI: 10.1007/3-540-55984-1_13. URL: http://dx.doi.org/10.1007/3-540-55984-1%5C_13 (Cited on page 16).

[KSP+12] Daniel Kästner, Marc Schlickling, Markus Pister, Christoph Cullmann, Gernot Gebhard, Reinhold Heckmann, and Christian Ferdinand. "Meeting Real-time Requirements with Multi-core Processors". In: *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*. SAFECOMP'12. Magdeburg, Germany: Springer-Verlag, 2012, pp. 117–131. ISBN: 978-3-642-33674-4. DOI: 10.1007/978-3-642-33675-1_10. URL: http://dx.doi.org/10.1007/978-3-642-33675-1%5C_10 (Cited on page 89).

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. "Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs". In: *ACM Trans. Program. Lang. Syst.* 18.3 (05/1996), pp. 268–299. ISSN: 0164-0925. DOI: 10.1145/229542.229545. URL: http://doi.acm.org/10.1145/229542.229545 (Cited on page 122).

[KV08] Johannes Kinder and Helmut Veith. "Jakstab: A Static Analysis Platform for Binaries". In: *Proceedings of the 20th International Conference on Computer Aided Verification*. CAV '08. Princeton, NJ, USA: Springer-Verlag, 2008, pp. 423–427. ISBN: 978-3-540-70543-7. DOI: `10.1007/978-3-540-70545-1\_40`. URL: `http://dx.doi.org/10.1007/978-3-540-70545-1%5C_40` (Cited on page 49).

[KV12] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 5th edition. Springer, 2012 (Cited on page 74).

[KVA+13] Leonidas Kosmidis, Tullio Vardanega, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. "Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources". In: *13th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Claire Maiza. Vol. 30. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013, pp. 97–108. ISBN: 978-3-939897-54-5. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2013.97`. URL: `http://drops.dagstuhl.de/opus/volltexte/2013/4126` (Cited on page 22).

[KY06] Amir Kamil and Katherine Yelick. "Concurrency Analysis for Parallel Programs with Textually Aligned Barriers". In: *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*. LCPC'05. Hawthorne, NY: Springer-Verlag, 2006, pp. 185–199. ISBN: 3-540-69329-7, 978-3-540-69329-1. DOI: `10.1007/978-3-540-69330-7\_13`. URL: `http://dx.doi.org/10.1007/978-3-540-69330-7%5C_13` (Cited on page 121).

[KYM+09] Florian Kluge, Chenglong Yu, Jörg Mische, Sascha Uhrig, and Theo Ungerer. "Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor". In: *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '09. Nice, France: ACM, 2009, pp. 33–42. ISBN: 978-1-60558-696-0. URL: `http://dl.acm.org/citation.cfm?id=1543820.1543828` (Cited on page 33).

[KZR09] Raimund Kirner, W. Zimmermann, and D. Richter. "On undecidability results of real programming languages". In: *Proc. of Kolloquium Programmiersprachen und Grundlagen der Programmierung*. 2009 (Cited on page 3).

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries". In: *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI '09. Savannah, GA: Springer-Verlag, 2009, pp. 214–228. ISBN: 978-3-540-93899-6. DOI: `10.1007/978-3-540-93900-9\_19`. URL: `http://dx.doi.org/10.1007/978-3-540-93900-9%5C_19` (Cited on page 49).

[LAL+13] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. *Analysis of Global EDF for Parallel Tasks*. Tech. rep. Campus Box 1045 - St. Louis, MO - 63130: Department of Computer Science & Engineering - Washington University in St. Louis, 2013 (Cited on page 34).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (07/1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563 (Cited on page 86).

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. "An Accurate Worst Case Timing Analysis for RISC Processors". In: *IEEE Transactions on Software Engineering* 21.7 (07/1995), pp. 593–604. ISSN: 0098-5589. DOI: 10.1109/32.392980. URL: http://dx.doi.org/10.1109/32.392980 (Cited on page 70).

[LBR10] Karthik Lakshmanan, Gaurav Bhatia, and Raj Rajkumar. "Integrated End-to-end Timing Analysis of Networked AUTOSAR-compliant Systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '10. Dresden, Germany: European Design and Automation Association, 2010, pp. 331–334. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871007 (Cited on page 35).

[LBS+10] Arun Lakhotia, Davidson R. Boccardo, Anshuman Singh, and Aleardo Manacero Jr. "Context-sensitive Analysis Without Calling-context". In: *Higher Order Symbol. Comput.* 23.3 (09/2010), pp. 275–313. ISSN: 1388-3690. DOI: 10.1007/s10990-011-9080-1. URL: http://dx.doi.org/10.1007/s10990-011-9080-1 (Cited on page 58).

[LCF+09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. "A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models". In: *International Symposium on Code Generation and Optimization (CGO)*. Seattle / USA, 03/2009, pp. 136–146 (Cited on page 41).

[LCS92] Corinna G. Lee, Paul Chow, and Mark G. Stoodley. *UTDSP Benchmark Suite*. University of Toronto, 10 King's College Road, Toronto, Canada M5S 3G4, 1992. URL: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html (Cited on pages 76, 150).

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (07/2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814 (Cited on page 39).

[LES+13] Björn Lisper, Andreas Ermedahl, Dietmar Schreiner, Jens Knoop, and Peter Gliwa. "Practical experiences of applying source-level WCET flow analysis to industrial code". English. In: *International Journal on Software Tools for Technology Transfer* 15.1 (2013), pp. 53–63. ISSN: 1433-2779. DOI: 10.1007/s10009-012-0255-9. URL: http://dx.doi.org/10.1007/s10009-012-0255-9 (Cited on pages 23 sq.).

[LGZ+09] Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. "A Survey of WCET Analysis of Real-Time Operating Systems". In: *International Conference on Embedded Software and Systems, 2009*. 05/2009, pp. 65–72. DOI: 10.1109/ICESS.2009.24 (Cited on pages 29 sq.).

[LHP09] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. "WCET Analysis of Multi-Level Set-Associative Data Caches". In: *WCET*. 2009 (Cited on page 73).

[LHP10] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. "Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures." English. In: *18th International Conference on Real-Time and Network Systems*. Toulouse, France, 11/2010, p. 2283. URL: `http://hal.inria.fr/inria-00531214` (Cited on page 87).

[LHS+98] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling". In: *Computers, IEEE Transactions on* 47.6 (1998), pp. 700–713 (Cited on pages 30 sq.).

[Liu00] Jane W. S. W. Liu. *Real-Time Systems*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0130996513 (Cited on page 86).

[LLM+07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. "Chronos: A Timing Analyzer for Embedded Software". In: *Science of Computer Programming* 69.1-3 (12/2007), pp. 56–67. ISSN: 0167-6423. DOI: `10.1016/j.scico.2007.01.014`. URL: `http://dx.doi.org/10.1016/j.scico.2007.01.014` (Cited on page 20).

[LM10] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 11/2010 (Cited on page 39).

[LM97] Yau-Tsun Steven Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.12 (12/1997), pp. 1477–1487. ISSN: 0278-0070. DOI: `10.1109/43.664229` (Cited on pages 56, 74).

[LMW95] Y.-T. S. Li, S. Malik, and A. Wolfe. "Efficient Microarchitecture Modeling and Path Analysis for Real-time Software". In: *Proceedings of the 16th IEEE Real-Time Systems Symposium*. RTSS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 298–. ISBN: 0-8186-7337-0. URL: `http://dl.acm.org/citation.cfm?id=827267.828940` (Cited on page 20).

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. "Cache Modeling for Real-time Software: Beyond Direct Mapped Instruction Caches". In: *Proceedings of the 17th IEEE Real-Time Systems Symposium*. RTSS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 254–. ISBN: 0-8186-7689-2. URL: `http://dl.acm.org/citation.cfm?id=827268.828947` (Cited on pages 20, 70).

[LPF+10] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. "Multi-Objective Exploration of Compiler Optimizations for Real-Time Systems". In: *Proceedings of the 13th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*. Carmona / Spain, 05/2010, pp. 115–122 (Cited on page 37).

[LPM97] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems". In: *Proceedings of the Thirtieth IEEE/ACM International Symposium on Microarchitecture.* 12/1997, pp. 330–335. DOI: 10.1109/MICRO.1997.645830 (Cited on pages 76, 150).

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. "Analytic Real-time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-time Systems". In: *Proceedings of the Seventh ACM International Conference on Embedded Software.* EMSOFT '09. Grenoble, France: ACM, 2009, pp. 107–116. ISBN: 978-1-60558-627-4. DOI: 10.1145/1629335.1629351. URL: http://doi.acm.org/10.1145/1629335.1629351 (Cited on page 34).

[LPW09] Philipp Lucas, Oleg Parshin, and Reinhard Wilhelm. "Operating Mode Specific WCET Analysis". In: *Proceedings of the 3rd Junior Researcher Workshop on Real-Time Computing (JRWRTC).* Ed. by Charlotte Seidner. 10/2009, pp. 15–18 (Cited on page 21).

[LRB+12] I Liu, J. Reineke, D. Broman, M. Zimmer, and E.A Lee. "A PRET microarchitecture implementation with repeatable timing and competitive performance". In: *Computer Design (ICCD), 2012 IEEE 30th International Conference on.* 09/2012, pp. 87–93. DOI: 10.1109/ICCD.2012.6378622 (Cited on pages 23, 89).

[LRL10] Isaac Liu, Jan Reineke, and Edward A. Lee. "A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties". In: *44th Asilomar Conference on Signals, Systems, and Computers.* 11/7/2010, pp. 2111–2115. URL: http://chess.eecs.berkeley.edu/pubs/803.html (Cited on page 89).

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. "Modeling Out-of-order Processors for WCET Analysis". In: *Real-Time Systems* 34.3 (11/2006), pp. 195–227. ISSN: 0922-6443. DOI: 10.1007/s11241-006-9205-5. URL: http://dx.doi.org/10.1007/s11241-006-9205-5 (Cited on pages 66, 88).

[LSL+09] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores". In: *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium.* RTSS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 57–67. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.32. URL: http://dx.doi.org/10.1109/RTSS.2009.32 (Cited on pages 87 sq., 91, 94).

[Ltd14] Rapita Systems Ltd. *RapiTime Explained.* http://www.rapitasystems.com/products/rapitime. 2014 (Cited on page 22).

[LTH02] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. "Pipeline Modeling for Timing Analysis". In: *Proceedings of the 9th International Symposium on Static Analysis.* SAS '02. London, UK, UK: Springer-Verlag, 2002, pp. 294–309. ISBN: 3-540-44235-9. URL: http://dl.acm.org/citation.cfm?id=647171.716098 (Cited on page 66).

[LYG+10] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software". In: *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*. RTSS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 339–349. ISBN: 978-0-7695-4298-0. DOI: `10.1109/RTSS.2010.30`. URL: `http://dx.doi.org/10.1109/RTSS.2010.30` (Cited on pages 33, 88).

[MA11] P. Montag and S. Altmeyer. "Precise WCET calculation in highly variant real-time systems". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 03/2011, pp. 1–6. DOI: `10.1109/DATE.2011.5763149` (Cited on page 21).

[Mäl05] Mälardalen WCET research group. *MRTC Benchmarks*. 2005. URL: `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html` (Cited on pages 76, 150).

[Mar10] Amine Marref. "Compositional Timing Analysis". In: *SAMOS X - International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 07/2010. URL: `http://www.es.mdh.se/publications/1819-` (Cited on page 28).

[Mar11] Peter Marwedel. *Embedded System Design*. 2nd. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2011. ISBN: 978-94-007-0257-8 (Cited on pages 1, 32 sq., 82, 86).

[MB11] Robert Mittermayr and Johann Blieberger. *Shared Memory Concurrent System Verification using Kronecker Algebra*. Tech. rep. 183/1-155. Cornell University Library, 2011 (Cited on page 31).

[MB12] Robert Mittermayr and Johann Blieberger. "Timing Analysis of Concurrent Programs". In: *12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012)*. 2012, pp. 59–68 (Cited on page 123).

[MGU+10] Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. "How to Enhance a Superscalar Processor to Provide Hard Real-time Capable In-order SMT". In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems*. ARCS'10. Hannover, Germany: Springer-Verlag, 2010, pp. 2–14. ISBN: 3-642-11949-2, 978-3-642-11949-1. DOI: `10.1007/978-3-642-11950-7\_2`. URL: `http://dx.doi.org/10.1007/978-3-642-11950-7%5C_2` (Cited on page 89).

[Mid12] Samuel P. Midkiff. "Automatic Parallelization: An Overview of Fundamental Compiler Techniques". In: *Synthesis Lectures on Computer Architecture*. Ed. by University of Wisconsin Mark D. Hill. Vol. Lecture #19. Morgan & Claypool, 2012. ISBN: 9781608458424. DOI: `10.2200/S003401D1V01Y201201CAC019` (Cited on page 33).

[Min12] Antoine Miné. "Static Analysis of Run-Time Errors In Embedded Real-Time Parallel C Programs". In: *Logical Methods in Computer Science* 8.1:26 (03/2012), p. 63. DOI: `10.2168/LMCS-8`. URL: `http://hal.archives-ouvertes.fr/hal-00748098` (Cited on page 122).

[MIS13] MISRA Ltd. *Guidelines for the Use of the C Language in Critical Systems*. 03/2013 (Cited on pages 2, 19).

[MMH01] G. Memik, W.H. Mangione-Smith, and W. Hu. “NetBench: A Benchmarking Suite for Network Processors”. In: *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 11/2001, pp. 39–42. DOI: `10.1109/ICCAD.2001.968595` (Cited on page 76).

[MMU11] Stefan Metzlaff, Jörg Mische, and Theo Ungerer. “A Real-Time Capable Many-Core Model”. In: *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*. Vienna, Austria, 2011 (Cited on page 89).

[MPH14] Daniel Muench, Michael Paulitsch, and Andreas Herkersdorf. “Temporal Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using PCIe SR-IOV”. In: *27th International Conference on Architecture of Computing Systems (ARCS)*. 02/2014, pp. 1–7 (Cited on page 90).

[MR12] Mohamed Abdel Maksoud and Jan Reineke. “An Empirical Evaluation of the Influence of the Load-Store Unit on WCET Analysis”. In: *12th International Workshop on Worst-Case Execution Time Analysis*. Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012, pp. 13–24. ISBN: 978-3-939897-41-5. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2012.13`. URL: `http://drops.dagstuhl.de/opus/volltexte/2012/3553` (Cited on page 23).

[MR99] Mark Moir and Srikanth Ramamurthy. “Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resources”. In: *Proceedings of the 20th IEEE Real-Time Systems Symposium*. RTSS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 294–. ISBN: 0-7695-0475-2. URL: `http://dl.acm.org/citation.cfm?id=827271.829072` (Cited on page 34).

[MRT14] MRTC Research Group. *SWEET (SWEdish Execution Time tool)*. `http://www.mrtc.mdh.se/projects/wcet/sweet/index.html`. 2014 (Cited on pages 20, 38).

[MUK+08] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. “Exploiting spare resources of in-order SMT processors executing hard real-time threads”. In: *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. 10/2008, pp. 371–376. DOI: `10.1109/ICCD.2008.4751887` (Cited on page 89).

[Myc07] Alan Mycroft. “Programming Language Design and Analysis Motivated by Hardware Evolution”. In: *Proceedings of the 14th International Conference on Static Analysis*. SAS’07. Kongens Lyngby, Denmark: Springer-Verlag, 2007, pp. 18–33. ISBN: 3-540-74060-0, 978-3-540-74060-5. URL: `http://dl.acm.org/citation.cfm?id=2391451.2391454` (Cited on page 1).

[NAC99] Gleb Naumovich, GeorgeS. Avrunin, and LoriA. Clarke. “An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs”. English. In: *Software Engineering — ESEC/FSE ’99*. Ed. by Oscar Nierstrasz and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 338–354. ISBN: 978-3-540-66538-0. URL: `http://dx.doi.org/10.1007/3-540-48166-4%5C_21` (Cited on page 121).

[NKJ10] Armand Navabi, Nicholas Kidd, and Suresh Jagannathan. *Path-Sensitive Analysis Using Edge Strings*. Tech. rep. 10-006. Purdue University, Department of Computer Science, 2010 (Cited on pages 17 sq.).

[NMR03] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. “Accurate Estimation of Cache-related Preemption Delay”. In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’03. Newport Beach, CA, USA: ACM, 2003, pp. 201–206. ISBN: 1-58113-742-7. DOI: `10.1145/944645.944698`. URL: `http://doi.acm.org/10.1145/944645.944698` (Cited on page 31).

[NN13] Farhang Nemati and Thomas Nolte. “Resource sharing among real-time components under multiprocessor clustered scheduling”. English. In: *Real-Time Systems* 49.5 (2013), pp. 580–613. ISSN: 0922-6443. DOI: `10.1007/s11241-013-9180-6`. URL: `http://dx.doi.org/10.1007/s11241-013-9180-6` (Cited on page 34).

[NP13] Jan Nowotsch and Michael Paulitsch. “Quality of service capabilities for hard real-time applications on multi-core processors”. In: *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*. 2013, pp. 151–160 (Cited on page 88).

[NPH+14] Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. “Monitoring and WCET Analysis in COTS multi-core-SoC-based Mixed-criticality Systems”. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE ’14. Dresden, Germany: European Design and Automation Association, 2014, 67:1–67:5. ISBN: 978-3-9815370-2-4. URL: `http://dl.acm.org/citation.cfm?id=2616606.2616689` (Cited on page 88).

[NSE09] M. Negrean, S. Schliecker, and R. Ernst. “Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources”. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE ’09*. 04/2009, pp. 524–529. DOI: `10.1109/DATE.2009.5090720` (Cited on page 87).

[Ope14] Open-Source Community. *Free OSEK RTOS*. `http://sourceforge.net/projects/opensek`. 2014 (Cited on page 40).

[ORM+09] Jin Ouyang, Raghuveer Raghavendra, Sibin Mohan, Tao Zhang, Yuan Xie, and Frank Mueller. “CheckerCore: Enhancing an FPGA Soft Core to Capture Worst-case Execution Times”. In: *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES ’09. Grenoble, France: ACM, 2009, pp. 175–184. ISBN: 978-1-60558-626-7. DOI: `10.1145/1629395.1629421`. URL: `http://doi.acm.org/10.1145/1629395.1629421` (Cited on page 23).

[ORS14] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. “Minimizing the Cost of Synchronisations in the WCET of Real-time Parallel Programs”. In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’14. Sankt Goar, Germany: ACM, 2014, pp. 98–107. ISBN: 978-1-4503-2941-5. DOI: `10.1145/2609248.2609261`. URL: `http://doi.acm.org/10.1145/2609248.2609261` (Cited on page 88).

[OSE01] OSEK/VDX. *OSEK/VDX Time-Triggered Operating System*. Version 1.0. 2001 (Cited on page 123).

[PC07] Rodolfo Pellizzoni and Marco Caccamo. "Toward the Predictable Integration of Real-Time COTS Based Systems". In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium.* RTSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 73–82. ISBN: 0-7695-3062-1. DOI: 10.1109/RTSS.2007.51. URL: http://dx.doi.org/10.1109/RTSS.2007.51 (Cited on page 32).

[PC10] R. Pellizzoni and M. Caccamo. "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems". In: *Computers, IEEE Transactions on* 59.3 (03/2010), pp. 400–415. ISSN: 0018-9340. DOI: 10.1109/TC.2009.156 (Cited on page 87).

[PH11] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* 5th edition. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2011. ISBN: 012383872X (Cited on pages 82 sq.).

[PHP14] Daniel Prokesch, Benedikt Huber, and Peter Puschner. "Towards Automated Generation of Time-Predictable Code". In: *14th International Workshop on Worst-Case Execution Time Analysis.* Ed. by Heiko Falk. Vol. 39. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014, pp. 103–112. ISBN: 978-3-939897-69-9. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2014.103. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4609 (Cited on page 38).

[PK08] C. Paukovits and H. Kopetz. "Concepts of Switching in the Time-Triggered Network-on-Chip". In: *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008.* RTCSA '08. 08/2008, pp. 120–129. DOI: 10.1109/RTCSA.2008.18 (Cited on page 86).

[PK89] P. Puschner and Ch. Koza. "Calculating the Maximum Execution Time of Real-time Programs". In: *Real-Time Systems* 1.2 (09/1989), pp. 159–176. ISSN: 0922-6443. DOI: 10.1007/BF00571421. URL: http://dx.doi.org/10.1007/BF00571421 (Cited on page 74).

[PKF+11] Sascha Plazar, Jan C. Kleinsorge, Heiko Falk, and Peter Marwedel. "WCET-driven Branch Prediction aware Code Positioning". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES).* Taipei, Taiwan, 10/2011, pp. 165–174 (Cited on page 37).

[PKH+12] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. "Compiling for Time Predictability". English. In: *Computer Safety, Reliability, and Security.* Ed. by Frank Ortmeier and Peter Daniel. Vol. 7613. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 382–391. ISBN: 978-3-642-33674-4. DOI: 10.1007/978-3-642-33675-1_35. URL: http://dx.doi.org/10.1007/978-3-642-33675-1%5C_35 (Cited on page 38).

[Plu10] Plurality Ltd. *The HyperCore Architecture Version 1.0.* 3 Hanotea St., Netanya 42300, Israel, 01/2010. URL: www.plurality.com (Cited on page 84).

[PM11] M. Paolieri and R. Mariani. “Towards Functional-safe Timing-dependable Real-time Architectures”. In: *Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium*. IOLTS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 31–36. ISBN: 978-1-4577-1053-7. DOI: 10.1109/IOLTS.2011.5993807. URL: http://dx.doi.org/10.1109/IOLTS.2011.5993807 (Cited on page 89).

[Pou12] Louis-Noel Pouchet. *PolyBench/C - The Polyhedral Benchmark Suite*. http://www.cs.ucla.edu/~pouchet/software/polybench/. 2012 (Cited on page 76).

[PP13] Dumitru Potop-Butucaru and Isabelle Puaut. *Integrated Worst-Case Response Time Evaluation of Multicore Non-Preemptive Applications*. Rapport de recherche RR-8234. INRIA, 02/2013. URL: http://hal.inria.fr/hal-00787931 (Cited on pages 88, 91, 139).

[PPE+08] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. “Timing Analysis of the FlexRay Communication Protocol”. In: *Real-Time Systems* 39.1-3 (08/2008), pp. 205–235. ISSN: 0922-6443. DOI: 10.1007/s11241-007-9040-3. URL: http://dx.doi.org/10.1007/s11241-007-9040-3 (Cited on page 35).

[PQC+09a] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. “An Analyzable Memory Controller for Hard Real-Time CMPs”. In: *IEEE Embedded Systems Letters* 1.4 (12/2009), pp. 86–90. ISSN: 1943-0663. DOI: 10.1109/LES.2010.2041634. URL: http://dx.doi.org/10.1109/LES.2010.2041634 (Cited on page 89).

[PQC+09b] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. “Hardware Support for WCET Analysis of Hard Real-time Multicore Systems”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: ACM, 2009, pp. 57–68. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555764. URL: http://doi.acm.org/10.1145/1555754.1555764 (Cited on page 89).

[PRU13] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. “A Real-Time Capable First-Level Cache for Multi-Cores”. In: *Workshop on High-Performance and Real-Time Embedded Systems*. HiRES 2013. Berlin, Gemany, 01/2013 (Cited on page 90).

[PS10] Christof Pitter and Martin Schoeberl. “A Real-Time Java Chip-Mulitiprocessor”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 10.1 (08/2010), 9:1–9:34. ISSN: 1539-9087. DOI: 10.1145/1814539.1814548. URL: http://doi.acm.org/10.1145/1814539.1814548 (Cited on pages 46, 89).

[PS91] Chang Yun Park and Alan C. Shaw. “Experiments with a Program Timing Tool Based on Source-Level Timing Schema”. In: *Computer - Special issue on real-time systems* 24.5 (05/1991), pp. 48–57. ISSN: 0018-9162. DOI: 10.1109/2.76286. URL: http://dx.doi.org/10.1109/2.76286 (Cited on page 74).

[PSC+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. “Worst Case Delay Analysis for Memory Interference in Multicore Systems”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’10. Dresden, Germany: European Design and Automation Association, 2010, pp. 741–746. ISBN: 978-3-9810801-6-2. URL: `http://dl.acm.org/citation.cfm?id=1870926.1871105` (Cited on page 87).

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. "TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008).* ISBN: 978-3-85403-237-3. Prague, Czech Republic: Österreichische Computer Gesellschaft, 2008, pp. 141–148. URL: `http://costa.tuwien.ac.at/papers/wcet08-tubound.pdf` (Cited on pages 20, 38).

[Ram00] G. Ramalingam. "Context-sensitive Synchronization-sensitive Analysis is Undecidable". In: *ACM Trans. Program. Lang. Syst.* 22.2 (03/2000), pp. 416–430. ISSN: 0164-0925. DOI: `10.1145/349214.349241`. URL: `http://doi.acm.org/10.1145/349214.349241` (Cited on page 121).

[RBS+10] Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. "WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).* Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2010, pp. 90–100. ISBN: 978-3-939897-21-7. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2010.90`. URL: `http://drops.dagstuhl.de/opus/volltexte/2010/2829` (Cited on page 88).

[RD14] Jan Reineke and Johannes Doerfert. "Architecture-Parametric Timing Analysis". In: *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS).* Ed. by Richard West. IEEE. 04/2014, pp. 189–200. URL: `http://chess.eecs.berkeley.edu/pubs/1070.html` (Cited on page 21).

[Rei14] Jan Reineke. "Randomized Caches Considered Harmful in Hard Real-Time Systems". In: *LITES* 1.1 (2014), 03:1–03:13 (Cited on pages 22, 45).

[RNE+11] J. Rosén, C. Neikter, P. Eles, Zebo Peng, P. Burgio, and L. Benini. "Bus Access Design for Combined Worst and Average Case Execution Time Optimization of Predictable Real-Time Applications on Multiprocessor Systems-on-Chip". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE.* 04/2011, pp. 291–301. DOI: `10.1109/RTAS.2011.35` (Cited on page 148).

[RS09] Jan Reineke and Rathijit Sen. "Sound and Efficient WCET Analysis in the Presence of Timing Anomalies". In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09).* Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11. ISBN: 978-3-939897-14-9. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2009.2289`. URL: `http://drops.dagstuhl.de/opus/volltexte/2009/2289` (Cited on page 26).

[RSL88] R. Rajkumar, Lui Sha, and J.P. Lehoczky. “Real-time synchronization protocols for multiprocessors”. In: *Real-Time Systems Symposium, 1988., Proceedings.* 12/1988, pp. 259–269. DOI: 10.1109/REAL.1988.51121 (Cited on page 33).

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. “A Definition and Classification of Timing Anomalies”. In: *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis.* 07/2006 (Cited on pages 25 sq., 106).

[Sch97] R.R. Schaller. “Moore’s law: past, present and future”. In: *Spectrum, IEEE* 34.6 (06/1997), pp. 52–59. ISSN: 0018-9235. DOI: 10.1109/6.591665 (Cited on page 1).

[SCT09] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. “Timing predictability on multi-processor systems with shared resources”. In: *Embedded Systems Week - Workshop on Reconciling Performance with Predictability.* 2009 (Cited on page 87).

[SCT10] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. “Timing Analysis for TDMA Arbitration in Resource Sharing Systems”. In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium.* RTAS ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 215–224. ISBN: 978-0-7695-4001-6. DOI: 10.1109/RTAS.2010.24. URL: http://dx.doi.org/10.1109/RTAS.2010.24 (Cited on page 87).

[SE09] Simon Schliecker and Rolf Ernst. “A Recursive Approach to End-to-end Path Latency Computation in Heterogeneous Multiprocessor Systems”. In: *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis.* CODES+ISSS ’09. Grenoble, France: ACM, 2009, pp. 433–442. ISBN: 978-1-60558-628-1. DOI: 10.1145/1629435.1629494. URL: http://doi.acm.org/10.1145/1629435.1629494 (Cited on page 34).

[SF99] Jörn Schneider and Christian Ferdinand. “Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation”. In: *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems.* LCTES ’99. Atlanta, Georgia, USA: ACM, 1999, pp. 35–44. ISBN: 1-58113-136-4. DOI: 10.1145/314403.314432. URL: http://doi.acm.org/10.1145/314403.314432 (Cited on page 66).

[SHK14] H. Shah, Kai Huang, and A Knoll. “Timing anomalies in multi-core architectures due to the interference on the shared resources”. In: *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific.* 01/2014, pp. 708–713. DOI: 10.1109/ASPDAC.2014.6742973 (Cited on pages 26, 34).

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. “A Primer on Memory Consistency and Cache Coherence”. In: *Synthesis Lectures on Computer Architecture.* Ed. by University of Wisconsin Mark D. Hill. Vol. Lecture #16. Morgan & Claypool, 2011. ISBN: 9781608455652. DOI: 10.2200/S00346ED1V01Y201104CAC016 (Cited on page 83).

[SJ96] Thomas Schwederski and Michael Jurczyk. *Verbindungsnetze - Strukturen und Eigenschaften.* Leitfäden der Informatik. Teubner, 1996, pp. I–XVI, 1–420. ISBN: 978-3-519-02134-6 (Cited on pages 84 sq.).

[SLL+11] Marcelo Santos, Björn Lisper, George Lima, and Veronica Lima. "Sequential Composition of Execution Time Distributions by Convolution". In: *Proc. 4th Workshop on Compositional Theory and Technology for Real&#8208;Time Embedded Systems (CRTS 2011)*. Ed. by Robert Davis and Linh Thi Xuan Phan. Best paper award. 11/2011, pp. 30–37. URL: `http://www.es.mdh.se/publications/2215-` (Cited on page 22).

[SNN+08] Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. "Reliable Performance Analysis of a Multicore Multithreaded System-on-chip". In: *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '08. Atlanta, GA, USA: ACM, 2008, pp. 161–166. ISBN: 978-1-60558-470-6. DOI: `10 . 1145 / 1450135 . 1450172`. URL: `http://doi.acm.org/10.1145/1450135.1450172` (Cited on page 87).

[SP10] Marc Schlickling and Markus Pister. "Semi-automatic Derivation of Timing Models for WCET Analysis". In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '10. Stockholm, Sweden: ACM, 2010, pp. 67–76. ISBN: 978-1-60558-953-4. DOI: `10.1145/1755888.1755899`. URL: `http://doi.acm.org/10.1145/1755888.1755899` (Cited on page 67).

[SP78] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural dataflow analysis*. Tech. rep. 2. 251 Mercer Street, New York, N.Y.: New York University, Department of Computer Science, 09/1978 (Cited on page 57).

[SPC+10] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. "Worst-case Response Time Analysis of Resource Access Models in Multi-core Systems". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 332–337. ISBN: 978-1-4503-0002-5. DOI: `10.1145/1837274.1837359`. URL: `http://doi.acm.org/10.1145/1837274.1837359` (Cited on pages 87, 155).

[SPH+07] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Guillaume Borios, Victor Jégu, and Reinhold Heckmann. "Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation". In: *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Ed. by Reinhard Wilhelm. Vol. 1. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2007. ISBN: 978-3-939897-24-8. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2005.810`. URL: `http://drops.dagstuhl.de/opus/volltexte/2007/810` (Cited on page 142).

[SPP+10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. "Worst-case Execution Time Analysis for a Java Processor". In: *Software Practice and Experience* 40.6 (05/2010), pp. 507–542. ISSN: 0038-0644. DOI: `10.1002/spe.v40:6`. URL: `http://dx.doi.org/10.1002/spe.v40:6` (Cited on page 23).

[SRK11] H. Shah, A Raabe, and A Knoll. "Priority division: A high-speed shared-memory bus arbitration with bounded latency". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 03/2011, pp. 1–4. DOI: `10.1109/DATE.2011.5763319` (Cited on page 86).

[SSB09] Andrew Stone, Michelle Strout, and Shweta Behere. “May/must analysis and the {DFAGen} data-flow analysis generator”. In: *Information and Software Technology* 51.10 (2009). Source Code Analysis and Manipulation, {SCAM} 2008, pp. 1440–1453. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2009.04.014. URL: http://www.sciencedirect.com/science/article/pii/S0950584909000482 (Cited on pages 18, 53).

[Str14] StreamIt Community. *The StreamIt Benchmark Suite*. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml. 2014 (Cited on page 76).

[Sut12] Herb Sutter. *Welcome to the Parallel Jungle!* http://drdobbs.com/parallel/232400273. 01/2012 (Cited on pages 4, 32).

[Syn14] Synopsys Inc. *CoMET System Engineering IDE*. http://www.synopsys.com. 2014 (Cited on pages 9, 44, 150).

[SZW+10] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. “Precise Calling Context Encoding”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 525–534. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806875. URL: http://doi.acm.org/10.1145/1806799.1806875 (Cited on page 58).

[Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309 (Cited on page 15).

[Tay83a] Richard N. Taylor. “A General-purpose Algorithm for Analyzing Concurrent Programs”. In: *Commun. ACM* 26.5 (05/1983), pp. 361–376. ISSN: 0001-0782. DOI: 10.1145/69586.69587. URL: http://doi.acm.org/10.1145/69586.69587 (Cited on pages 121 sq., 139).

[Tay83b] RichardN. Taylor. “Complexity of analyzing the synchronization structure of concurrent programs”. English. In: *Acta Informatica* 19.1 (1983), pp. 57–84. ISSN: 0001-5903. DOI: 10.1007/BF00263928. URL: http://dx.doi.org/10.1007/BF00263928 (Cited on page 121).

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. “Analysis of hard real-time communications”. English. In: *Real-Time Systems* 9.2 (1995), pp. 147–171. ISSN: 0922-6443. DOI: 10.1007/BF01088855. URL: http://dx.doi.org/10.1007/BF01088855 (Cited on page 31).

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. “Real-time Calculus for Scheduling Hard Real-Time Systems”. In: *International Symposium on Circuits and Systems ISCAS 2000*. Vol. 4. Geneva, Switzerland, 03/2000, pp. 101–104 (Cited on page 32).

[TD00] Hiroyuki Tomiyama and Nikil D. Dutt. “Program Path Analysis to Bound Cache-related Preemption Delay in Preemptive Real-time Systems”. In: *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*. CODES ’00. San Diego, California, USA: ACM, 2000, pp. 67–71. ISBN: 1-58113-268-9. DOI: 10.1145/334012.334025. URL: http://doi.acm.org/10.1145/334012.334025 (Cited on page 31).

[Tea14] TRACES Team. *OTAWA WCET Analysis Framework.* http://www.otawa.fr/. IRIT, 118 Route de Narbonne, F-31062 Toulouse, 2014 (Cited on page 20).

[Tec14] Tech-Pro.net. *How the PCI Bus Works.* http://www.tech-pro.net/intro_pci.html. 2014 (Cited on page 85).

[The00] H. Theiling. "Extracting Safe and Precise Control Flow from Binaries". In: *Proceedings of the Seventh International Conference on Real-Time Systems and Applications.* RTCSA '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 23–. ISBN: 0-7695-0930-4. URL: http://dl.acm.org/citation.cfm?id=580571.828823 (Cited on page 49).

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* Pirrot, 2004. ISBN: 9783937436005. URL: http://books.google.de/books?id=Sbq0SgAACAAJ (Cited on page 66).

[Thi05] Lothar Thiele. "Modular Performance Analysis of Distributed Embedded Systems". In: *Lecture Notes in Computer Science. FORMATS 2005.* Vol. 3829. Springer Verlag, 2005, pp. 1–2 (Cited on page 34).

[Tid04] Tidorum Ltd. *Bound-T Execution Time Analyzer.* http://www.bound-t.com/. 2004 (Cited on page 20).

[Tid10] Tidorum Ltd. *Bound-T Time and Stack Analyser, Application Note ARM7TDMI.* http://www.bound-t.com/. 2010 (Cited on page 49).

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. "An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software." In: *DSN.* 2003, pp. 625–632 (Cited on page 23).

[Val89] Antti Valmari. "Eliminating Redundant Interleavings During Concurrent Program Verification". In: *Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages.* PARLE '89. London, UK, UK: Springer-Verlag, 1989, pp. 89–103. ISBN: 3-540-51285-3. URL: http://dl.acm.org/citation.cfm?id=646427.692577 (Cited on page 121).

[Vol95] Jürgen Vollmer. "Data Flow Analysis of Parallel Programs". In: *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques.* PACT '95. Limassol, Cyprus: IFIP Working Group on Algol, 1995, pp. 168–177. ISBN: 0-89791-745-6. URL: http://dl.acm.org/citation.cfm?id=224659.224717 (Cited on page 122).

[WB05] Heinz Wörn and Uwe Brinkschulte. *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen.* eXamen. press Series. Springer, 2005. ISBN: 9783540205883 (Cited on pages 85 sq.).

[WEE+08] 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 Transactions on Embedded Computing Systems* 7.3 (05/2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389 (Cited on pages 18, 94).

[Weg03] Ingo Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen.* Springer, 2003. ISBN: 3-540-00161-1 (Cited on page 1).

[Weg12] Simon Wegener. "Computing Same Block Relations for Relational Cache Analysis". In: *12th International Workshop on Worst-Case Execution Time Analysis.* Ed. by Tullio Vardanega. Vol. 23. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012, pp. 25–37. ISBN: 978-3-939897-41-5. DOI: http://dx.doi.org/10.4230/OASIcs.WCET.2012.25. URL: http://drops.dagstuhl.de/opus/volltexte/2012/3554 (Cited on page 71).

[Weg99] Ingo Wegener. *Theoretische Informatik - eine algorithmenorientierte Einführung.* 2. Auflage. Teubner, 1999. ISBN: 3-519-12123-9 (Cited on page 3).

[Wei07] Karsten Weicker. *Evolutionäre Algorithmen (Leitfäden der Informatik).* Vieweg+Teubner Verlag, 2007. ISBN: 3835102192 (Cited on page 149).

[WFC+09] Reinhard Wilhelm, Christian Ferdinand, Christoph Cullmann, Daniel Grund, Jan Reineke, and Benoît Triquet. "Designing Predictable Multicore Architectures for Avionics and Automotive Systems". In: *Workshop on Reconciling Performance with Predictability (RePP).* 10/2009. URL: http://www.tik.ee.ethz.ch/~jchen/RePP/papers/2-3.pdf (Cited on page 89).

[WGK+10] J. Wolf, M. Gerdes, F. Kluge, S. Uhrig, J. Mische, S. Metzlaff, C. Rochange, H. Cassé, P. Sainrat, and T. Ungerer. "RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor". In: *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010.* 05/2010, pp. 193–201. DOI: 10.1109/ISORC.2010.31 (Cited on page 29).

[WGR+09] 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: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (07/2009), pp. 966–978. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2013287. URL: http://dx.doi.org/10.1109/TCAD.2009.2013287 (Cited on pages 19, 27 sq., 45).

[WHK+13] B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. "Making Shared Caches More Predictable on Multicore Platforms". In: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on.* 07/2013, pp. 157–167. DOI: 10.1109/ECRTS.2013.26 (Cited on page 7).

[Wil04] Reinhard Wilhelm. "Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone". English. In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 309–322. ISBN: 978-3-540-20803-7. DOI: 10.1007/978-3-540-24622-0_25. URL: http://dx.doi.org/10.1007/978-3-540-24622-0%5C_25 (Cited on pages 20, 88).

[Wil12] Stephan Wilhelm. “Symbolic Representations in WCET Analysis”. PhD thesis. Saarland University, 2012. ISBN: 978-3-8442-2463-4 (Cited on pages 63, 66).

[WT06] Ernesto Wandeler and Lothar Thiele. “Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-time Systems”. In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. ASP-DAC ’06. Yokohama, Japan: IEEE Press, 2006, pp. 479–484. ISBN: 0-7803-9451-8. DOI: 10.1145/1118299.1118417. URL: http://dx.doi.org/10.1145/1118299.1118417 (Cited on page 148).

[XMO13] XMOS. *XMOS Timing Analyzer Whitepaper, Rev. 1.1*. http://www.xmos.com/products/tools/xta. 2013 (Cited on page 90).

[YKS11] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. “WCET-Aware optimization of shared cache partition and bus arbitration for hard real-time multicore systems”. In: (2011) (Cited on page 148).

[YZ08] Jun Yan and Wei Zhang. “WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches”. In: *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 80–89. ISBN: 978-0-7695-3146-5. DOI: 10.1109/RTAS.2008.6. URL: http://dx.doi.org/10.1109/RTAS.2008.6 (Cited on page 87).

[ZCS03] Min Zhao, Bruce Childers, and Mary Lou Soffa. “Predicting the Impact of Optimizations for Embedded Systems”. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*. LCTES ’03. San Diego, California, USA: ACM, 2003, pp. 1–11. ISBN: 1-58113-647-1. DOI: 10.1145/780732.780734. URL: http://doi.acm.org/10.1145/780732.780734 (Cited on pages 37, 155).

[ZKW+04] Wankang Zhao, Prasad A. Kulkarni, David B. Whalley, Christopher A. Healy, Frank Mueller, and Gang-Ryung Uh. “Tuning the WCET of Embedded Applications”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. 2004, pp. 472–481 (Cited on page 38).

[ZKW+05] Wankang Zhao, William C. Kreahling, David B. Whalley, Christopher A. Healy, and Frank Mueller. “Improving WCET by Optimizing Worst-Case Paths”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*. 2005, pp. 138–147 (Cited on page 155).

[ZLT02] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. “SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization”. In: *EUROGEN2001 Conference*. 2002 (Cited on page 150).

[ZVS+94] Vojin Zivojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. “DSPSTONE: A DSP-oriented Benchmarking Methodology”. In: *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT’94)*. 1994 (Cited on page 76).

[ZY09] Wei Zhang and Jun Yan. "Accurately Estimating Worst-Case Execution Time for Multi-core Processors with Shared Direct-Mapped Instruction Caches". In: *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. RTCSA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 455–463. ISBN: 978-0-7695-3787-0. DOI: 10.1109/RTCSA.2009.55. URL: http://dx.doi.org/10.1109/RTCSA.2009.55 (Cited on page 87).

Appendix A
Employed Benchmarks

The following table contains a list of the benchmarks used in the evaluations. For each benchmark the name, the *Lines Of Code* (LOC) per benchmark, the binary code size S in bytes, the number of loops L, the maximum loop nesting depth D, the loop bound range B, the average lower ($B_{\text{min}}^{\text{avg}}$) and upper ($B_{\text{max}}^{\text{avg}}$) loop bound and the number of context blocks $|V_\tau^C|$ for a context graph with unlimited call string length but without virtual unrolling.

Name	LOC	S	$L(D)$	B	$B_{\text{min}}^{\text{avg}}/B_{\text{max}}^{\text{avg}}$	$\|V_\tau^C\|$
DSPstone-fixed-point						
complex-multiply-fixed	22	220	0(0)	–	0.0/0.0	6
complex-update-fixed	26	172	0(0)	–	0.0/0.0	6
convolution-fixed	27	108	2(1)	16	16.0/16.0	14
dot-product-fixed	22	124	1(1)	2	2.0/2.0	11
fft-1024-13	311	932	9(3)	[0 – 2048]	571.7/686.3	95
fft-1024-7	282	888	8(3)	[0 – 1024]	386.6/515.6	90
fft-16-13	156	928	9(3)	[0 – 32]	11.0/13.0	95
fft-16-7	149	884	8(3)	[0 – 16]	7.9/10.1	90
fir2dim-fixed	81	536	13(3)	[3 – 16]	5.4/5.4	111
fir-fixed	40	188	2(1)	[15 – 16]	15.5/15.5	21
iir-biquad-N-sections-fixed	42	284	3(1)	[4 – 20]	10.7/10.7	31
iir-biquad-one-section-fixed	25	204	0(0)	–	0.0/0.0	8
lms-fixed	44	316	3(1)	[15 – 16]	15.7/15.7	26
matrix1-fixed	48	288	6(3)	[10 – 100]	55.0/55.0	51
matrix1x3-fixed	23	116	2(2)	3	3.0/3.0	12
matrix2-fixed	48	316	6(3)	[8 – 100]	54.7/54.7	51
n-complex-updates-fixed	38	280	2(1)	16	16.0/16.0	21
n-real-updates-fixed	29	212	2(1)	16	16.0/16.0	21
real-update-fixed	21	112	0(0)	–	0.0/0.0	6
startup-fixed	99	340	5(1)	[1 – 64]	27.4/33.2	58
DSPstone-floating-point						
complex-multiply-float	22	288	0(0)	–	0.0/0.0	12
complex-update-float	28	316	0(0)	–	0.0/0.0	14
convolution-float	27	184	2(1)	16	16.0/16.0	19
dot-product-float	24	184	1(1)	2	2.0/2.0	13
fir2dim-float	81	852	13(3)	[3 – 16]	5.4/5.4	149
fir-float	40	264	2(1)	[15 – 16]	15.5/15.5	29
iir-biquad-N-sections-float	43	344	3(1)	[4 – 20]	10.7/10.7	40
iir-biquad-one-section-float	25	296	0(0)	–	0.0/0.0	17
lms-float	44	352	3(1)	[15 – 16]	15.7/15.7	34

Name	LOC	S	$L(D)$	B	$B^{\text{avg}}_{\text{min}}/B^{\text{avg}}_{\text{max}}$	$\lvert V^C_\tau \rvert$
matrix1-float	49	300	6(3)	[10 – 100]	55.0/55.0	53
matrix1x3-float	41	256	4(2)	[3 – 9]	4.5/4.5	28
matrix2-float	49	340	6(3)	[8 – 100]	54.7/54.7	56
n-complex-updates-float	38	352	2(1)	16	16.0/16.0	29
n-real-updates-float	29	228	2(1)	16	16.0/16.0	23
real-update-float	24	176	0(0)	–	0.0/0.0	8
MRTC						
adpcm-decoder	406	2812	14(1)	[0 – 2424]	66.9/322.2	216
adpcm-encoder	434	2944	15(1)	[0 – 2424]	63.3/303.5	249
binarysearch	35	156	1(1)	4	4.0/4.0	18
bsort100	54	224	3(2)	[2 – 100]	67.0/99.3	31
compressdata	203	1152	4(1)	[0 – 50]	14.0/14.0	32
countnegative	73	492	4(2)	20	20.0/20.0	40
cover	231	3192	3(1)	[10 – 120]	60.0/60.0	789
duff	44	296	1(1)	100	100.0/100.0	50
edn	204	2004	12(3)	[2 – 150]	53.8/53.8	80
expint	62	344	3(2)	[49 – 100]	83.0/83.0	40
fac	21	116	1(1)	6	6.0/6.0	13
fdct	143	1136	2(1)	8	8.0/8.0	14
fibcall	22	96	1(1)	29	29.0/29.0	10
fir	225	268	2(2)	[10 – 26]	13.5/18.0	22
insertsort	61	196	2(2)	[1 – 9]	5.0/9.0	12
janne-complex	61	108	2(2)	[0 – 9]	4.5/9.0	21
jfdctint	218	1460	3(1)	[8 – 64]	26.7/26.7	20
lcdnum	62	348	1(1)	10	10.0/10.0	74
lms	160	1876	10(1)	[0 – 201]	48.2/50.5	250
ludcmp	86	1092	11(3)	[1 – 6]	3.0/5.1	90
matmult	57	496	5(3)	20	20.0/20.0	54
minver	162	1648	17(3)	[1 – 3]	2.8/2.9	162
ndes	407	1800	12(1)	[2 – 32]	19.4/19.4	244
petrinet	500	3920	1(1)	2	2.0/2.0	404
prime	34	320	1(1)	[73 – 357]	73.0/357.0	89
qurt	88	820	1(1)	19	19.0/19.0	215
recursion	16	104	0(0)	–	0.0/0.0	13
select	62	772	4(3)	[0 – 16]	8.2/8.5	65
sqrt	45	320	2(1)	[6 – 19]	12.5/12.5	41
MediaBench						
cjpeg-jpeg6b-transupp	1599	2448	47(7)	[1 – 29]	5.1/7.5	310
cjpeg-jpeg6b-wrbmp	1295	580	5(1)	[30 – 512]	262.0/262.0	128
epic	994	4520	41(4)	[0 – 9801]	132.0/316.6	332
gsm-decode	1380	7480	19(2)	[0 – 648]	58.5/64.4	857
h264dec-ldecode-block	1571	4556	27(2)	[0 – 16]	7.3/7.9	400
MiBench						
bitcount	202	1088	4(2)	[3 – 31]	13.0/14.2	97
dijkstra	227	908	5(2)	[0 – 928]	167.6/292.2	86
PolyBench						

Name	LOC	S	$L(D)$	B	$B^{\text{avg}}_{\text{min}}/B^{\text{avg}}_{\text{max}}$	$\lvert V^C_\tau \rvert$
2mm	133	1244	14(2)	32	32.0/32.0	103
3mm	158	1396	19(3)	32	32.0/32.0	130
atax	106	664	8(2)	32	32.0/32.0	60
cholesky	136	952	9(3)	[0 – 32]	23.4/30.3	95
correlation	175	1500	14(3)	[0 – 32]	28.7/30.9	166
covariance	116	792	11(3)	32	32.0/32.0	73
doitgen	119	972	13(4)	10	10.0/10.0	88
durbin	110	1176	7(2)	[0 – 32]	22.7/31.6	69
dynprog	112	652	8(4)	[0 – 10]	4.2/5.5	50
fdtd-2d	133	844	13(3)	[2 – 32]	27.1/27.1	90
fdtd-apml	194	4376	13(3)	[8 – 9]	8.7/8.7	206
floyd-warshall	87	548	7(3)	32	32.0/32.0	56
gemm	124	936	11(3)	32	32.0/32.0	83
gemver	121	1220	10(2)	32	32.0/32.0	91
gesummv	107	884	5(2)	32	32.0/32.0	54
gramschmidt	165	1196	17(3)	[0 – 32]	29.4/31.2	145
jacobi-1d-imper	97	548	5(2)	[2 – 500]	399.6/399.6	45
jacobi-2d-imper	104	784	9(3)	[2 – 32]	27.8/27.8	71
lu	89	516	8(3)	[0 – 32]	20.0/32.0	57
ludcmp-Poly	131	1028	12(3)	[0 – 33]	16.2/32.2	96
mvt	97	876	7(2)	32	32.0/32.0	67
reg-detect	133	688	14(4)	[0 – 32]	1.9/6.9	81
seidel-2d	95	632	7(3)	[2 – 32]	27.1/27.1	59
StreamIt						
audiobeam	6764	6372	22(2)	[0 – 371]	26.2/26.2	1172
bitonic	56	352	3(1)	[0 – 32]	21.3/26.7	42
UTDSP						
adpcm	800	7644	8(2)	[0 – 256]	34.5/35.4	1340
compress	603	4284	12(4)	[8 – 16]	9.3/9.3	446
edge-detect	1134	716	10(4)	[3 – 128]	20.0/77.6	153
fft-1024	186	576	4(3)	[1 – 1024]	259.0/514.5	34
fft-256	90	580	4(3)	[1 – 256]	66.5/130.0	34
fir-256-64	546	200	2(2)	[1 – 256]	128.5/128.5	23
fir-32-1	33	136	1(1)	32	32.0/32.0	11
g721.marcuslee-decoder	2556	180	1(1)	2407	2407.0/2407.0	24
g721.marcuslee-encoder	2574	208	1(1)	2407	2407.0/2407.0	33
histogram	4132	300	6(2)	[64 – 256]	128.0/128.0	37
iir-1-1	29	216	0(0)	–	0.0/0.0	12
iir-4-64	73	504	3(2)	[4 – 64]	44.0/44.0	27
latnrm-32-64	73	348	3(2)	[31 – 64]	42.3/42.3	27
latnrm-8-1	47	356	2(1)	[7 – 8]	7.5/7.5	23
lmsfir-32-64	88	472	3(2)	[31 – 64]	53.0/53.0	48
lmsfir-8-1	40	284	2(1)	[7 – 8]	7.5/7.5	22
lpc	328	4092	23(2)	[0 – 320]	74.5/80.1	400
mult-10-10	232	220	3(3)	10	10.0/10.0	21
mult-4-4	64	196	3(3)	4	4.0/4.0	21
qmf-receive	8079	456	2(1)	[11 – 4000]	2005.5/2005.5	41

Name	LOC	S	$L(D)$	B	$B^{\text{avg}}_{\text{min}}/B^{\text{avg}}_{\text{max}}$	$\lvert V^C_\tau \rvert$
qmf-transmit	8078	456	2(1)	[11 – 4000]	2005.5/2005.5	41
v32.modem-eglue	507	144	1(1)	[239 – 240]	239.0/240.0	9
misc						
ammunition	2508	21704	77(2)	[0 – 4008]	581.6/639.0	1938
anagram	2722	2980	23(3)	[0 – 2279]	289.9/341.9	306
codecs-codrle1	111	772	4(2)	[1 – 128]	10.5/69.5	201
codecs-dcodhuff	640	1456	11(2)	[0 – 10280]	125.8/1061.3	358
g721-encode	899	6272	9(1)	[0 – 256]	32.3/33.7	578
g723-encode	897	6272	9(1)	[0 – 256]	32.3/33.7	578
h263	926	1672	7(2)	[40 – 1024]	205.3/205.3	208
hamming-window	62	188	3(2)	[19 – 401]	153.3/153.3	18
selection-sort	67	144	2(2)	[1 – 299]	150.0/299.0	20