Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System

Der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
Christian Werner Dietrich, M.Sc.
geboren am 5. Dezember 1989
in Rothenburg o.d.T., Deutschland

2019

Referent **Prof. Dr.-Ing. habil. Daniel Lohmann**
Korreferent **Prof. Dr.-Ing. Bernardo Wagner**
Tag der Promotion: **13. November 2019**

Abstract

Mechanical and electronic automation was a key component of the technological advances in the last two hundred years. With the use of special-purpose machines, manual labor was replaced by mechanical motion, leaving workers with the operation of these machines, before also this task was conquered by embedded control systems. With the advances of general-purpose computing, the development of these control systems shifted more and more from a problem-specific one to a one-size-fits-all mentality as the trade-off between per-instance overheads and development costs was in favor of flexible and reusable implementations. However, with a scaling factor of thousands, if not millions, of deployed devices, overheads and inefficiencies accumulate; calling for a higher degree of specialization.

For the area of *real-time operating systems (RTOSs)*, which form the base layer for many of these computerized control systems, we deploy way more flexibility than what is actually required for the applications that run on top of it. Since only the solution, but not the problem, became less specific to the control problem at hand, we have the chance to cut away inefficiencies, improve on system-analyses results, and optimize the resource consumption. However, such a tailoring will only be favorable if it can be performed without much developer interaction and in an automated fashion. Here, real-time systems are a good starting point, since we already have to have a large degree of static knowledge in order to guarantee their timeliness. Until now, this static nature is not exploited to its full extent and optimization potentials are left unused.

The requirements of a system, with regard to the RTOS, manifest in the interactions between the application and the kernel. Threads request resources from the RTOS, which in return determines and enforces a scheduling order that will ensure the timely completion of all necessary computations. Since the RTOS runs only in the exception, its reaction to requests from the application (or from the environment) is its defining feature.

In this thesis, I will grasp these interactions, and thereby the required RTOS semantic, in a control-flow–sensitive fashion. Extracted automatically, this knowledge about the reciprocal influence allows me to fit the implementation of a system closer to its actual requirements. The result is a system that is not only in its usage a special-purpose system, but also in its implementation and in its provided guarantees.

In the development of my approach, it became clear that the focus on these interactions is not only highly fruitful for the optimization of a system, but also for its end-to-end analysis. Therefore, this thesis does not only provide methods to reduce the kernel-execution overhead and a system's memory consumption, but it also includes methods to calculate tighter response-time bounds and to give guarantees about the correct behavior of the kernel. All these contributions are enabled by my proposed interaction-aware methodology that takes the whole system, RTOS and application, into account.

With this thesis, I show that a control-flow–sensitive whole-system view on the interactions is feasible and highly rewarding. With this approach, we can overcome many inefficiencies that arise from analyses that have an isolating focus on individual system components. Furthermore, the interaction-aware methods keep close to the actual implementation, and therefore are able to consider the behavioral patterns of the finally deployed real-time computing system.

Keywords — real-time operating system, interaction-aware system analysis, whole-system optimization, worst-case response time, automatic verification, worst-case stack analysis, application-specific processor design

Kurzfassung

Die von Mechanik und Elektronik getriebene Automatisierung war eine der treibenden Kräfte des technologischen Fortschritts in den letzten zweihundert Jahren. Spezialisierte Maschinen ersetzten die manuelle Arbeit und die Arbeiter waren fortan mit der Bedienung dieser Maschinen vertraut, bevor auch dieser Arbeitsbereich durch eingebettete Kontrollsysteme erobert wurde. Durch die steigende Verfügbarkeit universeller Rechenmaschinen wandelten sich diese Systeme immer weiter von problemspezifischen zu generalisierten Lösungen, da sich die Abwägung zwischen Instanz- und Entwicklungskosten zugunsten von flexiblen und wiederverwendbaren Implementierungen verschob. Dieses Bild ändert sich jedoch bei einem Skalierungsfaktor von Tausenden, wenn nicht gar Millionen, von verbauten Geräten: Die Effizienzverluste und Zuschläge generalisierter Lösungen summieren sich auf, was einen höheren Spezialisierungsgrad erstrebenswert macht.

Für den Bereich der Echtzeitbetriebssysteme (RTOS), welche die Grundlage für viele dieser Steuerungssysteme bildet, setzen wir weitaus mehr Flexibilität ein, als für die darauf laufenden Anwendungen tatsächlich erforderlich ist. Da allerdings nur die Lösungen, jedoch nicht die Steuerungsprobleme an sich, unspezifischer wurden, haben wir die Möglichkeit, Ineffizienzen zu beseitigen, die Ergebnisse von Systemanalysen zu verbessern und den Ressourcenverbrauch zu optimieren. Solche Anpassungen sind jedoch nur dann von Vorteil, wenn sie, ohne größeres Zutun von Entwicklerseite, automatisiert durchgeführt werden können. Echtzeitsysteme bilden hier einen guten Ausgangspunkt, da wir bereits über ein hohes Maß an statischem Wissen verfügen müssen, um deren Rechtzeitigkeit garantieren zu können. Bislang wird dieses statische Naturell nicht in vollem Umfang ausgenutzt und Optimierungspotentiale liegen brach.

Die Anforderungen an das RTOS manifestieren sich in den Interaktionen zwischen der Anwendung und dem Betriebssystemkern. Ausführungsfäden fordern Ressourcen vom Kern an, der im Gegenzug eine Ausführungsreihenfolge, welche die rechtzeitige Fertigstellung aller notwendigen Berechnungen gewährleistet, bestimmt und durchsetzt. Da das RTOS im Ausnahmefall läuft, ist seine Reaktion auf Anfragen von der Anwendung (oder von der Umgebung) *das* bestimmende Merkmal.

In dieser Arbeit werde ich diese Wechselwirkungen und damit die erforderliche RTOS-Semantik auf kontrollflusssensitive Weise erfassen. Automatisch extrahiert, ermöglicht mir dieses Wissen um die gegenseitige Beeinflussung, die Implementierung eines Systems näher an den tatsächlichen Anforderungen auszurichten. Das Ergebnis ist ein System, das nicht nur in seiner Verwendung ein Spezialzwecksystem ist, sondern auch in seiner Umsetzung und in den bereitgestellten Garantien.

Im Laufe diser Arbeit wurde deutlich, dass der Fokus auf Interaktionen nicht nur fruchtbar für die Optimierung des Systems, sondern auch für Ende-zu-Ende Analysen, ist. Daher erarbeite ich in dieser Arbeit nicht nur Methoden um Kernausführungslaufzeiten und den Speicherverbrauch zu reduzieren, sondern ich stelle auch Methoden zur Berechnung präziserer Antwortzeitabschätzungen und zum Nachweis des korrekten Kernverhaltens bereit. Meine vorgeschlagene interaktionsgewahre Methodik ermöglicht das gesamte System, bestehend aus RTOS und Anwendung, zu berücksichtigen.

Mit dieser Arbeit zeige ich, dass eine kontrollflusssensitive Gesamtsicht auf die Wechselwirkungen in einem Echtzeitsystem machbar und lohnend ist. Dadurch können wir Ineffizienzen überwinden, die sich aus Analysen, die sich auf einzelne isolierte Systemteile konzentrieren, ergeben. Darüber hinaus bleibe ich mit den vorgestellten interaktionsgewahren Methoden nahe an der tatsächlichen Implementierung und bin daher in der Lage, die tatsächlichen Verhaltensmuster des letztendlich verbauten Echtzeitrechensystems zu berücksichtigen.

***Schlüsselwörter* —** Echtzeitbetriebssystem, interaktionsgewahre Systemanalyse, umfassende Systemoptimierung, Antwortzeitanalyse, automatische Verifikation, Stackverbrauchsanalyse, anwendungsspezifisches Prozessordesign

Danksagungen

An dieser Stelle möchte ich all jenen Menschen meinen Dank aussprechen, die meinen Pfad, vom Startzustand zur Promotion, maßgeblich beeinflusst haben und ohne die das vorliegende Werk nicht möglich geworden wäre. Dabei will ich, wie das für eine Arbeit aus dem Bereich der Echtzeitbetriebssysteme angemessen ist, entlang der physikalischen Zeitachse vorgehen.

Mein erster, und größter Dank, gilt meinen Eltern, Rita und Gerhard, die früh mein Interesse an der Welt im Allgemeinen und an Computern im Besonderen geweckt haben; eine bessere Basis, um meiner Neugier nachzugehen, hätte ich mir nicht wünschen können.

Jochen Rössner und Stefan Siegl möchte ich für die vielen Nächte am Gerät und an der Tastatur danken; ohne Sie wüsste ich wohl bis heute nicht, dass jeder der `malloc()` sagt, auch `free()` sagen muss, und dass Kritik am Werk nicht zwingend eine Kritik am Autor ist.

Christoph Egger möchte ich dafür danken, dass er mich in die Fänge des Lehrstuhls 4 gelockt hat, wo Reinhard Tartler mir früh allergrößte Freiheiten zustand und mich bei meinen ersten Schritten im wissenschaftlichen Betrieb angeleitete. Mit Martin Hoffmann war mir, nach dem Wechsel zu den flüchtigen Fehlern, ein weiterer hervorragender Lehrer, der mir größtes Vertrauen entgegenbrachte.

Einen besonderen Dank möchte ich meinem Doktorvater Daniel Lohmann aussprechen. Ohne ihn hätte ich den roten Faden wohl nur im Aquarium, aber nie in der Ansammlung meiner wilden Ideen und meinem Schaffen gesehen. Das Abenteuer, auf welches er mich mitnahm und dass auf einer Fahrt durch den Thüringer Wald begann, dauert bis heute an und ich freue mich auf die kommenden Jahre, ebenso konstruktiver und vertrauter, Zusammenarbeit.

Auch Danken möchte ich meinen Kommilitonen, Kollegen, zahlreichen Koautoren und Studierenden für die unzähligen Stunden angefüllt voller interessanter Diskussionen. Mit Valentin und Andreas teilte ich Monokel, mit Bernhard wurde ich “fertig” und mit Peter und Peter beschritt ich systemweite Pfade. Am SRA, gaben mir Jürgen, Monika, Lars, Oskar, Gerion, Björn, Florian, Tobias und Stefan eine neue Heimat fern von fränkischer Scholle.

Während der ganzen Zeit hielt mich Maximilian Krüger mit Ofengemüse und hervorragendstem Bier am Leben und hatte immer ein offenes Ohr für meine Sorgen und Nöte. Mit ihm getraute ich mich auf höchste Gipfel und durchlebte die längsten Nächte.

Hannover, November 2019

Table of Contents

1 Introduction 1

1.1 Motivation . . . 3

1.2 Research Context of this Thesis . . . 5

1.3 Purpose of this Thesis . . . 6

1.4 Structure . . . 7

1.5 Typographical Conventions . . . 8

2 Fundamentals – Background and Context 9

2.1 Real-Time Computing Systems . . . 11

2.1.1 Real-Time System Concepts . . . 12

2.1.2 Event-Triggered Real-Time Operating Systems . . . 14

2.1.2.1 Activities: Interrupt Handlers and Threads . . . 15

2.1.2.2 Control: Job Release and Directed Dependencies . . . 17

2.1.2.3 Synchronization: Mutual Exclusion . . . 18

2.1.3 The OSEK Operating-System Standard . . . 19

2.2 From the Real-Time to the Implementation Domain . . . 22

2.2.1 Ambiguous Mapping of Real-Time Concepts . . . 23

2.2.2 RTOS API Flexibility . . . 25

2.2.3 Impact of Implementation Side-Effects . . . 26

2.2.4 Analysis on the Implementation Level . . . 27

2.3 Chapter Summary . . . 27

3 Foundation – Fine-Grained Interaction Knowledge 29

3.1 Introduction . . . 31

3.2 State of the Art . . . 32

3.3 Static-Analysis Techniques for Real-Time Systems . . . 34

3.3.1 Control-Flow Graph . . . 34

3.3.2 Atomic Basic Blocks . . . 36

3.4 Static State Transition Graph . . . 39

3.4.1 Abstract System State . . . 39

3.4.2 System-State Enumeration . . . 41

3.4.3 The State-Space Explosion Problem . . . 44

3.4.3.1 Constraints from the Real-Time Domain . . . 44

3.4.3.2 Simplification of the Application Model . . . 45

3.5 Global Control Flow Graph . . . 47

3.5.1 Code-Section–Based State Collapsing . . . 48

3.5.2 System-State–Flow Analysis . . . 49

3.6 Limitations of the Interaction Analysis . . . 54

3.7 dOSEK: A Framework for Whole-System Analysis . . . 55

3.8 Chapter Summary . . . 56

I Analysis 57

4 SysWCET and SysWCEC – Whole-System Analyses 59
- 4.1 Whole-System View on Consumable Physical Resources . . . 61
- 4.2 Compositional WCRT Analysis Considered Harmful . . . 61
- 4.3 Related Work . . . 64
- 4.4 Implicit Path Enumeration Technique . . . 66
- 4.5 Response-Time IPET Construction . . . 68
  - 4.5.1 The SSTG Subgraph of the WCRT Problem . . . 69
  - 4.5.2 The State Layer: Constraining State Counts . . . 70
  - 4.5.3 The Glue Layer: Deriving ABB Counts . . . 72
  - 4.5.4 The Machine Layer: Integrating the Application and Kernel Code . . . 74
  - 4.5.5 System- and Thread-Level Flow-Fact Constraints . . . 76
- 4.6 Comparison to Compositional WCRT Analysis . . . 77
  - 4.6.1 Evaluation Scenario . . . 78
  - 4.6.2 Micro Benchmarks . . . 79
  - 4.6.3 *i4*Copter: A Realistic Task Set of a Safety-Critical Real-Time System . . . 82
  - 4.6.4 Discussion . . . 84
    - 4.6.4.1 Scalability of SysWCET . . . 84
    - 4.6.4.2 Limitations to the Applicability . . . 85
    - 4.6.4.3 More Complex Hardware Models . . . 87
- 4.7 Application to Worst-Case Energy Consumption . . . 87
  - 4.7.1 System-State Dependent Energy Consumption . . . 88
  - 4.7.2 SysWCEC – SysWCET for Energy Consumption . . . 89
  - 4.7.3 Qualitative Insights from SysWCEC . . . 91
- 4.8 Summary . . . 92

5 Automated Kernel Verification 95
- 5.1 Problem Field and Related Work . . . 97
- 5.2 Automated Application-Specific Verification . . . 98
  - 5.2.1 Verification of the Interaction Model . . . 99
  - 5.2.2 Dynamic Exploration of the State Space . . . 102
- 5.3 Experiments . . . 104
  - 5.3.1 Positive Tests . . . 104
  - 5.3.2 Negative Tests and Fault Injection . . . 105
  - 5.3.3 Discussion . . . 106
- 5.4 Summary . . . 107

II Optimization 109

6 Semi-Extended Tasks – Stack as a Shared Resource 111
- 6.1 Memory Consumption of Statically-Allocated Stacks . . . 113
- 6.2 Stack-Space Saving in the Literature . . . 116
- 6.3 Hybrid Thread Execution on Two Stacks . . . 119
  - 6.3.1 A Mechanism for Intra-Thread Stack Switch . . . 120
  - 6.3.2 Fine-Grained Worst-Case Stack Consumption Analysis . . . 122
  - 6.3.3 Selection of Stack-Switch Functions . . . 125
- 6.4 Experimental Evaluation . . . 128

6.4.1 Benchmark Generation 128
6.4.2 Evaluation Scenario and Method 129
6.4.3 Run-Time and Scalability of the Optimization 130
6.4.4 Comparison with Basic-Task Systems 132
6.4.5 Discussion 136
6.5 Summary 139

7 OSEK-V – An Application-Specific Processor Pipeline 141
7.1 Problem Field and Related Work 143
7.2 System-State Machine as Executable Interaction Model 145
7.3 The OSEK-V Processor 148
7.3.1 Pipeline Integration of the SSM 148
7.3.2 External System-Activation Patterns 150
7.3.3 System Generation and Startup 151
7.4 Experimental Evaluation 151
7.4.1 Evaluation Scenario 151
7.4.2 FPGA Synthesis Costs for the OSEK-V Core 152
7.4.3 Run-Time Latencies 153
7.4.4 Discussion 155
7.5 Summary 157

8 Summary, Conclusions, and Further Ideas 159
8.1 Summary of the Thesis 161
8.2 Further Ideas 163
8.3 Research Questions 164
8.4 Conclusion 165

Lists 167
Glossary 167
Bibliography 171
List of Figures 195
List of Tables 197
List of Listings 199
Lebenslauf 201

1

Introduction

The White Rabbit put on his spectacles. “Where shall I begin, please your Majesty?” he asked. “Begin at the beginning,” the King said gravely, “and go on till you come to the end: then stop.”

Alice’s Adventures in Wonderland, 1865, LEWIS CARROLL

Related Publications

[▷Fie+18] Björn Fiedler, Gerion Entrup, **Christian Dietrich**, and Daniel Lohmann. “Levels of Specialization in Real-Time Operating Systems.” In: *Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT ’18)* (Barcelona, Spain). July 2018.

1.1 Motivation

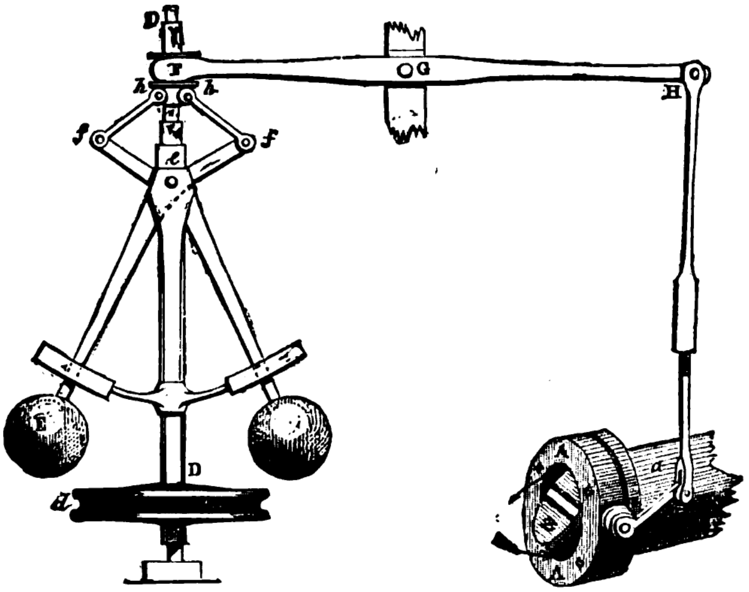


Fig. 4.—*Governor and Throttle-Valve.*

Figure 1.1 – Centrifugal Governor. A continuous control system to regulate the pressure in a steam engine; the faster the governor spins, the higher rise the two balls, and the lower becomes the steam flow. While James Watt already used the control principle in his steam engine in 1788, different engineers mapped the principle to the technology of the day. Illustration from [Rou81, p. 6].

Mechanical and automated control of technical processes was one of the enabling factors of the industrial revolution [Lan73, p. 52]. In 1788, James Watt[1] used the *centrifugal governor* (Figure 1.1) for his improved steam engine to keep the rotational speed of the engine uniform [Rou81, p. 6]. A belt connects engine and governor such that both spin with the same rotary motion. The faster the governor spins the higher rise the two metal “fly balls”, which leads to an opening of the throttling valve. Thereby, an increasing stream flow is subtracted from the engine’s steam input and a constant steam current fuels the engine.

The centrifugal governor is a perfect example of a dedicated control system with real-time requirements. Its only task is to translate the rotation speed of the engine into a regulated steam current fast enough that the engine runs with the desired conversions. If the governor fails to react on a pressure increase in time, because of a slipped or broken belt, the input increases, the engine accelerates, and even an explosion becomes possible.

Engineers have built such dedicated systems as special-purpose systems for a long time. They designed and optimized these systems for a specific environment as they must (and often only can) fulfill a single task. While this specificity allows for a high tailoring potential of the system towards the desired use case, it comes also with the *imperative* to specialize it. For a machine to be effective and efficient, the mechanical engineer must combine, connect, and adjust different parts into a

---

[1] Although, James Watt is often cited as the inventor of the centrifugal governor, Christiaan Huygens already described it 1673 in connection to windmills [Bat45].

dedicated machine with a limited set of use cases. Therefore, these special-purpose systems came with high engineering and integration costs.

The age of general-purpose machines took its first glimpse when Charles Babbage proposed the Analytical Engine as the first memory-programmable computer [Bab64, cha. 8]. It took another 70 years, until Alan Turing showed that such a machine could calculate everything that is computable [Tur36]. Once designed and built, a computer can fulfill different, even multiple, tasks without changing the physical design of the machine. In control systems, computers revolutionized the field as they can implement every computable control function if connected to the appropriate *sensors* and *actors*. Therefore, computers replaced the mechanical and pneumatic control systems of the industrial age steadily and with an increasing pace.

ARM alone declared that they shipped a 100 billion chips from 1991 until 2017 [Hug17], with half of them being sold in the last 4 years. Over 85 percent (Classic ARM, Cortex-M, Cortex-R) of these are produced for the embedded market, where the computer is not the product itself but only part of a larger stand-alone product. In terms of shipped units, the embedded market is the largest segment for chip vendors [Ten00], although it remains mostly invisible for the customer. And with the predicted and promised advent of the Internet of Things, we will scatter even more general compute power all over the physical world.

However, with the decreasing pressure to build special-purpose mechanics to solve an engineering problem came the illusion that tailoring is no longer necessary. We connect highly dynamic, and even distributed, computer systems to the physical world and use programming languages with a high abstraction level to express our ideas. While these abstractions foster the productivity, the time–to–market, and allow for the dynamic self-adaption of systems, they come at a cost. The more complex and dynamic a system gets, the harder becomes a verification of its *functional properties*, like correct operation or timeliness of the reactions. Furthermore, flexibility and run-time adaptability can have a negative impact on the system's *non-functional properties*, like memory usage or energy consumption.

However, the underlying problem did not become less specific just because we solved them with adaptable, dynamic, and easily reusable machines. Therefore, we often over fulfill the actual requirements of the problem and use the capabilities of the employed compute systems only partially. This huge, wasted potential is a thorn in the eye for market segments that ship a high number of units and are, therefore, sensitive to higher-than-necessary per-unit costs.

A prime example for a price sensitive sector is the automotive industry. Every modern car entails around a hundred control units with at least one processor [Cha09]. If Volkswagen could reduce the cost of only one of these processors in every sold car by a single cent, the company's profit would have been 107 000€ higher in 2017 [Akt17].

However, the price-per-unit is not the only factor, but we also must consider the cost for the engineering. Unlike mechanical systems, a computer can more easily choose at run time the action that should be performed. So, even if the concrete system always chooses the same action, the engineer can include, without much thought, more functionality and postpone the decision to the run time. While a centrifugal-governor valve with a Gardena plug is ridiculous even if "it always comes with one", this reflects the common practice in writing software. Manual tailoring is often avoided, since it is too laborious and has the potential to introduce more bugs due to the complexity of the software stack. Therefore, it is crucial to *automate* the analysis and the tailoring of software systems towards the actual application requirements.

An area, where the potential for automated, in-depth analysis and tailoring is especially high, are *real-time computing systems (RTCSs)*. Since we have to guarantee that the RTCS reacts to some stimuli within a given timing bound, we already must have a large amount of knowledge about the system to proof this ahead of time. Even more, often all software components, like *real-time* *operating system (RTOS)* and *real-time application (RTA)*, are inseparably combined into a single system image, which is then loaded onto the *microcontroller unit (MCU)*. If an update becomes necessary, we generate a new system image and replace the old one entirely. This tight combination on the implementation level provides us with a *closed world assumption* and eases system analysis and tailoring.

However, the development of RTOS and application is often done in isolation by different engineering teams, or even different companies. Therefore, the interweaving of both components still remains limited and often a *common-of-the-shelve (COTS)* RTOS is used with only minor adaptations, like a different coarse-grained feature selection. Thereby, we waste the potential for stricter guarantees on functional properties and for improved non-functional properties.

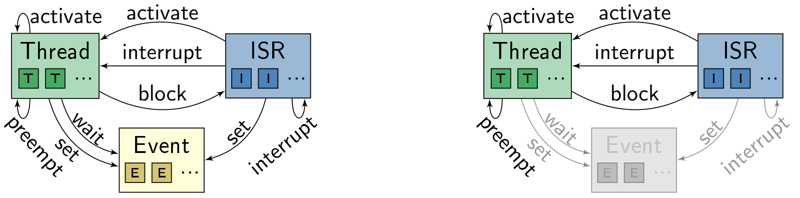
In order to ease these problems, we must get a better understanding of the concrete application and what functionality is actually requested from the system software. I aim to achieve this better understanding by a flow-sensitive *interaction analysis* of RTOS and application. With the resulting whole-system view, I want to achieve a better tailoring of the RTOS towards the actual application requirements and give a better quantification of the actual implementation properties (e.g., response times or stack consumption).

1.2 Research Context of this Thesis

This thesis takes place in the context of the AHA research project (DFG LO 1719/4-1, "Automated Hardware Abstraction in Operating Systems"). There, we apply automated use-case requirement analysis and specialization to whole software stacks (application, operating system, and hardware) with a special focus on the system software. It is the project's objective to find the highest possible degree of software- and hardware specialization that is possible, beneficial, and desirable. For this, we [▷Fie+18] developed a taxonomy that is based on the notion of *interaction* to classify different levels of system-software specialization.

specialization

First of all, we define what we mean by *specialization* (of system software): Specialization is an adaptation of the hardware platform, the operating system, or the application code, which must meet three necessary conditions: (1) The specialized system exposes the same observable behavior (like scheduling order) from the applications point of view. (2) The specialized system eases the verification of the functional properties or shows improved non-functional properties over the unspecialized system. (3) The specialized RTOS–hardware combination loses the ability to produce the correct observable behavior for other valid application.



(a) General Specification

(b) Specialized RTOS

Figure 1.2 – Interaction Graph for General and Specialized RTOS. The specialized RTOS is no longer able to fulfill its function for applications that require events. Adapted from Fiedler et al. [▷Fie+18].

interaction graphs

In Fiedler et al. [▷Fie+18], we stated the third condition more precisely by introducing the concept of *interaction graphs*. In these graphs, we connect RTOS abstractions (or instances thereof), like threads, such that the edges indicate potential interactions, like preemption or wait-for relations, between them. For example, Figure 1.2a shows the most general interaction graph for an example RTOS specification with threads, *interrupt-service routines (ISRs)*, and events. Here, threads can wait passively for an event until another thread or an ISR signals the event. If an RTOS–hardware stack supports all specified interactions and the unlimited creation of system objects, it fulfills the specification and can execute every possible application.

However, most applications do not need an unlimited number of instances and they do not invoke all possible interactions. For example, an application that never creates an event will never wait passively and no wake-up signal can ever be sent. If we disable event support (see Figure 1.2b), the compiled RTOS still works perfectly for our application, but it does not meet the full specification; applications with events will not work correctly. Summarized, a specialized RTOS instance exhibits only a subset of the full interaction graph that the RTOS specification prescribes.

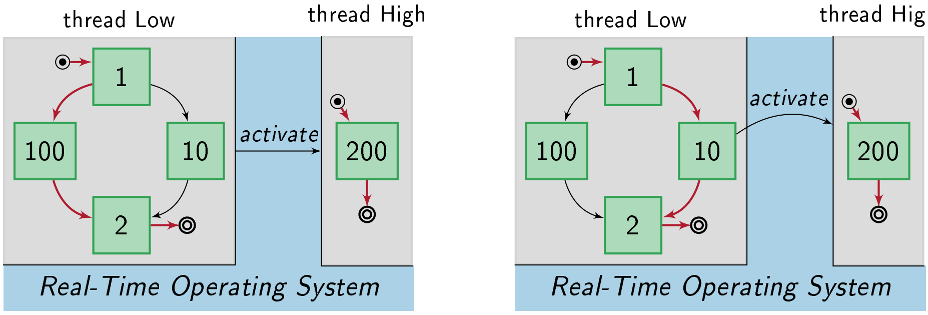
control-flow sensitivity

In this thesis, I investigate on a detailed variant of interaction graphs that grasps requirements with a high precision, but is also highly specific to a single application: the *control-flow–sensitive interaction graph*. This variant does not only describe how abstractions interact with each other but it goes down to the source-code and implementation level. While the interaction graph from Figure 1.2b excludes waiting in general, a control-flow–sensitive interaction graph can, for example, expel waiting when a thread currently executes a specific function.

1.3 Purpose of this Thesis

To illustrate the benefits of a control-flow–sensitive interaction analysis, I will anticipate the integrated response-time analysis from Chapter 4. In a nutshell, the *worst-case response time (WCRT)* is the longest time a system takes to react upon some event, like an external interrupt. This duration covers the execution time of all necessary and interfering tasks, all interrupts, and all hardware-induced delays.

In Figure 1.3a, we see a system that consists of two threads and their *control-flow graphs (CFGs)*, where we annotated each code block with its *worst-case execution time (WCET)*. Furthermore, we know that the low-priority thread activates (at some point) the high-priority thread and the RTOS



(a) Flow Insensitive **(b)** Flow Sensitive

Figure 1.3 – Example for Response Time Analysis. The end-to-end response time for thread low, which can activate the high priority thread, benefits from the flow-sensitive analysis of the interaction between RTOS and application. When having only flow-insensitive information, we must give an upper bound of 303 cycles, with flow-sensitive information we can sharpen this bound to 213 cycles.

will immediately preempt Low in favor of High. For Low's WCRT, we accumulate the duration of the longest execution path in Low (103 cycles) and the longest execution path in High (200 cycles).

However, in a more detailed picture of the system (see Figure 1.3b), we see that Low activates High only in the right branch of its condition statement. We know that the interaction (activate), which the RTOS mediates in form of a context switch, is only invoked in a certain flow-sensitive context. Therefore, we can give a much tighter bound for the WCRT (200 + 13 cycles) as the worst-case path from flow-insensitive situation cannot occur in reality.

The problem of the flow-insensitive variant, which leads to the more pessimistic WCRT estimation, is the isolated execution-time analysis of each thread. After we pessimistically calculate an upper bound for the execution time of each thread, we pessimistically accumulate it into an end-to-end WCRT. Here, the flow-sensitive interaction analysis is able to overcome this segregation between threads and allows for an integrated WCRT.

In this thesis, I will discuss the benefits of (control) flow-sensitive interaction analysis that spans multiple execution threads for real-time systems. Thereby, I will answer the following research questions:

Research Question 1 *Is a control-flow sensitive view on the RTOS–application interaction feasible for whole-system analysis?*

Research Question 2 *What analysis and run-time inefficiencies arise in the real-time application from its segregation into distinct execution threads?*

Research Question 3 *What beneficial non-functional RTOS properties can we achieve, and what statements about functional properties can we make, if we have an integrated view on the real-time computing system?*

1.4 Structure

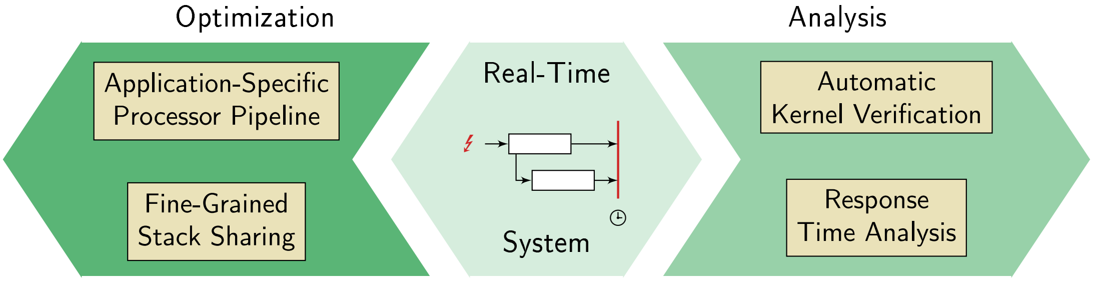

Optimization
Application-Specific Processor Pipeline
Fine-Grained Stack Sharing
Real-Time
System
Analysis
Automatic Kernel Verification
Response Time Analysis

Figure 1.4 – Structural Overview of this Thesis

For this thesis, I investigated and validated my approach in four different projects (see Figure 1.4) that make use of flow-sensitive interaction graphs. These projects are collated in two parts that focus on the *analysis* of the RTCS and its *optimization*. Thereby, I answer the stated research questions cross-cutting to these parts.

Chapter 2 *Fundamentals – Background and Context* (pp. 9–28)
In this thesis, I analyze the interaction between application and operating system in *event-triggered real-time systems*. Therefore, I provide an introduction to these systems, describe the used system model, and discuss the mismatch between the view of real-time and implementation engineer.

Chapter 3 *Foundation – Fine-Grained Interaction Knowledge* (pp. 29–56)
In this chapter, I recap the foundation of the control-flow–sensitive interaction analysis and describe different methods to grasp the RTOS–application interaction. As analysis result, I describe two application-specific, flow-sensitive interaction models.

Part I *Analysis* (pp. 59–108)
In the analysis part, I use the interaction model to improve other static analyses of the RTCS to give better (non-)functional guarantees for the examined system.

Chapter 4 *SysWCET and SysWCEC – Whole-System Analyses* (pp. 59–93)
As first analysis chapter, I present the SysWCET approach as an integrated worst-case response time analysis that spans from the machine-code level to the scheduling analysis. Furthermore, I show how this method, which was developed for timing analysis, can also improve the estimation of the worst-case response energy consumption.

Chapter 5 *Automated Kernel Verification* (pp. 95–107)
As second analysis chapter, I use the interaction graph to verify that a given kernel binary exhibits the correct RTOS behavior. Thereby, the verification is done specifically for a given application and does not rely on a semantic source-code analysis.

Part II *Optimization* (pp. 111–159)
In the optimization part, I use the application-specific interaction model to constructively improve the non-functional properties of the RTCS implementation, like jitter and memory usage.

Chapter 6 *Semi-Extended Tasks – Stack as a Shared Resource* (pp. 111–140)
As first optimization of a non-functional property, I present a method to share stack-space memory efficiently between different blocking threads. For this, I develop a fine-grained method for finding upper bounds on the worst-case stack consumption and use a genetic algorithm to find a good system configuration.

Chapter 7 *OSEK-V – An Application-Specific Processor Pipeline* (pp. 141–158)
In this chapter, I push RTOS specialization to its limits by replacing the RTOS implementation with a behavioral-equivalent finite-state machine. Integrated into an application-specific processor pipeline, the resulting system exhibits low kernel run-time overheads with a low jitter.

Chapter 8 *Summary, Conclusions, and Further Ideas* (pp. 159–165)
In the last chapter, I summarize my contributions, point out further research ideas, and give a conclusion to my research questions.

1.5 Typographical Conventions

For citations, an open triangle indicates that I was the main author or one of the co-authors (e.g., [▷DHL17]). Newly introduced terms are highlighted in *italic* and the margin notes provide a course guideline of the touched topics. Functions and program variables are set in with a mono-space font (`function()`, `variable`). If I show figures or results from previously published articles, I use the phrase “Adapted from [citation]” to indicate a modified version from the original source, and “From [citation]” to indicate an unmodified one.

2

Fundamentals

Background and Context

> It is often asserted that discussion is only possible between people who have a common language and accept common basic assumptions. I think that this is a mistake. All that is needed is a readiness to learn from one's partner in the discussion.
>
> *Conjectures and Refutations*, 1963, KARL POPPER

In order to ensure its timeliness, the real-time engineers model, and often develop, a system within the concepts and the abstractions of the real-time domain. These concepts have a heavy influence on the structure of real-time applications and on the services that a real-time operating system has to provide. Therefore, I will present an introduction to these concepts and to interaction patterns that arise on the level of the real-time model.

However, it is not this model of the real-time system that gets deployed onto the actual execution platform, but one concrete implementation. While the model narrows the focuses on the real-time analysis, the actual implementation has to solve engineering problems and hardware limitations, and can therefore lead to diversions from the intended interactions. Therefore, I will discuss this mismatch and argue for the necessity of implementation-centric analyses that cover the actual interactions between RTOS and application.

2.1 Real-Time Computing Systems

In general, a *real-time system (RTS)* is a technical system whose correctness depends not only on the correct outputs but also on their timeliness [Mar65; LRG95]. In particular, every system that must react correctly to an external event within a (guaranteed) bounded amount of time is a real-time system. And while a functionally-correct implementation and working I/O interfaces are already sufficient for the correct results, their timeliness is much harder to ensure. Thereby, the word prefix "real-time" does *not* make a statement about the speed of the system's operation, but only about the maximal distance of triggering event and corresponding reaction on the physical-time (the *real* time) axis. Hence, the timeliness of a RTS not only depends on its implementation but is also heavily influenced by the operation environment.

It is this dependence on the timeliness that lifts the reaction time from a *non-functional property* to a *functional property* of the system. While a functional property fulfills a functional requirement that is essential to the logical correctness of a system [CP09], non-functional properties are about the quality of the provided system service. Thereby, the classification is not inherent to a certain property, but it depends on the environment: Where the reactiveness of a graphical user interface is a (desirable) quality of a web browser, it becomes an (essential) functional property for the user interface of a fighter yet if is implemented with HTML5 and CSS.

real-time computing system

While computer scientists often believe that every RTS contains a computer, we have seen in Chapter 1 that also mechanical and pneumatic systems are subject to real-time constraints. Therefore, we make a distinction between the RTS and the embedded RTCS [Kop11] that calculates the control decisions. Figure 2.1 gives an overview how the different concepts are enclosed into each other. The whole system, which measures and controls the environment in a timely fashion, is the *real-time system*. The *real-time computing system* executes the required control task with the help of one or multiple processors, which are connected via sensors and actors to the environment. The *real-time application (RTA)* is a program that implements one or multiple control tasks in software. The *real-time operating system (RTOS)* manages the execution of the RTA and exposes a system-service interface to provide access to the hardware and to manipulate the executor mechanism. As this thesis focuses exclusively on systems with a RTCS, I will use the terms RTCS and RTS interchangeably.

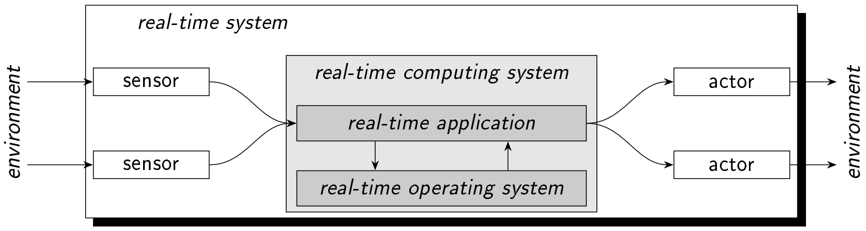


Figure 2.1 – Real-Time System

There are at least three different lenses under which computer-implemented RTSs are viewed: control theorist, real-time (scheduling) analyst, and implementation engineer. From a high-level perspective, these three points of views reflect the development phases of a RTCS in a waterfall development model [Roy70]. The control theorist grasps the physical dynamics of the controlled (external) object and chooses an appropriate controller (e.g., a PID controller [Min]). Since controllers are normally designed in the continuous domain; the theorist provides a time-discretized implementation, which the RTCS must invoke periodically. The real-time analyst is confronted with one or multiple software-implemented controllers and their timing requirements. Under an appropriate task model (e.g., the sporadic task model [Mok83]), he chooses task parameters (e.g., priority), and makes a statement about the *schedulability* of the system: Is it possible to meet every deadline for every task? The implementation engineer takes the RTA and maps the task-model concepts to the available RTOS abstractions, like threads and semaphores. With every step, we decrease the level of abstraction as we come closer to the actual implementation on the deployed hardware. However, we also come closer and closer to the eventually operated RTCS and, therefore, the actual run-time behavior.

As this thesis focuses on the interface between real-time analysis and implementation engineer, I will give an overview of commonly used real-time terminology and its mapping to RTOS concepts and primitives. For this, I will stick to the Burns standard notation [Dav13] as close as beneficial and base the description on the sporadic task model [Mok83]. Table 2.1 gives an overview over of the most important concepts.

2.1.1 Real-Time System Concepts

tasks and jobs

In digital control systems, values cannot be calculated continuously, but we must discretize the calculation into work packages, called *jobs*. Upon some external event, a job J_i is released at a certain time t_r and must be completed before its relative *deadline* D_i is elapsed at $t_r + D_i$. Since the released jobs are highly regular, we use templates in form of *tasks* for instantiating jobs. Furthermore, all jobs that are instances of the same task τ_i inherit their relative deadline, and other real-time properties. The entirety of tasks within an RTCS constitutes its *task set* τ.

hard real-time

One of the most important distinction for RTCSs is the criticality of deadline misses. According to this characteristic, we distinguish between *soft*, *firm*, and *hard* real-time systems [HR95]. For a hard RTS, it is catastrophic, in the sense of extremely high costs, to miss just a single deadline. Therefore, a multitude of theoretical analyses methods must be applied before the deployment to guarantee the timely execution of all jobs. The classic example of a hard real-time system is the break controller of a car. There, a missed deadline in a critical situation can provoke a crash and might lead to (deadly) injuries.

A less strict class is the firm RTS, where the result of a job becomes useless immediately after its deadline has passed without completion. An example for a firm RTS is a robot at an assembly line that assembles the current work piece incorrectly if it cannot calculate its movement trajectory in time. However, with specific task models, like the “(m,k) firm deadlines”-model [HR95], we still can make guarantees about the quality of service.

	Parameter	Description
τ	task set	set of all tasks that a RTCS executes
τ_i	task	static description for a class of jobs; each task release instantiates a job; *sporadic* and *periodic* tasks
p_i	period	time between two releases of a periodic task
I_i	minimum interarrival time	minimum time between two releases of a sporadic task
C_i	worst-case execution time	longest computation time of a released job if it is executed without interference
J_k	job	one instance of one task, released at time
D_k	relative deadline	maximal time between release and completion of a job

Table 2.1 – Important Real-Time System Parameters

The most forgiving class are soft RTSs. Their quality of service degrades if deadlines are missed in operation and there is no discrete point in time when the utility function of a job result drops to zero. For example, the experience of computer games degrades if the game logic misses more and more frames and the player gets continuously more frustrated. However, as long as the game remains playable it fulfills its purpose.

From the reflection on different timing-strictness classes, we see that RTCS elevate the otherwise non-functional property "timeliness" increasingly to a functional property. A hard RTS that does not meet its deadlines is not a functionally correct system.

dependencies

It is the chore of the RTOS to schedule the released jobs onto the available processors such that all deadlines are met. However, often the deadline of a job is not the sole constraint the RTOS must obey, but a job can also *depend* on the execution of other jobs. While this was explicitly excluded in early task models [LL73; Mok83], mechanisms to include task dependencies were developed over time [CL90; Bak91; HKL91; Aud+92].

Thereby, we can distinguish between *directed* and *undirected* dependencies [Sha+04]. Both of them express that the execution of two tasks are not independent and their relationship has to be considered by the real-time analyst. A directed dependency, or *precedence*, between two tasks ensures that the dependent job is executed only after the dependee has finished [Aud+92]. With *directed acyclic graph (DAG)* task models [Sti+11; Bar10], dependencies were introduced on the intra-task level. However, a general understanding of inter-task precedence is still an open topic of research [Sha+04].

An undirected dependency expresses *resource sharing* constraints between two tasks and prohibits the interleaved execution of their jobs. Such mutual-exclusive constraints can, for example, arise if both tasks access the same object or memory region. However, such mutual-exclusive constraints can be problematic in the context of real-time systems as waiting times must be bounded [SRL90a] and deadlocks avoided [Bak91]. While these undirected dependencies can be expressed on the task level [WS99], they are normally declared programmatically between fine-grained sections of task code [SRL90a; Bak91].

If we add dependencies to a RTCS, we can use them as a distinguishing feature between tasks: While *complex tasks* have at least one dependency on another task, *simple tasks* have no dependencies and can, therefore, be scheduled with considerably fewer constraints [Kop11, cha. 9.2].

task activation

Until now, we have ignored the causes that lead to a job release. We have already discussed that the control theorist hands a discretized implementation of the controller to the real-time analyst for periodic activation. Therefore, RTCSs provide the concept of *periodic tasks*, which are released with period p_i and an offset ϕ_i, to cover such application patterns [LL73; BB97].

However, as not all jobs fit into this fixed grid of periodic tasks, RTCSs also provide *sporadic tasks* [Mok83]. For a sporadic task, we do not know the exact release time, but only a minimum interarrival time i_i. As the period p_i of a periodic task is its minimum and maximal interarrival time, the sporadic-task model is a generalization of the periodic task model. However, it is still useful to distinguish these two task types as they are often mapped differently to RTOS abstractions.

worst-case execution time

The task set and its inter-task dependencies describe the logical structure of the RTCS. However, the real-time analyst also has to consider the work-load that this structure carries. For this, she quantifies the computational requirements of each task in its WCET C_i; the longest job execution time. While the actual WCET is often hard to obtain [Wäg+17], we normally use a statically-derived safe upper bound. Thereby, the WCET does explicitly not include delays that stem from preemptions or interference with other jobs. Unlike the other task parameters, the WCET is hardware dependent and must capture the influence of processor pipelines, caches, and branch predictors on the task code. We will come back to the WCET analysis topic in Chapter 4.

2.1.2 Event-Triggered Real-Time Operating Systems

The concepts from the last section (Section 2.1.1) are abstractions that the real-time analyst uses as a *model* of the RTA. She adheres to the chosen task model and captures the structure and the real-time properties of the system. In the *scheduling analysis*, the analyst derives other real-time properties (e.g., priorities) and makes guaranteed statements about the system's timeliness.

As next step in the product development, the implementation engineer maps the RTA onto a concrete hardware in such that the system adheres to chosen scheduling and task model. For this, about 67 percent of the embedded developers make use of some kind of OS or RTOS [AEe17][2]. An RTOS, or real-time extension to an existing OS, provides system services and interfaces that allow the implementation of the RTA. Although many open-source and commercial RTOSs exist, most provide similar abstractions, like threads and interrupts. For this thesis, I will stick to the notation and terminology of OSEK [OSE05] systems, since my system model, which I will describe in Section 2.1.3, is also based on this RTOS standard.

The most basic distinguishing dimension of RTOSs is the *real-time architecture* [Sch11, cha. 3.4]. The two most prominent representatives for *real time (RT)* architectures are *time-triggered* RTOSs and *event-triggered* RTOSs. The two differ in their scheduling time: time-triggered systems schedule offline; event-triggered systems schedule online at run-time. As this thesis concentrates on event-triggered systems, I will only briefly outline the time-triggered RTOS architecture to highlight the decisive features of event-triggered systems.

Time	Action
0	Start and switch to thread τ_1
4	Start and switch thread τ_2
6	Enforce deadline $D_2 = 2$ for τ_2
6	Resume to τ_1
8	Enforce deadline $D_1 = 6$ for $\tau_1 s$

Table 2.2 – Example for a Time-Triggered Scheduling Table

time-triggered

The central data structure for a time-triggered RTOS is its scheduling table, which describes what action should be taken at a specific point in time [BS88; ATB93]. Therefore, they are also referred to as *table-driven* RTOSs. As *all* scheduling decisions are known before run time, the timeliness of the system can be ensured by analyzing the scheduling table and the task's WCETs. Therefore, the highly safety-sensitive civil avionics defaults [Pri08] to use table-driven RTOSs, like ARINC 653 [AEE03], which also sparked interest in the space community [DR05].

The scheduling table links relative time offsets to actions like a thread switch or the dynamic enforcement of a deadline. Table 2.2 shows an example of such a scheduling table. There, two threads with a combined WCET of 8 time units are executed interleaved. At run time, a hardware-triggered timer ISR carries out the action of one table row and configures the timer for the next table row. If the execution reaches the end of the table, it wraps around and starts over in the first row. As no other RTOS mechanisms are provided at run time, the table must be constructed such that all directed and undirected dependencies are fulfilled.

Besides the good analyzability, such a table-driven mechanism is also simple to implement and results in small run-time overhead. However, the generation of scheduling tables is considered a hard problem [LW82] and the tables quickly become large. Therefore, not only optimal strategies for table generation were proposed [XP90; AS99], but also heuristics [CK88].

---

[2] For the remaining 33 percent, 86 percent said that the application is simple enough to not require OS services.

Compared to the time-triggered architecture, event-triggered systems avoid the large tables by delaying all scheduling decisions to the run time. There, an online scheduler decides on certain scheduling events, like a system call or an interrupt, which thread should be executed next according to the scheduling strategy. Therefore, event-triggered systems allow for lower reaction times as an important thread can be dispatched immediately upon some external event instead of waiting for its table row to be executed.

Another benefit of event-triggered systems, which might be another reason for their popularity [AEe17], are their OS abstractions and their interface. They often resemble the well-known system-call interfaces of general-purpose operating systems. In the case of the POSIX operating-system standard, event-triggered real-time extensions were even integrated into a general-purpose OS specification [98].

In the following, I will describe the important abstractions that an RTOS must provide to implement an event-triggered real-time system. Thereby, I will sketch how we map the concepts from the real-time domain onto the RTOS abstractions.

2.1.2.1 Activities: Interrupt Handlers and Threads

One of the core chores of operating systems is the *multiplexing* of limited hardware resources. Multiple applications or other users require the operation of a single hardware component, which often cannot be shared between different users at the same time. Thereby, the processor is the most basic component to allocate and multiplex, as only running software can ask for more resources. Therefore, *timesharing* operating systems were invented early on [CMD62; Org72; RT74].

threads

In these systems, the basic entity of processor-time allocation is the *thread*. In order to service multiple threads on a single CPU, the RTOS switches between these threads and, thereby, performs a time-division multiplexing of the CPU. We call the selection of the currently running thread *scheduling* and the enforcement of this decision the thread *dispatching*.

In its core, a thread is a control flow whose execution is managed by an operating system; without an OS, there can be no threads. Thereby, a control flow is an active entity that executes on a processor and is constituted by a program counter, which points into the program memory, and some execution context. For most operating systems [Dun+06], at least the register contents and a run-time stack are associated with a thread, but more OS resources, like an address space, may be referenced. On the thread dispatch, the execution context of the currently running thread is saved to memory, and replaced by the context of the dispatched thread.

The CPU multiplexing by the OS gives every thread the illusion of being alone on its own *virtual CPU*. For the thread, it seems that instructions from its program memory are continuously executed, since it cannot detect being stalled when it is currently *suspended*.

interrupts

Besides threads, there is another kind of activity the RTOS manages: *hardware interrupts*. With an *interrupt request (IRQ)*, the periphery signals the processor the occurrence of an external event. The interrupt controller detects this external signal and informs the CPU about the IRQ. If the current processor state allows for it, the CPU saves parts of its execution state and executes an *interrupt-service routine (ISR)* that is associated with the IRQ source. Normally, these ISRs are installed by the RTOS and it is a central RTOS chore to synchronize between service requests from the hardware and the software.

From the RTOS point of view, both concepts, threads and interrupts, are very similar and researchers have shown that they can be mapped onto each other [KE95; Hof+09]. Therefore, I will use the term *activity* as the generalization for these RTOS-managed activities.

task vs. thread

When we want to map tasks to threads, we face the problem that these concepts are of a different category. While a task is a static description of work, a thread is a currently executing activity and, hence, more like a job. Therefore, either the RTOS must provide some kind of persistent threads or we have to mimic this with other OS services. Figure 2.2 shows both cases. If an RTOS has native task support, you can release a job from a static task description as a thread activation, which terminates itself on completion. If there is at least one thread activation outstanding, the RTOS thread is *runnable*, otherwise it is *suspended*. One example of an RTOS with direct task support is the OSEK OS standard [OSE05].

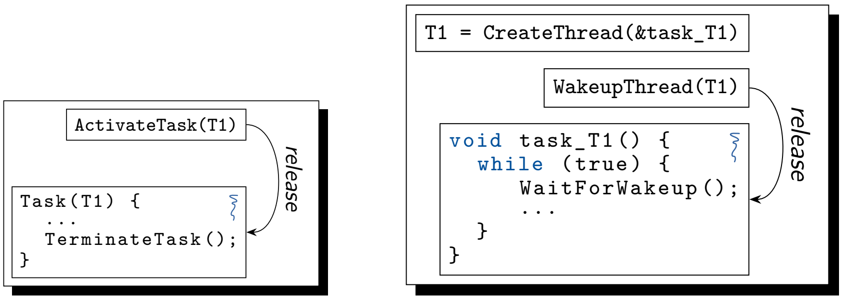
priorities

For an event-triggered system, we use an online scheduling strategy that decides upon the system state and different real-time parameters what thread should be dispatched next. For this, the priority-driven online strategies and their theoretical analysis was an important milestone for the real-time domain [Sha+04]. Thereby, the real-time analyst derives priorities as secondary parameters from the primary real-time parameters (e.g., period, deadline, or WCET). Attached to tasks, jobs, or parts of jobs, the analyst guarantees that the real-time system will meet all deadlines if the scheduler adheres to the priorities. The implementation engineer then ensures that among all active jobs, however they are mapped to RTOS abstractions, the highest-priority job is executed. This job runs until it terminates or an even higher-priority jobs is released and *preempts* the, now lower-priority, job.

We already see that the real-time analyst, as well as the implementation engineer have their parts in priority-driven scheduling. The analyst decides which entities (tasks or jobs) have attached priorities, where rescheduling should take place, and what are the priorities. The implementation engineer maps the tasks and jobs to control flows and implements an online scheduler that enforces the policy for this mapping. If the RTOS fails to enforce the mapping and a low-priority job is executed although a high-priority job is pending, we speak of a *priority inversion*.

The decision where to attach the priorities, distinguishes different classes of real-time scheduling: For task-level, fixed-priorities systems, we attach the priority to the task and all released jobs inherit this priority. Therefore, we also call them *static-priority* systems. The most prominent example how to derive static priorities from the real-time parameters is *rate-monotonic scheduling* [LL73], which assigns the highest priority to the task with the shortest period. However, also other priority-assignment policies like *deadline-monotonic scheduling* are available [LW82].

The other important class are the job-level, fixed-priority systems where a priority is derived for every released job. Due to this dynamicity, we call such systems also *dynamic-priority* systems. Here, the most important representative is the *earliest deadline first (EDF)* policy [LL73]. For EDF, the absolute deadline becomes a job's priority; the closer the deadline is to the current time, the higher is a job's priority.


```
ActivateTask(T1)
```
```
Task(T1) {
  ...
  TerminateTask();
}
```
```
T1 = CreateThread(&task_T1)
```
```
WakeupThread(T1)
```
```
void task_T1() {
  while (true) {
     WaitForWakeup();
     ...
  }
}
```

(a) with task support

(b) without task support

Figure 2.2 – RTOS with and without Native Task Support

Although, dynamic-priority systems can achieve optimal utilization [LL73] for some classes of task sets on uniprocessors, it is argued that fixed-priority systems are the most common choice in industry [Bra11, cha. 1.2][OSE05; Sak98]. One reason, as Brandenburg [Bra11] suggests, is the simplicity of their implementation. While we can implement the ready list for a fixed-priority system with a bit mask, we have to spent more development effort to get a fast implementation for EDF scheduling [Sho10]. In this thesis, I will concentrate on static-priority systems.

2.1.2.2 Control: Job Release and Directed Dependencies

With thread and priorities, our RTOS already provides abstractions to execute jobs coordinated. However, we also have to release jobs and express directed dependencies between them with the help of RTOS services. If our RTOS already provides a task abstraction, like discussed in the previous section, we release a job by starting the corresponding thread when a periodic or sporadic event occurs.

For sporadic tasks, the activating event is external to the RTCS and we do not know the exact release time. In actual implemented RTCS, peripheral components (e.g., digital switch or an acceleration sensor) send these events to the processor and the RTOS receives and interprets them, normally with the help of specialized interrupt-detection hardware. Upon a job release, it depends upon the RT model, whether the RTOS must reschedule immediately or if the reschedule happens at some later point in time (e.g., at the next system call).

This direct temporal connection between the external event and the job release is the defining factor for an event-triggered system. In a purely time-triggered system, we have to include table entries to poll for the activation of sporadic tasks. Furthermore, we must reserve time budgets in the table to execute sporadic tasks by using, for example, a sporadic server [SSL89].

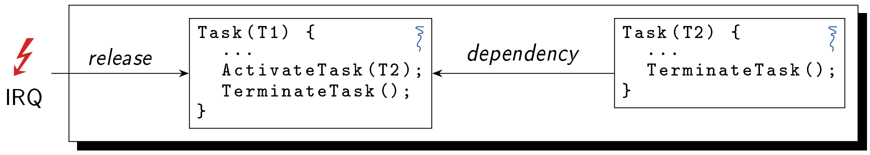
For periodic tasks, an event-triggered RTOS provides the possibility to activate threads periodically. Alarm objects, like the OSEK standard [OSE05] specifies them, provide such a timing service. Configured with the period and the offset of a periodic task, the RTOS releases jobs for a specific task in equidistant time intervals. Often such timing services, not only allow periodic thread activations, but they also offer one-shot timers and abortable alarms.

Unlike a time-triggered, table-driven RTOS, the periodic activation only releases periodic jobs and does not directly lead to a preemption or resumption decisions. Take, for example, the rows for $T = 0$ and $T = 6$ in Table 2.2: both rows result in the dispatch of the τ_1 but only the first one is responsible for the job release.

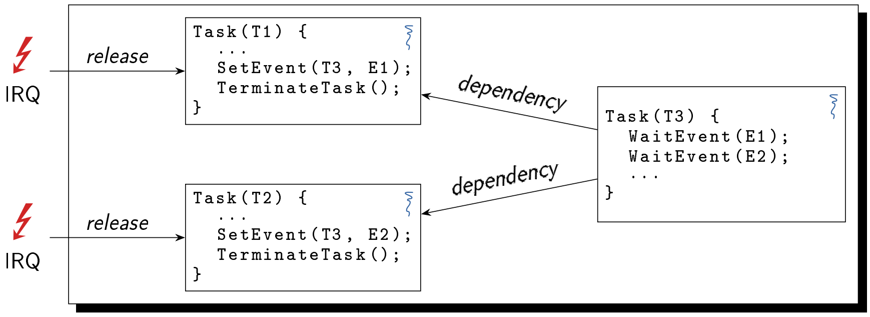
Besides the periodic and sporadic release of a job, an RTOS should also provide mechanisms to express directed dependencies. If a task has only one predecessor that must have finished its execution beforehand, we can just activate the dependent thread after the dependency is fulfilled (see Figure 2.3a). For this, the RTOS must provide a system call for activating a thread from a thread context, additionally to the activation from an ISR context. As these synchronous activations are similar to external events, [Sch11] also refers to them as *logical events* in comparison to *physical events*.

However, with synchronous activations, we are still not able to express multiple directed dependencies. While an RTOS could provide annotations for such task dependencies, most RTOS do not provide such a descriptive interface. Instead, we can use mechanisms for passive waiting and inter-thread signaling to mimic multiple dependencies on the implementation level. When an RTOS provides such a mechanism, a thread can voluntarily wait on a waiting object. Thereby, the RTOS transfers the thread into the *waiting* state until another thread (or ISR) signals the waiting object. Until then, the thread is not in the *ready* state and the RTOS excludes it from scheduling decisions.

waiting



(a) Single Dependency



(b) Multiple Dependencies

Figure 2.3 – Mapping of Task Dependencies with RTOS Services

The classical example for such a waiting object is the semaphore [Dij65], which also allows for all other kinds of inter-thread synchronization.

For the RT mapping, we have to use distinct patterns of waiting system calls to express multiple dependencies. For example, for an AND dependency on two tasks, the dependent thread sleeps on the occurrence of two event signals in sequence (see Figure 2.3b). As long as the events are not signaled, the thread is waiting and keeps preempted. After each predecessor thread has signaled one event object, the waiting thread proceeds after both signals occurred. Thereby, the signal must be buffered by the RTOS to make the reception of signals order invariant and to avoid the lost-wakeup problem [Lam67].

2.1.2.3 Synchronization: Mutual Exclusion

As the last aspect of our abstract real-time model, we must implement undirected dependencies. An undirected dependency between two tasks instructs the RTOS to prevent the interleaved execution of jobs from these tasks. So even if a job from a higher-priority job gets released, it is not scheduled before the lower-priority job finishes if they share an undirected dependency. While undirected dependencies mostly stem from application requirements, like access to a shared indivisible resource, they may also arise from real-time concepts like *preemption thresholds* [WS99], which increase the schedulability of task sets. There, each task gets a preemption priority that it uses to preempt other tasks and a preemption threshold that it uses to prevent preemption by other tasks. Thereby, all tasks with the same preemption threshold cannot be active at the same time.

preemption thresholds

Since the mutual exclusion of program sections is a common *synchronization* problem, most operating systems provide blocking locks for the application to synchronize their threads. At the beginning of the to-be-protected program section, an activity takes the lock and releases it only at the section end. As long as another activity holds this lock, the RTOS transfers other lock-requesting threads to the waiting state; hence the name blocking lock. When the lock holder gives the lock back, the RTOS wakes up one of the waiting threads and hands over the lock ownership.

It is one distinguishing feature of such mutual-exclusion mechanisms, if other threads are able to make requests for an already acquired lock, or if the scheduling avoids this situation altogether. Examples for the former class are semaphores [Dij65] and priority-inheritance locks [SRL90b]. However, as these mechanisms are prone to *deadlocks*, where two thread hold one lock each and wait circular for the lock of the other thread, the systems community developed mechanism in the latter class, like non-preemptive critical sections or stack-based, priority-ceiling locking [Bak91].

While time-triggered systems can avoid locking by constructing the scheduling table such that all undirected dependencies are already full filled, event-triggered systems require such explicit synchronization mechanisms in the application code. However, they also give the implementation engineer a much more fine-grained control (down to the statement level) about mutual-exclusive sections. For the static analysis, they are explicit annotations in the application code and can benefit the blocking-time analysis of a system [BA10].

2.1.3 The OSEK Operating-System Standard

While the discussed abstractions (i.e., threads, ISRs, activation, signaling, and mutual exclusion) are present in most RTOSs, many provide additional features and have, more or less slightly, deviations in their semantics. To make things concrete but not too pinned down by a single implementation, I chose the OSEK operating-system standard [OSE05] as a well-defined interface for the analyzed real-time applications. Throughout the thesis, I will use it for example systems and it also defines my investigated operating-system model.

Starting in 1993 [Joh98], the German automotive industry started an initiative to harmonize the software architecture used in automotive *electronic control units (ECUs)* to increase the portability, extendability, and reusability of software components. The OSEK[3] standard covers communication between ECUs [OSE04b], the network management [OSE04c], and the operating-system interface [OSE05].

The OSEK-OS standard, which I will just refer to as OSEK standard from now on, is specified as a single-core operating system. However, the AUTOSAR [AUT13] specification, which is the successor to OSEK, extends the OSEK world with multiprocessor support. Nevertheless, the AUTOSAR standard describes a partitioned multi-processor model, where an OSEK-OS instance is spawned on each core and threads cannot migrate between cores. Therefore, I focus on the single-core OSEK standard, which still reflects the state-of-the-art in automotive ECUs and automotive safety-critical real-time systems.

While OSEK was developed by practitioners, it directly supports a wide range of concepts from the real-time domain and is therefore a good base for my work on the interaction analysis of real-time systems. Therefore, I will, in the following, describe OSEK, its design philosophy, and its system services in detail as far as they are important for this thesis.

static operating system

The most drastic paradigm change for developers that come from the world of desktop computing is that OSEK is a *static operating system*. This means that all RTOS objects, like threads, locks, and wait objects, must be declared before run time. They are known by their declared name and no additional objects can be allocated at run time. While this constraint looks like a harsh restriction on the flexibility, the common real-time task models and most scheduling tests also prescribe a fixed number of tasks. Furthermore, this large amount of ahead-of-time knowledge allows for a memory-efficient implementation of the RTOS, as all objects can be stored in statically allocated arrays. Hence, no dynamic data structures, like linked lists, or memory allocators are required to

---

[3] OSEK stands for "**O**ffene **S** und deren Schnittstellen für die **E**lektronik in **K**raftfahrzeug" (engl., "Open systems and the corresponding interfaces for automotive electronics").

manage the OS objects. As OSEK targets small embedded systems, this focus on memory efficiency is considered as one of the major success factors of OSEK [DMT00].

Furthermore, an OSEK-compatible RTOS does not only know all system objects but also their real-time properties, like priorities or preemptability. The developer has to declare the system objects and their properties in a domain specific language called *OSEK implementation language (OIL)* [OSE04a], which is interpreted by a system generator. A short example of an OIL file is shown in Listing 2.1: task T1 has a static priority of 2 and is fully preemptable.

Listing 2.1 Example for an OIL file

```
TASK T1 {
  PRIORITY = 2;
  SCHEDULE = FULL;
}
```

activities in OSEK

OSEK directly supports tasks instead of providing only a thread abstraction. Behind the declaration of task T1 in Listing 2.1, an entry function with the same name is present within the application source code. For each released job, the entry function starts from the beginning in a newly created thread until it terminates, which the RTOS schedules under a fixed-priority policy. Rescheduling only takes place at well-defined rescheduling points (i.e., system-call invocations, alarm expiration). After the job finishes its execution, the thread is destroyed and no data is left on the execution stack. While OSEK tasks are threads with a statically-defined way to create them, they are meant to be continuously created and destroyed on every job execution.

OSEK specifies two kinds of tasks with different abilities: *basic tasks* and *extended tasks*. Basic tasks always execute in a run-to-completion manner and are not allowed to wait passively. This means that they execute instructions until they terminate themselves, but they are not allowed to invoke a blocking system call such that they never enter the waiting state. Thereby, basic tasks can all be started on a single shared stack; an aspect of OSEK that we will revisit in Chapter 6. Extended tasks, on the other hand, do not have this non-blocking restriction and are allowed to invoke all system services.

Basic tasks and extended tasks have several real-time properties: Besides the static priority, which the job inherits on creation, tasks can be marked as non-preemptable. While a preemptable task can be preempted in favor of a higher-priority task at every rescheduling point, a non-preemptable task is not removed from the processor until it terminates or enters the waiting state. Thereby, OSEK allows for fully-preemptable, mixed-preemptable, and fully–non-preemptable systems. If a task is marked as `AUTOSTART`, the RTOS releases one job at boot time. Furthermore, the developer must also declare the maximal number of pending jobs that can released per task.

Besides tasks, OSEK also supports the definition of application-specific ISRs, which again come in two different abilities: A category-1 ISR is not allowed to invoke any system call, has no influence on the scheduling, and comes with the smallest amount of RTOS overheads. The category-2 ISRs (*ISR2*) can invoke a limited set of system calls (i.e., task activation, event signaling) in order to influence the scheduling.

control in OSEK

For the synchronous management of jobs, OSEK provides three system calls: `ActivateTask(⟨TASK⟩)` activates a thread, `TerminateTask()` immediately finishes the execution of the currently running job, and `ChainTask(⟨TASK⟩)` combines the self-termination and the activation of another thread atomically. Thereby, the thread-activating system calls are the mechanism to express simple directed dependencies. Table 2.3 gives an overview about these system calls, as well as about the other important system calls.

`System Call`	Arguments	Description
`ActivateTask`	TASK	Releases one job of the specified task.
`TerminateTask`	–	Self Termination of a job.
`ChainTask`	TASK	Atomic combination of ActivateTask and TerminateTask
`SetRelAlarm`	ALARM, offset, period	Configures an alarm relative to the current time which triggers periodically after the initial offset.
`SetAbsAlarm`	ALARM, offset, period	Like SetRelAlarm, but with an absolute starting time.
`CancelAlarm`	ALARMTYPE	Disable an armed alarm.
`WaitEvent`	EVENTMASK	Transfer currently running job into the waiting state until another job or an ISR signals *one* of the events.
`ClearEvent`	EVENTMASK	Reset signaled events from the jobs event mask.
`SetEvent`	TASK, EVENTMASK	Signal all specified events to the given task.
`SuspendAllInterrupts`	–	Blocks ISR1 and ISR2 interrupts.
`ResumeAllInterrupts`	–	Unlocks ISR1 and ISR2 interrupts.
`GetResource`	RESOURCE	Acquires a resource and boost the jobs priority to the ceiling priority
`ReleaseResource`	RESOURCE	Gives a resource back to the RTOS and decreases the dynamic job priority.

Table 2.3 – OSEK System Calls. Selection of important OSEK system calls. Arguments in SMALL CAPS are references to system objects that are declared in the OIL file.

As timing service for periodic tasks, OSEK provides *counters* and *alarms*. Each counter is an integer-typed variable that overflows to zero if it reaches a predefined upper bound. A hardware-timer ISR increments all counters in equidistant time intervals and performs all necessary downstream operations. Statically connected to one counter, an alarm object has a period, an offset from $t = 0$, and two Boolean flags that indicate whether the alarm is periodic and is armed. While we can manipulate these parameters dynamically via system calls, each alarm is statically connected to invoke one of two actions on expiration: activate a thread or signal an OSEK event.

For extended tasks, OSEK provides the possibility to wait passively for *events*, which are similar to POSIX signals. Every thread has a *private* set of signaled events, which he can (partially) clear with `ClearEvent()` and others can extend with `SetEvent()`. On job-execution start, the task's event set is implicitly cleared [OSE05, p. 27], which is an inherent difficulty for avoiding the lost-update problem. If a thread waits (`WaitEvent()`) for an event that is not yet signaled, it enters the waiting state until another activity signals the requested event.

Furthermore, `WaitEvent()` takes not only a single event but a set of requested events and wakes up/continuous if at least one requested event is signaled; `WaitEvent()` checks an OR-condition on the task's event set. For an AND-condition, the implementation developer has to invoke `WaitEvent()` multiple times with different event sets (see Figure 2.3b). Thereby, the events are the mechanism to express arbitrary directed dependencies.

The OSEK standard mandates that the developer has to declare the list of events that a thread can wait on. Due to the private nature of a task's event set, where the same event (name) can be signaled to different threads, we must address each send and received event via the tuple (thread, event).

mutual exclusion in OSEK

For the expression of mutual exclusion, OSEK provides two different mechanism: ISR blockades and OSEK resources. While the former exposes the CPU-level synchronization mechanism to the application, the latter works purely by influencing the scheduler.

For ISRs, OSEK provides system services to block and to unblock the execution of ISRs. Thereby, the user can choose to delay only ISRs of category 2 (e.g., `SuspendOSInterrupts()`) or all interrupts that the hardware supports disabling for (e.g., `SuspendAllInterrupts()`), including hardware timers.

With *resources*, OSEK provides a second mechanism to express the mutual exclusion of program sections. OSEK resources are thread-level locks that use the *stack-based resource protocol (SRP)* [Bak91] to avoid priority inversion and deadlocks by construction. The SRP is an extension to the *priority ceiling protocol (PCP)* [SRL90a] that eases the implementation and minimizes the number of context switches.

For the SRP, each lock must know all possible lock-holding tasks to calculate the lock's *ceiling priority* as the maximum of the task's static priorities. Whenever a thread acquires the lock (`GetResource()`), we boost the thread's *dynamic priority* to the ceiling priority. The OSEK scheduler schedules according to the dynamic priorities, which the RTOS initializes for every job with the static priority of the released thread.

After the acquisition, the currently running thread has the highest priority of its resource group and no other thread that could acquire the lock will be scheduled. Therefore, the acquisition of a resource will always succeed immediately and a thread will never wait for a resource to become available. Thereby, deadlocks cannot occur as threads cannot enter the waiting state due to an acquisition request.

OSEK resources come in two different flavors: A normal resource must be acquired explicitly by a system call, and we use it to annotate program sections. Besides that, OSEK also provides internal resources which are implicitly taken by a thread if we dispatch it. Therefore, implicit resources are equivalent to preemption thresholds [WS99], which was also shown by Gai, Lipari, and Di Natale [GLD01].

Furthermore, OSEK always provides a `RES_SCHEDULER` resource that every thread can acquire. Therefore, its ceiling priority is the highest priority in the system, and acquiring it effectively becomes a non-preemptive critical section.

In order to reduce the resource consumption for smaller ECUs, OSEK defines four conformance classes that imply an increasing amount of RTOS complexity. In the basic conformance class 1 (BCC1), the RTOS only supports basic tasks and, hence, has no support for events. Furthermore, multiple activations and more than one task per priority level are only mandated in the basic conformance class 2 (BCC2). With the extended conformance class 1 (ECC1), the RTOS supports extended tasks and event objects. However, support for multiple basic-task activations or multiple tasks per priority level come only in the extended conformance class 2 (ECC2). For this thesis, I will concentrate on BCC1/ECC1 systems.

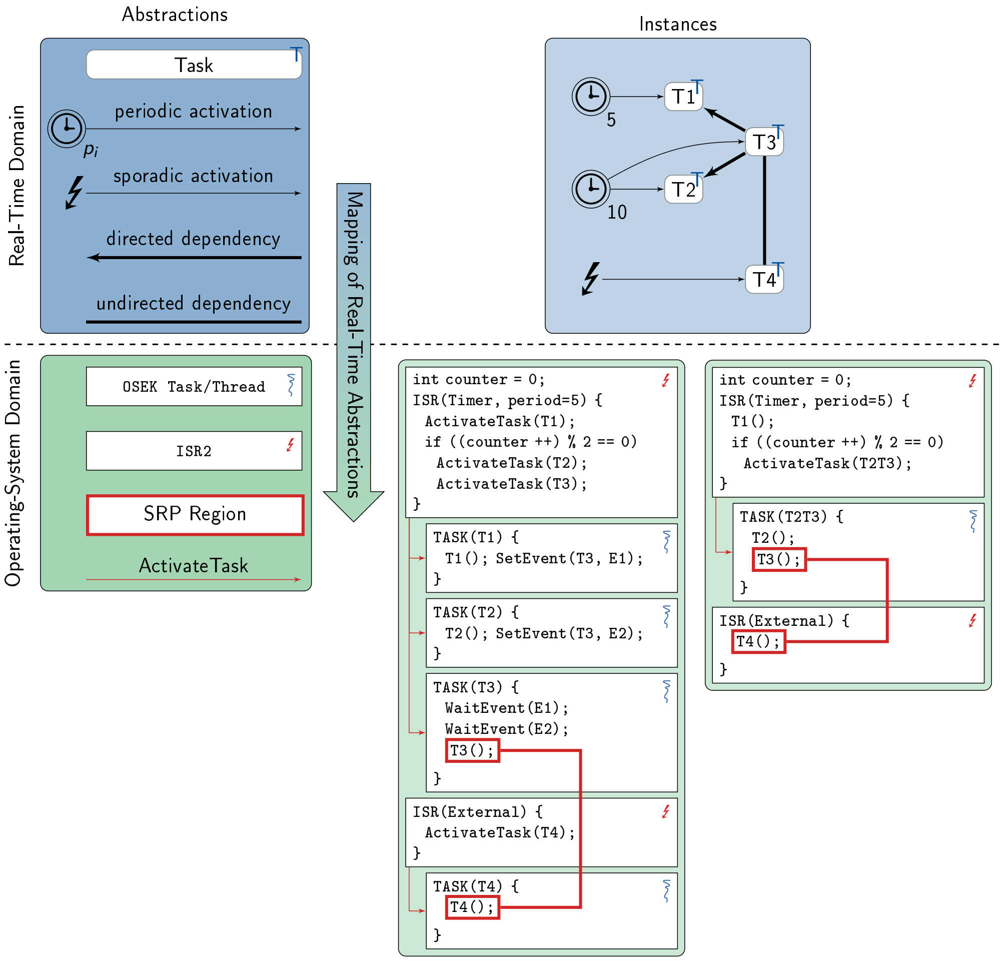
2.2 From the Real-Time to the Implementation Domain

In Section 2.1.1, we discussed the real-time analyst's mental model of RTSs. And although event-triggered operating systems (Section 2.1.2) are close to this model, problems arise from the transition between both domains. These difficulties are a call to action to take a closer look not only on the real-time application in the RT domain, but also on the implemented application in the OS domain.

2.2.1 Ambiguous Mapping of Real-Time Concepts

Given an RTCS in the real-time domain, different mappings for tasks, activations and dependencies onto RTOS and hardware mechanisms are possible. Figure 2.4 illustrates the general process of mapping from the real-time domain to the implementation-level domain and shows an example system with two possible mappings. In the real-time domain, we have three periodic tasks (T1-T3) and one sporadic task (T4). Furthermore, the execution of T3 depends on the completion of T1 and T2; T3 and T4 are mutual exclusive.

For the OS domain, we will for now ignore the priorities and possible priority inversions that can occur from an incorrect mapping. Figure 2.4 contains two possible mappings of the example system onto an OSEK API. The work of the real-time tasks is located in functions of the same name (e.g., task T1 becomes the function `T1()`). On the left, we mapped each task into its own thread/OSEK task



```
int counter = 0;
ISR(Timer, period=5) {
  ActivateTask(T1);
  if ((counter ++) % 2 == 0)
   ActivateTask(T2);
   ActivateTask(T3);
}
  TASK(T1) {
   T1(); SetEvent(T3, E1);
  }
  TASK(T2) {
   T2(); SetEvent(T3, E2);
  }
  TASK(T3) {
   WaitEvent(E1);
   WaitEvent(E2);
   T3();
  }
ISR(External) {
  ActivateTask(T4);
}
  TASK(T4) {
   T4();
  }
```

```
int counter = 0;
ISR(Timer, period=5) {
  T1();
  if ((counter ++) % 2 == 0)
   ActivateTask(T2T3);
}
  TASK(T2T3) {
   T2();
   T3();
  }
ISR(External) {
  T4();
}
```

Figure 2.4 – Diffrerent Mapping of Real-Time Concepts onto RTOS Abstractions

and implemented the directed dependencies with two events that are signaled by T1 and T2. T3 waits in its task-entry function `T3()` until both events are signaled. The activation of the periodic task is done with a timer-driven ISR with a period of 5. The sporadic task (T4) is activated by its own ISR that is connected to the external event.

While the left mapping reflects the exact structure of the task model, it is not the most resource efficient implementation. On the right side, we implemented the same RTCS with a smaller number of RTOS objects. For example, we inlined T1 and T4 directly into their respective ISRs and combined `T2()` and `T3()` into the T2T3 thread. Thereby, we fulfill the directed dependency between T2 and T3 by the sequential execution in T2T3. For the T1–T3 dependency, the location of T1 in the Timer ISR and the subsequent `ActivateTask()` ensure the correct ordering.

While such a compressed mapping can be beneficial for the resource consumption, it increases the verification complexity that the model was correctly implemented. However, such compression during the mapping process is common practice in industry. For example, the AUTOSAR [AUT13] calls its (real-time) tasks “runnables” and a system generator maps multiple runnables into one RTOS thread [YB10]. Thereby, ECU vendors are able to shrink the memory usage of the systems, as only one RTOS thread data structure is required and the subsequently executed tasks will surely reuse the same stack space.

By this compression, we lose the strict one-on-one mapping between RT tasks and OS threads that the standard literature suggests [Liu00, cha. 1.1]. This combination of tasks also has influence of the scheduling and, therefore, has to be done carefully. For example, if T4 has the lowest priority in the system, a priority inversion arises from the execution of T4 in the ISR (right side). However, if the execution time of T4 is shorter than the RTOS overhead for activation, dispatch, and termination, the deliberate inversion has less impact on the real-time schedule than the canonical form.

Another difference between both mappings is the timing characteristic of the multiple-dependency pattern of T3. In the direct mapping, the RTOS schedules T3 up to three times before the T3 thread executes one job. When the ISR activates the thread, it starts and waits for the first event E1. When E1 arrives, T3 wakes up for a short period only to wait for the second event E2. If E2 also arrives, the thread is scheduled a third time, executes `T3()`, the actual task, clears E1 and E2, and goes back to sleep. In the compact mapping, the directed dependencies are implicitly encoded in the order of job-function invocations.

It is noteworthy that the distinction between real-time task and operating-system thread is a common source of confusion between both domains. While the task is work that the RTCS must somehow execute, a thread is an OS object, normally a co-routine, which the OS scheduler selects at run time. A task is a static template for many transient, short-lived jobs, while a thread often processes many work packages or messages, as it can also behave as a server for other threads. In order to minimize the confusion, I will keep terms from both domains separate and I will avoid the usage of “task” as some general duty or work, but I will use the word “chore” instead. If I want to address the OS activity that is managed by the scheduler and is a technical entity, I will use “thread”.

This disassociation from OSEK tasks with the idea of real-time tasks, also becomes visible when we look at a common usage pattern of the OSEK API. The looped-waiting pattern in Figure 2.5a is another possibility to implement arbitrary patterns of multiple directed dependencies[4]. Here, we implement the same dependencies for T3 as in Figure 2.4, but the thread T3 executes multiple T3 jobs. We mark T3 as `AUTOSTART=TRUE` in the OIL file and it processes T3 jobs in an endless loop. In this example, which is a correct usage of the OSEK API, the thread `T3` lost its notion of a real-time task. It “degenerated” to an (eternally living) OS thread that only carries a task in its while–true loop.

---

[4] Scheler [Sch11] describes even more patterns to implement multiple directed dependencies with the OSEK API.

```
TASK(T3) {
  while(1) {
    WaitEvent(E1);
    WaitEvent(E2);
    // T3 job implementation
    T3();
    ClearEvent(E1|E2);
  }
}
```

(a) OSEK Task to Thread Degeneration. The actual work of the T1 task is embedded into a loop executes one step for every E1 or E2 event. However, the EventLoop OSEK task is no longer a task by itself, but it becomes an eternally living thread that carries a task.

```
unsigned int c = 0;
unsigned int freq; // 0..20

TASK(Coordinator) { // peridically activated
  c++;
  if ((c % 2) == 0) ActivateTask(T1);
  if ((c % 3) == 0) ActivateTask(T2);
  if ((c % (5 * freq)) == 0)
    ActivateTask(T3);
  TerminateTask(); }
```

(b) Emulation of Time-Triggered RTOS in OSEK. The developer of this OSEK task misused the alarm capabilities of OSEK to emulate an time-triggered with different operation modes via a helper thread.

Figure 2.5 – Non-Canonical Usages of the RTOS API

We have seen that the implementation engineer has multiple possibilities to map the same real-time application onto the RTOS APIs. These different mappings have different characteristics when it comes to resource consumption, but they also exhibit different timing characteristics. However, it is not the real-time application in the RT domain that must exhibit a correct timing, but the actual implementation on a concrete hardware. Therefore, it is essential to take a close look at application behavior on the implementation level, which is closely tied to the RTOS–application interaction for an event-triggered real-time system.

2.2.2 RTOS API Flexibility

Another aspect of the mapping process is the flexibility of the RTOS API and to what degree it fosters and enforces explicitness. While the RT domain has a limited and well-defined set of task dependencies and relationships, an OS interface is normally designed to support all kinds of applications. For this, OS APIs often support a small set of basic operations that can be composed into more complex operations. For example, in Linux, the `fork(2)` system call and the dynamic creation of threads is not supported separately, but the C library maps them both to the `clone(2)` system call.

From the RT analysts point of view, it would be desirable that the RT-domain operations are reflected one-to-one in the set of available OS operations. Thereby, the analyst could ensure that her model covers the actual RTCS entirely and that the implementation does not manipulate the scheduling in an unforeseen way. However, such a restriction is unrealistic as the RT model of the system is exactly that: a model and, thereby, a problem specific simplification of the required solution. When the model of the RTA comes into contact with the system environment, the hardware platform, and the behavior of peripheral devices, the implementation engineer often requires a more flexible and composable RTOS interface. We leave out these implementation necessities in the RT model on purpose, but they will still end up in the deployed system.

For example, if we assume a task model without undirected dependencies (no mutual exclusion), the implementation will often still require a mechanism to synchronize access to shared resources in order to ensure a correct operation. These mutual-exclusive sections might not even stem from the application code itself but from communication with the hardware or the usage of third-party libraries (i.e., logging). Therefore, the restriction to the exact set of RT-domain operations is unrealistic and the RTOS API must remain flexible and composable. From this dichotomy between RT and OS domain, different mapping issues can arise.

First of all, a flexible RTOS API that does only support basic composable operations leads to different mappings for the same RT-domain operation. We have already seen this situation for the implementation of multiple directed dependencies. As OSEK has no canonical support for the arbitrary combination of wait conditions, different implementation patterns (Figure 2.5a, Figure 2.4, [Sch11]) with different preemption characteristics can occur in the real-world code. However, even if there would be the compound operation `WaitMultiple(⟨bool. exp.⟩)`, legacy code could still contain one of the composed patterns and developers could still use them.

The second issue, which is similar but distinct from the first one, is the "creative" use of the RTOS API. As the RTOS interface has to stay flexible, the developer can use it to implement patterns that are not covered by the RT task model. And while such constructs should not occur, they will, to a certain degree, develop in real-world application code bases given enough time.

For example, in Figure 2.5b, we see such a creative usage of the RTOS API. Here, the RTOS periodically activates the `Coordinator` thread, which increments a counter variable `c` and activates other threads in regular intervals. Furthermore, the activation interval of thread T3 is not even constant but can be adjusted at run time by modifying the variable `freq`. Even worse, the variable `c` is globally visible and could be set by other parts of the system to every possible value, as the range of possible values is only noted in a comment. This usage pattern suggests that the developer had more or less a time-triggered system in mind but his manager instructed him to use an event-triggered system.

We see that the RT model and the implementation will diverge to a certain degree, since the implementation must not only adhere to the real-time requirements but must also solve other engineering necessities. Therefore, the implementation level becomes the ground truth for the correctness and timeliness and it is essential to take a close look at the implemented interaction between RTOS and application.

2.2.3 Impact of Implementation Side-Effects

The third aspect of the mapping from RT to OS domain that must be considered for RTCSs, are the side effects of the implementation on the real hardware with an existing operating system. Even if the OS implements the desired functionality correctly, the side effects of multiplexing multiple jobs onto the processor, which also executes the OS itself, impact the properties of the RTS.

Hardware issues like *cache-induced preemption delays* [LMW96; Mue00] and *pipeline-induced delays* [HWH95; ZBN93; CP00] are an issue that influences the WCET. However, they are normally [CP01] considered on the task level where one job is analyzed in isolation and the effect of the other system components is added pessimistically in the *response-time analysis* [JP86]. However, as we have seen, our RTOS does not manage tasks and jobs, but threads that execute mapped tasks and jobs. Therefore, only an analysis of the implementation yields a final certainty of the system properties.

rate-monotonic priority inversion

An example where this influence of the implementation was neglected for a long time is the *rate-monotonic priority inversion* [LMN06]. This problem arises if the RTOS uses two priority spaces for ISRs and for threads, and if interrupts are serviced with a higher priority. In this scenario, a thread with a high priority, which stems from a short periodicity in rate-monotonic scheduling, is interrupted by an interrupt with a long minimal interarrival time. The ISR should actually have a lower priority than the thread according to rate-monotonic scheduling, but the implementation violates the structure of priorities. There were several methods proposed to solve this problem by changing the implementation and unifying the priority space [LMN06; HLS11].

Also in this third dimension, we see that it is not sufficient to look only at the RT domain to make statements about a RTCS, but we must inspect the deployed implementation for side effects.

2.2.4 Analysis on the Implementation Level

In the last three sections (Section 2.2.1–2.2.3), we have seen different problems that arise from the transition between RT domain and OS domain:

- Ambiguous mapping of concepts and operations from RT domain onto RTOS services.
- Implementation necessities that requires a flexible RTOS API, which can be misused.
- Side effects of the system software and hardware must be considered in the real-time domain.

From these observations, I draw multiple conclusions: First, I conclude that the implementation of the RTCS as a whole is the only source of truth when it comes to making statements about system properties. Second, we must analyze system properties on a more or less precise model of the deployed RTCS; the closer our model is to the implementation the more precise are our statements about the actual behavior. Third, the presence of deviations from the real-time model in the implementation stems not only from bugs, but they are often born out of necessities and have to be taken seriously.

In addition, the model-driven development approach that I outlined in this chapter and that includes a distinct mapping step from the RT to the OS domain does necessarily reflect the reality of software development. Sometimes, the RT model is only used in the initial development phase and all maintenance fixes are done on the implementation level such that the RTA continuously diverges from the model. Some projects manually develop RT and OS model in parallel and try to keep them in sync. And other real-time systems are developed in a bottom-up fashion purely on the implementation level. However, for all these scenarios, the source code of the application is always available, is always up-to-date, and is the fix point that covers all development models.

These observations led me to the approach of implementation-level *interaction analysis* of the RTCS. I start out with the source code of the real-time application that is sprinkled with system calls. These system calls are the “markup” language that the developer used to indicate the desired interaction between RTOS and application. The investigated RTAs are result of the RT–OS mapping process and already have assigned real-time properties, like thread priorities and preemption thresholds. These attributes, as well as the application structure, are ought to be *fixed*, and we are not allowed to adapt them in later optimization steps. My analyses combine RTA and real-time parameters with the semantic of the RTOS to provide more realistic (and stricter) statements about functional and non-functional properties. The presented approach is not a replacement for the analysis on the RT domain, but acts as a *post-mapping* validation, analysis, and optimization in the OS domain.

2.3 Chapter Summary

In this chapter, I gave an introduction to the concepts of the real-time domain that are used to model a real-time application and presented the mechanisms and the abstractions that the RTOS provides at its interface. While the real-time domain narrows the focus on the analysis of the timing behavior, the RTOS and the actual implementation of the application additionally face hardware limitations, engineering constraints, and reusability aspects. Therefore, both worlds, the real-time domain and the operating-system domain, diverge in their view of the system and we have to ensure that guarantees from the upper levels are not ridiculed in the implementation.

Therefore, I argue that an implementation-centric approach to whole-system analysis is a good fit to answer my research questions as it acknowledges the reality of threads, the RTOS, and the interaction between both that the application code contains. With this thesis, I want to close the semantic gap between real-time and operating-system domain and develop analyses and optimizations techniques of the whole system that are performed on the actual implementation.

3

Foundation

Fine-Grained Interaction Knowledge

> "The physical world is real." That is supposed to be the fundamental hypothesis.
>
> *Letter to Eduard Study*, 1918, ALBERT EINSTEIN

After we have covered the abstractions of real-time computing systems and their mapping onto OS primitives, we focus on the RTOS-mediated control transfer between threads. As we have seen, the scheduling of threads is crucially important for real-time systems to perform actions on time. On the real-time domain, the structure and the properties of the schedule are already under heavy investigation by the real-time community. However, on the post-mapping implementation level, we also have to grasp the system's structures and properties since they define the actual behavior of the deployed system.

The knowledge about the interaction between RTOS and application holds the potential to better understand the actual requirements of concrete applications. By understanding these requirements, we will be able to perform better whole-system analyses that provide tighter bounds and stricter correctness guarantees, as well as system-wide optimizations that improve non-functional system properties. In this chapter, I will present those data structures and methods that capture these requirements in terms of the interaction between application and RTOS. For this, we will combine the implementation-level application logic, the RTOS configuration, and the system-call and interruption semantics.

Related Publications

[▷DHL15a] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems.” In: *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES ’15 (Portland, Oregon, USA). New York, NY, USA: ACM Press, June 2015. ISBN: 978-1-4503-3257-6. DOI: `10.1145/2670529.2754963`.

[DHL15b] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Globale Kontrollflussanalyse von eingebetteten Echtzeitsystemen.” In: *Betriebssysteme und Echtzeit, Echtzeit 2015, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V. (GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG) sowie der Fachgruppe Betriebssysteme von GI und ITG, Boppard, 12. und 13. November 2015*. 2015, pp. 128–136. DOI: `10.1007/978-3-662-48611-5_14`.

[▷DHL17] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis.” In: *ACM Transactions on Embedded Computing Systems* 16.2 (2017), 35:1–35:25. DOI: `10.1145/2950053`.

[▷Die14] **Christian Dietrich**. “Global Optimization of Non-Functional Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses.” Master’s Thesis. Department of Computer Science 4, Distributed Systems and Operating Systems; University of Erlangen-Nuremberg, 2014.

3.1 Introduction

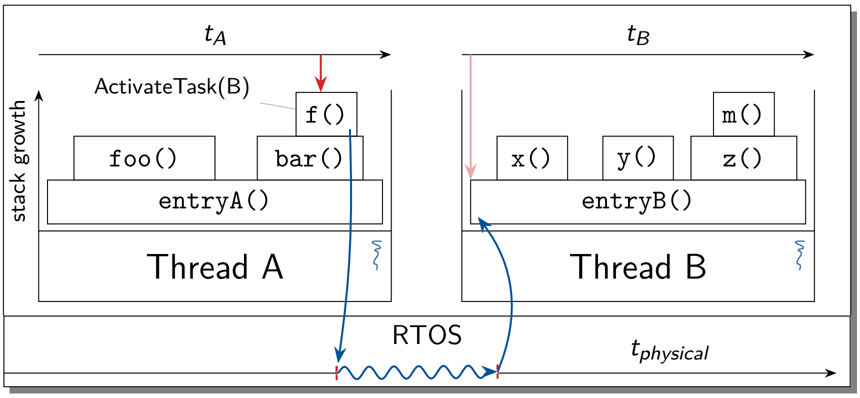


Figure 3.1 – Interaction between Threads and RTOS. For two threads, we see the call graph visualized as a flame chart. Each thread executes along its own time axis (t_A, t_B) but *interacts* with the RTOS via system calls. In the example, thread A mandates the RTOS to activate thread B, which schedules the latter one at some point on the physical-time axis.

interaction

First, we will take a closer look at the term "interaction" and its meaning in the interplay between RTOS and application. In general, the Merriam-Webster dictionary defines interaction as "mutual or reciprocal action or influence"[5]. In our setting, we identify a single thread and the RTOS as the subjects of the mutual influence, since the RTOS mediates the thread execution, but can be influenced by the thread. Thereby, it is an RTOS design decision to do this mediation directly via explicit system calls, or indirectly by giving controlled access to shared memory and peripheral devices. The "action or influence" between thread and RTOS is indeed a mutual one: While the thread hands work to the RTOS, the RTOS controls the execution of and the data flow towards the thread. A good example for this mutual influence is the blocking file-system operation `read()`: The thread mandates the input of data from the disk and the RTOS blocks the thread until the result arrives in the memory.

For real-time computing systems, the control over the processor and the interleaved execution of multiple threads are the most important aspects of the reciprocal influence as they are directly related to the system's timeliness. While the inter-thread scheduling for general-purpose systems is often driven by non-functional properties like throughput or reactiveness, the schedule of a RTCS is closely connected to the application' structure and its requirements.

In Figure 3.1, we see several important aspects of the reciprocal influence in a RTCS with two threads. For both threads, we see their inner structure in terms of their call graph as a flame chart [Gre16]. The stacking indicates that a function calls a child function (e.g., `bar()` calls `f()`) and the horizontal extend indicates the execution time. Each thread executes on its own (virtual) time axis (t_A, t_B), which does not progress if thread is currently preempted. It is noteworthy that the *worst-case execution time (WCET)* is measured on this virtual time axis, while the *worst-case response time (WCRT)* is measured on the physical time axis ($t_{physicial}$), which is managed by the RTOS. In the figure, we see a situation where thread A mandates the RTOS to activate thread B. As we have only one processor in the example, the RTOS schedules at some later point in time thread B for execution and *hands over* the control about the processor to thread B, whose time axis t_B starts

---

[5] https://www.merriam-webster.com/dictionary/interaction

progressing. This transfer of control between threads is the heart of my interaction models, where we will build a system-wide control-flow graph that contains edges between code blocks if they can execute directly in sequence on the physical time axis.

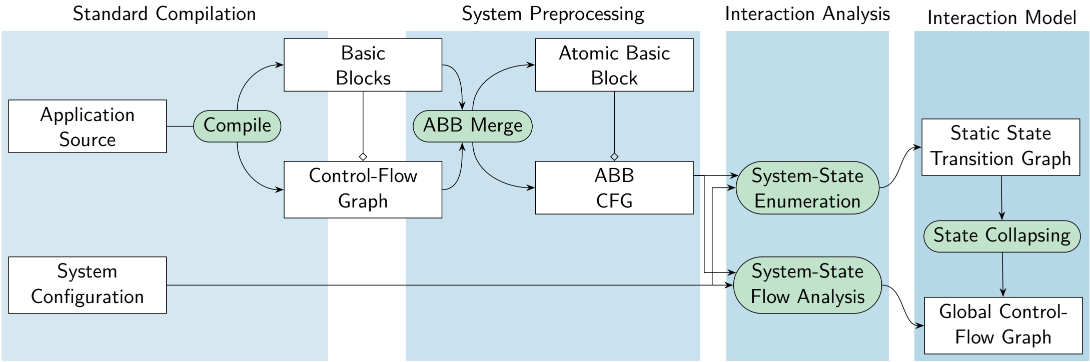


Figure 3.2 – Fine-Grained Interaction Analysis. For the interaction analysis, we use the application's compilation toolchain to extract a CFG, which gets coarsened to a CFG consisting of *atomic basic blocks (ABBs)*. The interaction analyses combine the ABB-CFG and the system configuration and calculate the interaction models.

In this chapter, I will explain the data structures and develop the methods to calculate such interaction models (see Figure 3.2): Starting with the application source code, we extract its logic as a *control-flow graph (CFG)* (Section 3.3.1) with basic blocks as nodes. Since large parts of the code do not influence the RTOS, we can coarsen the CFG and form *atomic basic blocks (ABBs)* by merging blocks that are indistinguishable from the RTOS' point of view (Section 3.3.2). Thereby, we speed up all subsequent analyses, as the number of CFG nodes drops drastically.

The combination of application logic and system configuration defines the interaction between RTOS and application. We capture this interaction with *static state-transition graph (SSTG)* (Section 3.4), which covers all possible system states and, thereby, provides a precise whole-system view. I calculate it in an abstract interpretation of the whole system: the *system-state enumeration (SSE)* (Section 3.4.2).

From the SSTG, we derive the more coarse-grained *global control-flow graph (GCFG)* (Section 3.5) interaction model, which lifts the CFG idea onto the system level. However, since the SSE suffers from the state-explosion problem (Section 3.4.3), I provide another, less detailed, analysis method that directly computes the GCFG: the system-state flow (Section 3.5.2).

3.2 State of the Art

Before I go into more detail about the GCFG and how to compute it for a given system, I want give an overview about other abstractions that are used to model the behavior of real-time threads and their interaction with the RTOS. For this, I will start with the related work that is close to the code level and includes little system semantic and proceed to the more abstract models.

Bertran et al. [Ber+06] propose a global system view that includes the application code, libraries, and parts of the operating system. By static analysis, they calculate a graph, which they also call "global control-flow graph", for a complex Linux-based embedded system. They connect system-call sites and library-call sites to their implementation in the kernel, respective in the libraries. Thereby, they form a global view of the control flow of a single thread and gain knowledge about that code a thread executes or that is executed on behalf of the thread in the kernel. As a result, they can exclude dead system-call handlers and dead library calls that are never called by the application from the system image. In their flow sensitive view, they handle the kernel as a higher-privilege–level application extension and ignore scheduling, context switches, and the semantic of the kernel execution.

Barthelmann [Bar02] uses static analysis of the application to calculate an *interference graph*, which describes possible preemptions between a code block and other threads. This calculation is done flow-insensitive and takes only the priority-driven scheduling and SRP critical sections into account. With the extracted knowledge, the author proposes an optimized *inter-task* register-allocation where the compiler minimizes the interference set between preempted and preempting code blocks. Afterwards, different versions of the context switch that save only parts of the register file are used to minimize the RAM overhead of saved thread contexts. While the analysis is flow-insensitive and only little OS semantic is used, this paper is one of the first attempts to take both, application and RTOS, in a generative whole-system optimization, into account.

For interrupt-driven systems, Regehr, Reid, and Webb [RRW05b] use abstract interpretation of the machine code to build an interrupt-preemption graph. While this analysis is precise and considers the control-flow of each interrupt handler, the analyzed systems had no RTOS and the threads were independent in their execution. However, the idea to follow the execution flow of the system along the physical time axis is similar to my approach of extracting the interaction graph.

On a more abstract level, the model checking community has proposed to formally grasp the application structure and combine it with the RTOS semantic in order to make statements about different system properties. Waszniowski and Hanzálek [WH08] models the OSEK OS standard for the UPPAAL model checker. For this, they used timed automata as the main abstraction to model all system components, taking also undirected and directed dependencies into account. The combination of application and RTOS model were used to verify different application properties, like the freedom from deadlocks and schedulability analysis of a gear-box controller. While they captured application structure in a flow-sensitive timed automaton, the model was manually extracted from the application and enriched by the application semantic.

Similar, Tigori et al. [Tig+17] uses *extended finite automata* of OSEK applications for model checking. Besides checking of general safety properties, they also used the UPPAAL model checker to find and remove unreachable kernel code. However, they give no hint how to extract the finite automata from the source code or how to ensure the equivalence of automata and source code. Furthermore, in both model checking methods, the actual interaction between application and RTOS is not calculated explicitly but is only implicitly calculated within the model checker. In the following chapters, I will present several methods that make beneficial use of explicitly calculated interaction models.

On the level of the RT domain, Bohlin et al. [Boh+08] calculates a preemption graph on the granularity of whole tasks. They derive possible preemptions between tasks that have release offsets and are scheduled with fixed-priority scheduling. However, their tasks are independent of each other, they ignore the application's inner structure, and they neither consider passive waiting nor external interruptions. Similar to this, other preemption analyses [DF04; Lee+01] on the granularity of whole tasks also ignore the application's inner structure.

From the related work, we see that there are two important dimensions about interaction analysis: The representation of the interaction as an interaction model and the method to fill this model with data for a specific application. In this work, I will use control-flow–graph structures as interaction model (similar to [Ber+06]) and, for some applications, derived finite-state machines (similar to [WH08; Tig+17]). These graph structures get filled by static analysis and abstract interpretation of an application model on top of an abstract RTOS model (similar to [RRW05b]).

This doctoral thesis is an extension and direct continuation of the topics that I touched in my Master's thesis [▷Die14]. There, I developed the foundational interaction analyses (SSE, SSF) and proposed three applications of the results that improve the kernel's run time and its resilience to transient hardware errors. I presented these methods in a conference publication [▷DHL15b] and in a journal publication [▷DHL17], which is an extended version of the conference paper.

As the resulting interaction models are the base for the interaction-aware analyses and optimizations, I will recap them and their general operation briefly in this chapter. In the following chapters, I will continue this line of research and investigate how we can further exploit the resulting interaction models. For this, I extend the focus from the RTOS alone to the whole system and consider important properties like timeliness, energy consumption, memory usage, correctness, and hardware integration. By widening the context, I am able to give much broader answers to my initial research questions (see Section 1.3).

3.3 Static-Analysis Techniques for Real-Time Systems

For a source-code level analysis of the RTCS implementation, we need abstractions to grasp the application logic on a higher abstraction level than individual instructions. As our interaction analysis takes place before run time and works without executing the system, it is a *static analysis*. Therefore, we can use abstractions from this domain and the closely related compiler and programming-languages domain. Mainly, I will use the notion of the *control-flow graph* and the *atomic basic block* abstraction.

3.3.1 Control-Flow Graph

Motivated by the prospect of optimizing compilers, the concept of *control-flow graphs (CFGs)* was introduced early in the area of compiler construction [All70]. These graphs capture the logic of a function on the level of sequentially-running code blocks and allows us to use a multitude of graph algorithms for the program analysis. As the CFG connects *basic blocks*, we start out with definitions for both concepts:

Definition 1 (Basic Block) *([ASU86, p. 529]) A* basic block *is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.*

Definition 2 (Control-Flow Graph) *([All70]) A control-flow graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow-paths.*

control flow

Both definitions are rooted in the notion of flow of control and control-flow paths. The term *control flow* stems from the control unit of the classical von-Neumann architecture [Von45], which controls the next machine instruction to be executed. So the control flow is the sequence of instructions that the processor executes directly after each other. Therefore, a control-flow path is one possible control flow through a program and visits a definite sequence of instructions.

In Figure 3.3a, we see a small example program that is grouped into basic blocks, which are connected in a CFG. For simplicity reasons, I use the LLVM [LA04] *immediate representation (IR)*, which is not yet on the level of machine instructions but resembles them closely enough. We see that the stream of instructions is already interrupted by labels (e.g., line 1 or line 10), which mark the targets of branch statements. These labels are not present in the resulting program, but they will resolve to the memory address of the following instruction, which can be used as a jump target.

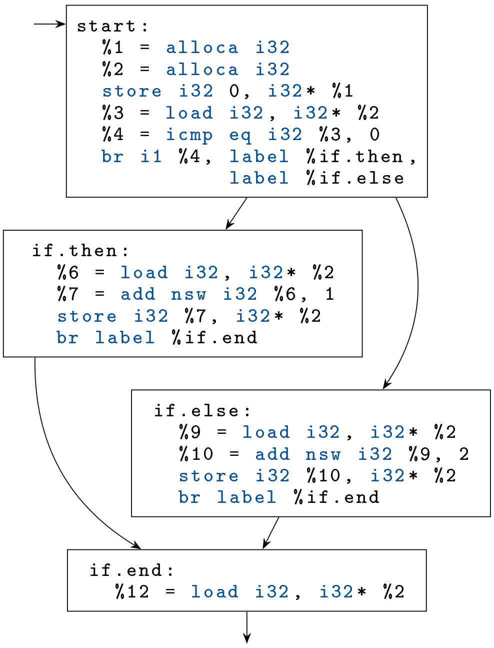
```
start:
  %1 = alloca i32
  %2 = alloca i32
  store i32 0, i32* %1
  %3 = load i32, i32* %2
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %if.then,
            label %if.else

if.then:
  %6 = load i32, i32* %2
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* %2
  br label %if.end

if.else:
  %9 = load i32, i32* %2
  %10 = add nsw i32 %9, 2
  store i32 %10, i32* %2
  br label %if.end

if.end:
  %12 = load i32, i32* %2
```

(a) LLVM Immediate Representation



```
start:
  %1 = alloca i32
  %2 = alloca i32
  store i32 0, i32* %1
  %3 = load i32, i32* %2
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %if.then,
            label %if.else

if.then:
  %6 = load i32, i32* %2
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* %2
  br label %if.end

if.else:
  %9 = load i32, i32* %2
  %10 = add nsw i32 %9, 2
  store i32 %10, i32* %2
  br label %if.end

if.end:
  %12 = load i32, i32* %2
```

(b) Control-Flow Graph

Figure 3.3 – Basic Blocks and Control-Flow Graph. An example program given in LLVM immediate representation of a program with a single branch. The branch has an then and an else statement.

As the basic block definition demands that the flow of control enters a basic block only at its first instruction, each label marks the beginning of a new basic block. Furthermore, in the example, all basic blocks end with a branch instruction even if they only jump to the following instruction. By this, LLVM can reorder blocks without changing the program logic.

In Figure 3.3b, the basic blocks of the example program are connected in a CFG. We see that the `start` block becomes the *entry block* of the CFG and end is an *exit block* that leads to the termination of the program. There are two control-flow paths through this program: `[start, if.then, end]` and `[start, if.else, end]`.

While the definition of basic blocks and the CFGs is simple, it allows room for interpretation. For basic blocks, we normally use the *maximal* basic blocks, which are the longest sequence of instructions that adheres to the basic-block definition. However, also smaller basic blocks, or even *minimal* basic blocks, which only contain a single instruction, are possible.

The other important dimension of CFGs is their scope, or level. As we defined the control flow by the stream of instructions, we must take a closer look at `call` instructions, which invoke other functions. While the processor proceeds to the first instruction of the called function after the `call`, the caller stops executing until the invoked procedure returns the control with the `ret` instruction. Figure 3.4 illustrates the situation: The control flow of the calling function suspends until the instruction stream of the callee reached the return instruction. From the perspective of the calling function, the call is a special instruction that produces all side effects of the called function.

CFG, ICFG

Hence, we define two different CFGs for the program. The *function-local* CFG follows the dashed line and describes only the control flow within a function. The *interprocedural control-flow graph (ICFG)*, or *thread-local* CFG, follows call instructions and describes possible control-flow paths through

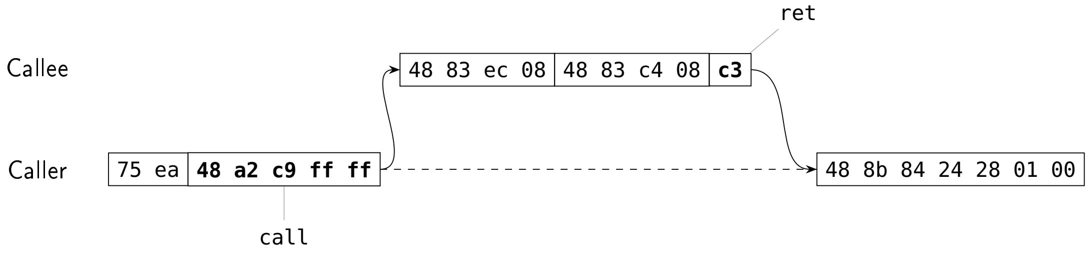


Figure 3.4 – Instruction Stream

different functions [SP81] as long as they remain in the same thread context. While the CFG contains only basic blocks from one function, the ICFG connects all blocks that are reachable from the thread entry function. In Section 3.5, I will define a third kind of CFGs that captures possible control flows across threads and which is the result of my interaction analysis.

3.3.2 Atomic Basic Blocks

Coming back to the whole-system view of the implementation engineer, our static analysis will be confronted with (too) many application basic blocks. For example, in Fiedler et al. [▷Fie+18], we looked at a GPS logging application on the base of an FreeRTOS system. Disassembled with `radare2`[6], the compiled binary contains around 20 000 instructions and 6429 basic blocks, which are located in 2654 functions. However, not every basic block contains a system call and thereby interacts with the RTOS. Therefore, we can improve on the efficiency of the subsequent static analyses if we virtually collapse system-call-free regions into *atomic basic blocks (ABBs)*.

From the RTOS's point of view, it is insignificant how an application structures its inner loops as long as they contain no system call. Actually, the RTOS does not even notice differences in this *fine structure* as it is activated, and only runs, in the exception; activated by an interrupt or an explicit system call. Therefore, we abstract from the application's fine structure without loss of generality but with improved analyses performance, since we can concentrate on the relevant parts of the application logic.

For this coarsening, I will use an adaptation of the ABB abstraction, which was introduced by Scheler et.al [Sch11; SS11]. In a nutshell, an ABB is a group of basic blocks that either contains a single system call or it contains only computational code that does not interact with the operating system. Thereby, ABBs are constructed such that they can be connected in a CFG that is a coarsened variant of the basic-block CFG.

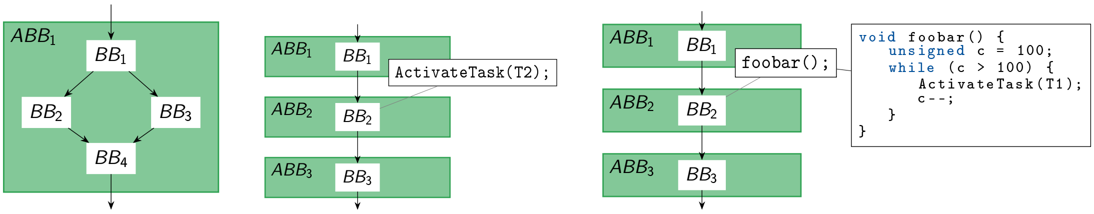
Definition 3 (Atomic Basic Block)

1. *An ABB is a single-entry–single-exit region of basic blocks that has one definitive entry block and one exit block; both might be equal.*
2. *ABBs can either be* computational *or invoke, directly or indirectly a single system call.*

Due to the second part of the definition, ABBs execute atomically from the perspective of the operating system. Leaving interruptions aside, a computation ABB, once started, will execute in a run-to-completion manner, before the RTOS can be activated in a subsequent ABB. Hence, ABBs are

---

[6] `radare2` is a professional free-software disassembler tool used in reverse engineering and provides various binary analyses, `https://www.radare.org`.



```
void foobar() {
    unsigned c = 100;
    while (c > 100) {
        ActivateTask(T1);
        c--;
    }
}
```

(a) Computation ABB **(b)** Split at System Calls **(c)** Indirect System Call

Figure 3.5 – Atomic Basic Blocks. Atomic basic blocks subsume multiple basic blocks, form a *single-entry–single-exit (SESE)* region, but they either contain computation or a direct or indirect system call.

able to subsume large parts of the application logic that are irrelevant for the interaction with the RTOS. At minimum, an ABB consists of one regular basic block.

For the ABB construction, and for my analyses in general, I *demand* that the application structure is static in those parts that are relevant for RTOS interaction:

Restriction 1 (Static Application Structure) *For every relevant basic block, the set of possible followup blocks has to be fixed and known ahead of time. For every relevant call site, the set of call target has to be fixed and known.*

For my implementation, I satisfied this restriction by forbidding the invocation of system calls via function pointers. With this assumption, the location of every system-call site is fixed, and we can statically partition the application code into system-call and computation blocks. In Section 3.6, I will discuss how it becomes possible to handle function pointers to system-relevant functions by employing points-to analyses.

In Figure 3.5, we see examples for ABBs and their context. For the ABB in Figure 3.5a, the basic block BB_1 is the entry block and BB_4 is the exit block. For the operating system it is irrelevant, whether the application takes the left or the right branch of the conditional. In Figure 3.5b, we see a situation where a system call is neighbored by computation code. In the normal function-local CFG, all three basic blocks would be merged into a single one, in order to get a maximal basic block. However, due to our second part of the definition, we have to split the code into three blocks and put each of them into its own ABB. Similar to this, we also have to split a call to a function that invokes a system call into its own ABB (see Figure 3.5c).

ABB construction

In order to transform an RTA into its ABB form, we have to perform three steps: (1) identify *system-relevant functions* that directly or indirectly invoke a system call. (2) split basic blocks at system-call sites or at call sites to system-relevant functions. (3) form SESE regions that fulfill the ABB definition.

For the first step, we perform an iterative fixpoint analysis on the call graph. The call graph is a directed graph, where each function becomes a node and an edge indicates a caller–callee relationship between two functions. For the fixpoint analysis, we start out by marking all functions that directly invoke a system call as system relevant. Iteratively, we mark all functions that have an edge to a system-relevant function as system relevant themselves and visit their predecessor nodes again. We repeat this process until no further changes to the system-relevant flags happen. As the update function is monotonous, we have to perform at most as many iterations as we have nodes.

After we have split the basic blocks and isolated the system calls, we have to find SESE regions. For this, we have several possibilities. Scheler [Sch11, p. 66] proposed an approach based on *interval analysis* [Sha80]. In essence, the proposed method starts with a minimal ABBs, which enclose only a single basic block. These initial ABBs are iteratively merged into larger ABBs until no further merging is possible. For this, Scheler defines a set of known patterns that are SESE regions and identifies them in the current ABB control-flow graph. Figure 3.5a is one example of such a pattern. If the identified SESE region adheres to the definition, it is merged into a single ABB.

The second variant to find ABBs is based on the *dominance relation* in control-flow graphs:

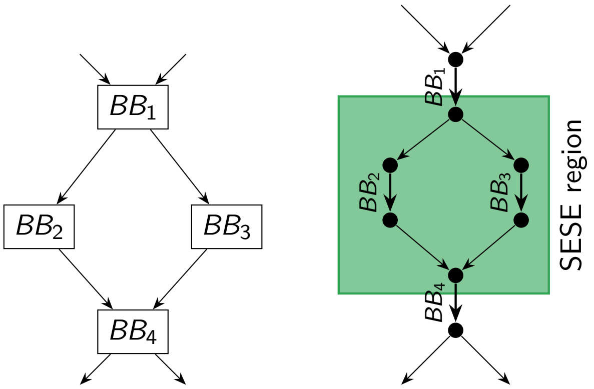
Definition 4 (Dominance Relation) *[Sha80] In a control-flow graph, node x is said to* dominate *node y if every path from the start node to y includes x. A node x is said to* postdominate *a node y if every path from y to the end node includes x.*

To illustrate this definition, I give a few examples of dominance and postdominance: (1) The entry block of a function dominates all other blocks of a function. (2) If there is only one return block in a function, this block post dominates all other blocks. (3) A loop-header block in a while-loop dominates the whole loop body.

single-entry single-exit region

As Sharir [Sha80] states, it is necessary for a SESE region that the entry node dominates the exit node and the exit node postdominates the entry node. This means that if the control-flow enters the SESE region through the entry node, it must also exit it through the exit node. Furthermore, Sharir gives a third, sufficient, condition that every cycle in the graph that includes the entry node also must include the exit node. We need this third condition, as back edges that start after the SESE region and jump into the region do not alter the dominance relationships.

From these three conditions, we can derive another ABB merge strategy: We wrap every basic block into a minimal ABB and continuously merge ABBs until no further merge is possible. First, we calculate the dominance and postdominance sets ([LT79; CHK01]) for each ABB. We then identify candidate regions by iterating over all nodes and considering each of them as a possible entry node. If a node is in the postdominance set of one of its own dominated nodes, we have found an entry and an exit node. Then, we check if the third SESE condition holds (no edges from the outside enters the region), and if all blocks in the region are computational (ABB condition 2). In contrast to Scheler [Sch11], the dominance-based method also works on CFGs with irreducible regions[7] and, therefore, achieves a denser ABB graph.



(a) Original Graph **(b)** Expanded Graph

Figure 3.6 – Node Expansion for SESE Regions and ABB Merging

The third option uses the program-structure tree [JPP94], which is a tree of SESE regions. Scheler [Sch11] dismissed this algorithm as Johnson et. al have a slightly different definition of SESE region

---

[7] A CFG becomes irreducible if it contains at least one loop with multiple entries [All70]. In such CFGs it is problematic to identify high-level structures. For an overview about irreducible C programs, please refer to [SW12].

where a region must be entered and left by one edge instead of a single entry/exit node. However, we can circumvent this issue by replacing every node by two nodes a and b with a single edge $a \rightarrow b$ between them. All incoming edges to the original node are attached to a, all outgoing edges are attached to b. This node expansion step was also described by Johnson, Pearson, and Pingali [JPP94, Figure 8], but not set into context. Figure 3.6 exemplifies the node expansion in the context of finding ABBs.

As the performance of the ABB merge phase was sufficiently fast during all my experiments, I implemented only the first and the second variant, leaving out the linear-time algorithm [JPP94]. Nevertheless, as the second ABB formation strategy is also based upon the dominance property, the results are equal.

With the ABB construction done, I am able to express code of the real-time application as a set of function-local CFGs that have ABBs as nodes. The control-flow edges between these ABBs express the existence of a control-flow path between the exit basic block of the predecessor and the entry block of the successor.

3.4 Static State Transition Graph

Based from the (coarsened) control-flow structure, I will describe two static analyses, a precise and a fast one, to infer the potential RTOS–application interaction. The precise analysis uses an abstract interpretation of the application structure, while the fast one is a fixpoint data-flow analysis.

Abstract interpretation [CC77] is a common technique for the static analysis of programs. For this, we transfer the program and its code to an abstract universe, which covers only those parts of the computation that we are interested in. In this abstract universe, we execute an abstracted version of program and record execution traces of possible program states. Thereby, the execution traces fork on every decision that cannot be statically derived, and we can record a state graph. Afterwards, we can learn from the combination of the potential execution paths about the possible behavior of the program.

For our scenario, I consider the whole system as the program-under-test and use the *abstract system state (AbSS)* as the central data type to describe a potential program state. This AbSS is the representation of the current state of the RTOS and the application in the abstract universe. As an abstraction for the system structure, we use the function-local ABB graph and an abstract model of the RTOS. For this, I use the OSEK-OS RTOS, like it is specified by [OSE05], as an example. The result of this abstract interpretation is the *static state-transition graph (SSTG)*, which combines all possible AbSS–AbSS transitions between threads as they are mediated by the RTOS.

3.4.1 Abstract System State

Before we define the AbSS, I will present an example application and its function-local ABB graph as it is the result of the ABB construction (see Section 3.3.2). In Figure 3.7, we see a system with three threads with ascending priority. The low-priority thread gets activated by some external event (not shown for simplicity), which indicates that new data is available. The thread reads data into a buffer and activates the high-priority thread if a newline was received. The high-priority thread prints the buffer and terminates itself afterwards. The medium-priority thread is never activated (in this example) and is, therefore, dead code. We see that the `ActivateTask()` system call splits the code of the conditional branch into two computation ABBs (ABB_2, ABB_4) and one system-call block (ABB_4).

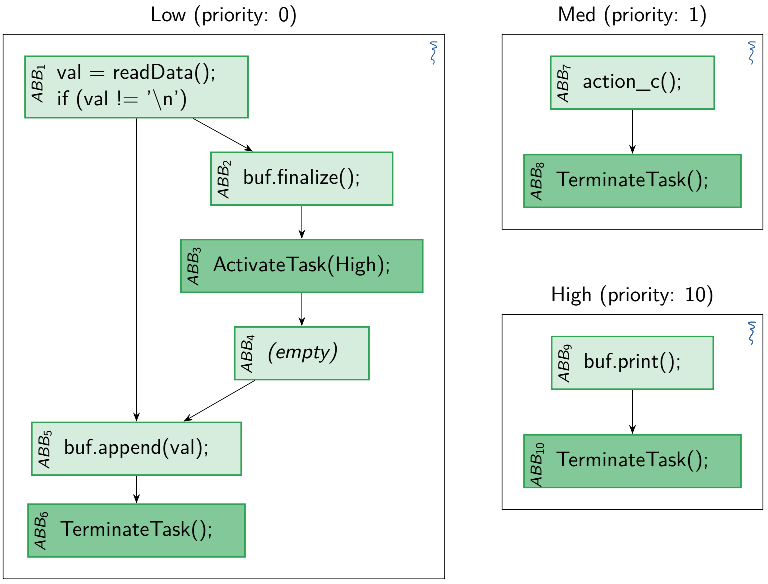
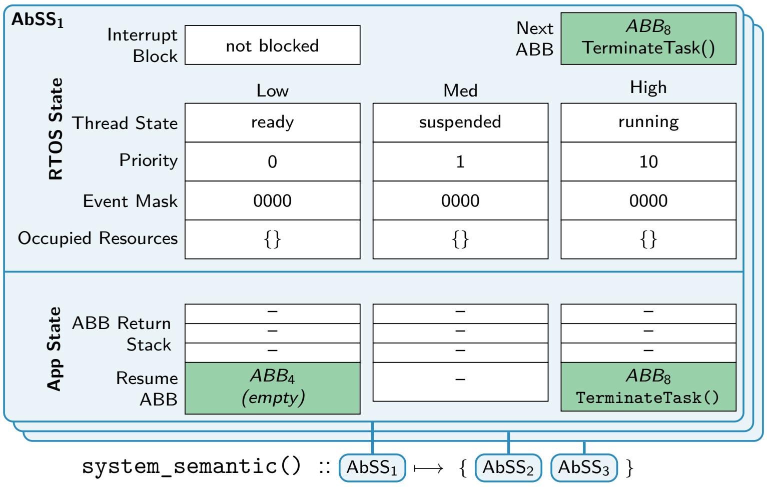


Figure 3.7 – Function-local ABB Control-Flow Graphs. Example system with three threads with increasing priority. The low-priority thread activates the high-priority in a conditional branch. Adapted from [▷Die14].



	Low	Med	High
Interrupt Block	not blocked		
Next ABB			ABB_8 TerminateTask()
Thread State	ready	suspended	running
Priority	0	1	10
Event Mask	0000	0000	0000
Occupied Resources	{}	{}	{}
ABB Return Stack	– – –	– – –	– – –
Resume ABB	ABB_4 (empty)	–	ABB_8 TerminateTask()

Figure 3.8 – Abstract System State. The AbSS captures the state of the real-time computing system at one point in time. Thereby, it considers the RTOS state as well as the call stack and the resume ABB of the application threads. Adapted from [▷DHL17].

For an abstract interpretation of an application in the context of a specific RTOS semantic (e.g., OSEK fixed-priority scheduling), we define the AbSS according to the investigated application. For this, the AbSS has to hold the relevant information that influences the RTOS behavior, as well as the relevant part of the application context that is required to determine the next ABB that is executed in the context of a thread. Figure 3.8 depicts an AbSS for the example system, and we see the division into state that is relevant for the RTOS and state that is relevant for the execution of the application.

application state

For the application state, we store, for each thread, the resume ABB, which indicates the next ABB that would be executed by that thread in the next step. In the example, thread Low would execute ABB_4 next, while thread High would invoke a `TerminateTask()` system call. Furthermore, as we use function-local ABB graphs, we need to record a (condensed) call stack to determine the next ABB after a function returns. This call/return stack is also the reason, why we have to split ABBs not only at system calls but also at function calls to system-relevant functions. All in all, the application state is a condensed representation of the (saved) program counter and the execution stack of the threads.

RTOS state

For the RTOS state, we have to store all information that can influence scheduling decisions and the interrupt relevant part of the processor state. For each thread, we store its current thread state, which indicates whether the thread is suspended (=terminated), ready to run, sleeping for an event, or currently selected for execution. This state is set by the scheduler according to occurred system calls and other parts of the RTOS state. We also use a priority field to store the current dynamic priority, which is influenced by the currently occupied SRP resources. Furthermore, each thread has a bit mask that holds the currently signaled events and controls the blocking behavior for `WaitEvent()` system calls. Besides this abstract version of the thread control block, we also use a Boolean flag to indicate if interrupts are currently accepted by the system. The global "next ABB" field indicates the ABB block that the system will execute next; it is only a shortcut for the resume ABB of the currently running thread.

IRQ handling

Besides the normal threads, we also have to handle ISRs and OSEK alarms in the AbSS. As we have discussed in Section 2.1.2.1, interrupt handlers and threads are both activities that are managed by the RTOS. Therefore, we model ISRs as pseudo threads and add another column for each ISR in Figure 3.8. By this unified view on interrupt handlers, we are able to handle interrupt semantics, like nested interrupts or interrupts with a lower priority than some thread, by lowering the priority of the pseudo ISR thread. As alarms are normally driven by a timer interrupt, we model all OSEK alarms with one artificial ISR, which activates threads that should be activated periodically.

For OSEK systems, it is possible to define the AbSS as a fixed-sized datatype (if we neglect the call stack for now), as all system objects are declared in the OIL file. For systems that can dynamically create threads, the AbSS has to reflect this dynamic. However, as we will discuss, later, such dynamic systems are anyhow much harder to analyze by abstract interpretation as the number of system states can easily become unbounded. Coming back to the AbSS data type, we define it according to a specific system configuration, create accessors for the data fields, and provide an operation to *compare* two AbSS instances for equality.

As the AbSS holds the threads' return stacks, we are able to carry out a flow-sensitive analysis of the RTOS–application interaction. However, it is noteworthy that the AbSS does not contain any timing information (there is no t_A). Therefore, my interaction analysis is flow sensitive, but (largely) *timing invariant*.

3.4.2 System-State Enumeration

The second part of the abstract interpretation is the execution of the program in the abstract universe. For the interaction analysis, the "program" is the combination of the application logic (ABB graph), the application parameters (OIL file), the kernel semantic, and an environment model that describes when interrupts can occur. We encapsulate this combination in the `system_semantic()` function, which produces a set of followup states for a given input AbSS:

$$\texttt{system\_semantic} :: \text{AbSS} \longrightarrow \{\text{AbSS}\}$$

The produced followup states are states that the whole system can potentially transition to during or after the execution of the input state. Thereby, all followup states are equally likely and all possible from the analyses' perspective.

We divide the `system_semantic()` function into two phases: (1) the `execute()` function applies the semantic of the currently executed ABB and produces several followup states. (2) the `schedule()` function updates the thread states and determines a new currently executed thread by applying the OSEK scheduling rules.

system-call blocks

The `execute()` function captures the influence of the currently executed ABB, where different ABB types have different impact on the AbSS: For system-call ABBs, we produce exactly one followup state as OSEK has a deterministic system semantic that reflects the effect of the system call on the AbSS. Thereby, these system-call influences reflect the specified behavior of the OSEK standard, and they are independent from concrete kernel implementations.

Listing 3.1 WaitEvent in the Abstract Universe. The execute function produces one followup AbSS for each system call. For `WaitEvent()`, we decide upon the system-call argument and the already signaled events if the currently running thread becomes blocked.

```
set<AbSS> execute(AbSS input) {
   ....
   // Extract information from the state and the syscall
   Thread active = input.running_thread();
   ABB abb = input.next_ABB();

   if (abb.type == WaitEvent) {
      EventMask wait_mask = abb.wait_mask;
      EventMask set_mask  = input.event_mask(active);

      // In most parts, the return state is equal to the input state
      AbSS ret = input.copy();

      // If no event that we wait for is set, block.
      // otherwise: continue to next ABB
      if ((wait_mask & set_mask) != 0) {
         // immediate return
         ret.set_resume_ABB(active, abb.successor);
      } else {
         // block thread
         ret.set_thread_state(active, BLOCKED);
      }

      // Produce only one followup state
      return [ret];
   }
   ... // Other system calls, computation blocks, and function calls
}
```

Listing 3.1 depicts a part of the `execute()` function for the `WaitEvent()` system call to exemplify the derivation of followup states. First, we extract information about the currently running thread and the ABB that is active in this AbSS (`next_ABB()`). If the current ABB is a `WaitEvent()` system call, we copy the input state as starting point for the followup state. We also extract the system-call argument `wait_mask` from the system-call site.

In order to be able to make this argument extraction statically, I put another restriction on the usage of system calls:

> **Restriction 2 (Static System-Call Arguments)** *All system-call arguments that reference another RTOS object must be statically derivable from the call site.*

Examples for such RTOS-object–referencing arguments in the OSEK API are printed in SMALL CAPS in Table 2.3.

After we have extracted, for the current thread, the `wait_mask` and the set of currently signaled events, we decide whether this thread must sleep or if the `WaitEvent()` system call immediately returns to the application. If at least one event in the `wait_mask` has already arrived, we set the resume ABB to the successor of the system-call ABB. Otherwise, we mark the thread as blocked and keep the resume ABB pointing to the `WaitEvent()` ABB in order to reassess the wait condition when another thread signals an event to the thread. At last, we return the prepared followup state.

For non-system call blocks, the employed system semantic is more complex as the influence of the application code itself must be captured. If the `execute()` function encounters a function-call block, it pushes the successor ABB of the function call onto the ABB return stack and sets the resume ABB of the currently executing thread to the entry ABB of the called function.

computation blocks

The most complex ABB influence stems from normal computation ABBs as they must capture the function-local control flows, as well as the influence of IRQs. For each successor in the function-local CFG, we produce a separate followup state and set the resume ABB of the current thread to the successor. For example, when executing ABB_1 in Figure 3.7, we end up with two followup states, one for each branch of the conditional, with ABB_2 and ABB_5 as resume ABBs respectively.

Furthermore, computation ABBs are also the blocks where the effect of IRQs take place. For each IRQ that can occur in a given computation ABB, we emit a followup state where the representative ISR pseudo thread is set to runnable. Similarly, we handle the pseudo ISR handlers for the OSEK alarm subsystem.

At first, this restriction of the IRQ occurrence to computation blocks looks like a restriction for implementation flexibility of the kernel as being non-interruptible. However, if we interpret a system-call ABB as the atomic point in time when the effect of the system call actually manifests, every IRQ can be either linked to the computation ABBs before or after the system-call block. Therefore, we ensure constructively that each system call is surrounded by computation ABBs and insert empty computation ABBs if necessary (see ABB_4 in Figure 3.7).

After the `execute()` function emitted a set of followup states, we apply the `schedule()` function on each state to reflect the scheduling semantic and to update the currently running thread. For this, `schedule()` recalculates the dynamic priorities to react to SRP resource acquisitions, determines if rescheduling to another thread is necessary, and updates thread states accordingly.

For the abstract interpretation, we start out by preparing an initial system state, which represents the system directly after boot. For example, all threads that are marked as `AUTOSTART=TRUE` in the OIL file are set to ready. Afterwards, we repeatedly execute the `system_semantic()` function on discovered states, beginning with the initial state, until we have discovered all possible system states. In order to detect states that we have already discovered, we use the *Comparable* property of AbSSs. As SSE result, we construct the *static state-transition graph (SSTG)*, which is a directed graph where the nodes are AbSS and the edges reflect the followup-state relation.

In Figure 3.9, we see the SSTG for the example system from Figure 3.7. In the example, we start with a state were High is suspended and Low is the currently running thread (red arrow). After this initial state, which executes the conditional block of thread Low, the SSTG forks, where the left branch directly executes ABB_4 and the right branch activates thread High (ABB_3). As thread

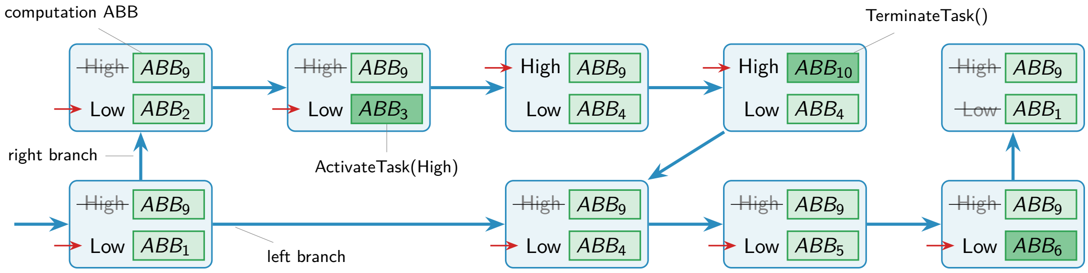


Figure 3.9 – Static State Transition Graph. The SSTG for Figure 3.7. AbSSs (blue) are simplified to show only the thread state: the red arrow indicates the running thread, suspended/terminated is strike through, otherwise the thread is preempted/ready. Furthermore, the resume ABBs are shown for each thread, while the color intensity (green) discriminates between computation and system-call ABBs. Blue edges indicate a SSTG transition.

High has a higher priority than Low, it is directly executed after the `ActivateTask(High)`. After the `TerminateTask()` of thread High, both branches merge and the system proceeds to the stable state where all threads are suspended as no interrupts are declared for this system. In later chapters, I will show and discuss more complex SSTG examples that also include interruptions.

3.4.3 The State-Space Explosion Problem

One general problem of abstract interpretation and, therefore, also for the SSE is the explosion of the state space [Val98]. The problem is rooted in the number of possible states which is potentially the cross product of the domains of all fields in the abstract program state. For our AbSSs, alone the thread state field with its four possible values (suspended, ready, running, blocked) adds a significant factor of $4^{\#threads}$ to the size of the potential state space.

However, there are several constraints that already restrict the set of *valid* AbSS. For example, a state where all threads are activated and the lowest priority thread is marked as running is not valid according to the OSEK semantic. Furthermore, the application logic itself restricts the state space: Since the function-local CFG constraints the possible ABB sequences, not all combinations in the resume-ABB fields are possible. For example, in Figure 3.9, it is impossible to be in a state where the thread High is running in ABB_{10} (`TerminateTask()`) and thread Low is ready at ABB_1, although this would be a valid state according to the OSEK semantic. However, the actually observed state space explosion for real OSEK systems with the SSE analysis remains a problem.

As my experiments showed [▷Die14], the biggest contributor to the actually reached state space are the interrupts; they can occur in every computation block and potentially fork the SSTG for every interrupt source. Therefore, I will describe two different measures that ease the IRQ-induced state-space–explosion problem.

3.4.3.1 Constraints from the Real-Time Domain

The first measure is a top-down approach where we incorporate more information about the possibility of IRQ occurrences from the RT domain. In general, this strategy to state-space reduction for abstract interpretation is called *guided construction-time reduction*[Val98, sec. 6]. As interrupts are used to implement periodic and sporadic events, which have periods and minimum interarrival times, the occurrence of interrupts will be restricted for the final real-time system. However, as the

SSE is deliberately time insensitive, we cannot use a quantitative statement about the minimum time span between two interruptions, but we have to use more qualitative constraints.

One example of such a qualitative statement is the period–equals–deadline assumption, which states that the period of a task must be longer than its execution time in order to meet its deadline. We can use this RT-domain knowledge to restrict the generation of IRQ transitions in the `execute()` function. If such a constraint is available, `execute()` inspects the AbSS whether a thread that implements the RT task is ready or currently running and emits an IRQ transition only if the associated threads has finished. Furthermore, we can also extend such a constraint to a group of threads whose activation is dependent on a certain interrupt source.

In order to give an intuition on the SSTG sizes, I applied the SSE to a realistic example application with 11 threads and 2 interrupts, which we discuss in more detail in Section 4.6.3 as the *i4*Copter. Without the IRQ constraint, the SSTG contains 1 563 169 states. If we utilize the knowledge from the RT domain, the number of states drops to 20 063 [▷DHL17]. As the interaction analysis itself is only mean but not the focus of this thesis, I refer you to the discussion in that paper about the additional impact of the incorporation of such a constraint.

Other possible incorporateable high-level facts are logic-of-action constraints. For example, if an external interrupt signals the completion of an asynchronous chore that is carried out by the hardware (like a sensor measurement), the completion can only be signaled *after* the chore has been handed over to the hardware. Therefore, we could restrict interrupts to occur only after a certain computation ABB has executed. However, I have not conducted experiments with such a constraint and it remains as a topic of future research.

3.4.3.2 Simplification of the Application Model

We can also lower the potential size of the SSTG by reducing the total number of computation ABBs in the application code, as every computational CFG node is a source of interrupt edges. Taking a look at the example SSTG in Figure 3.9, we see that the initial state and both successor states differ only in their computation ABB, but not in the RTOS state as no system call occurred. If we add one interrupt source (see Figure 3.10), we have to insert three interrupt edges from these states to disjunctive subgraphs, which eventually resume to the interrupted computation block. Most of the time, the three subgraphs that follow these interrupt edges will only differ in their IRQ-resume point.

For some applications of the interaction analysis, it is necessary to know where exactly a thread was interrupted and to which ABB it will resume to. For example, there is a difference between the situation where thread Low was interrupted in the left branch or in the right branch as it will invoke the `ActivateTask()` system call only in the latter case. However, there are applications that

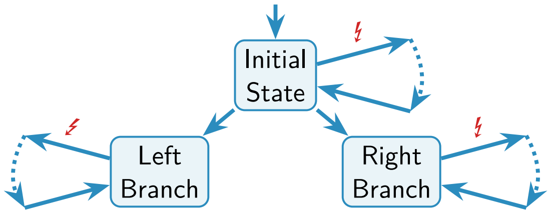


Figure 3.10 – State Explosion for Computation ABBs. As interrupts occur by default in every computation ABB, we get many regions interrupt-handling subgraphs in the SSTG that differ only in their application AbSS state but not in the RTOS state.

do not require this detailed knowledge; opening a trade-off between analysis time and precision. This possible precision trade-off is considered one of the strengths of abstract interpretation [CH94].

application state machines

For applications with lesser demand for precision, we “coarsen” [Val98, sec. 7.1] the granularity of computation nodes and transform the CFG to a finite state machine that emits system calls on each transition. These *application state machines (ASMs)* produce the same sequences of system calls like the original CFG, however the number of interruptible nodes is drastically reduced. The construction of ASMs as an optimization to the SSE was not part of my Master’s thesis but was first presented in Dietrich and Lohmann [▷DL17].

As intuition, we collapse all computation ABBs between system-call blocks into a single application state. For this, we greedily merge all reachable computation blocks in a depth-first search that starts at a system-call block and stops before the next system call. The transitions between these merged states are then drawn and labeled according to the invoked system calls at their boundaries. An example for this application-logic compression is shown in Figure 3.11 for thread Low of the example system. Here, ABB_1, ABB_2, and ABB_5 result in the L1 ASM state, which is interruptible and is left by invoking a `TerminateTask()` or an `ActivateTask()` system call.

The ASM is a generator that emits one system calls towards the RTOS on every transition. If the example ASM is in state L2, it can only emit a `TerminateTask()` system call when transitioning to the final ASM state L3. As the ASM generates the same sequences of system calls, it is *trace equivalent* with respect to system calls to the original CFG.

Another, more graph-formal algorithm to form the ASM from the CFG executes in three steps: (1) We convert the CFG to its *line graph*, where each vertex becomes an edge and every edge becomes a vertex, and use the ABB contents as labels. (2) We replace all computation ABB labels with ε transitions such that only system calls are labels on edges. (3) We minimize the ASM by applying standard ε-elimination in forward direction [HMU01, cha. 2.5.5]. With this description, we see that the system-call ordering of the original CFG and the ASM are equivalent, as all three steps keep the relative order of the system calls intact.

With the ASMs, we execute the SSE but store ASM states on the ABB return stack, in the resume ABB fields, and in the next ABB field. During the SSE, the `system_semantic()` produce an followup state for every outgoing ASM-state system-call edge and iterates over all interrupts that can currently in the current ASM state. Since the number of ASM states is significantly smaller than the number

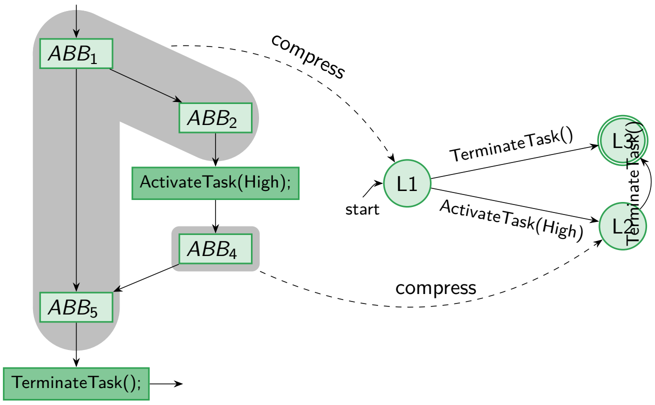


Figure 3.11 – Compression for Application State Machines. By merging regions of ABBs that are reachable in a depth-first search without invoking a system call, we create the compressed trace-equivalent ASM.

of computation ABBs, we end up with a much smaller SSTG. Combined with the IRQ constraint from the Section 3.4.3.1, the *i4*Copter SSTG decreases even further from 20 063 states to 4834 states [▷DL17]. In Chapter 7, I will use these ASM-derived SSTGs to integrate the RTOS behavior into the hardware.

3.5 Global Control Flow Graph

The SSTG draws a very detailed picture of interaction between the threads and the operating system, as two states are distinct if any field in the OS state differs. Due to this fine-grained distinction, the SSTG has a high demarcation power about situations that look similar on the surface (e.g., ready list is equal) but will behave different in the future (e.g., event is already signaled vs. it is still cleared). However, not all users of an interaction model require this high level of detail, but they would only suffer from the large number of SSTG states. Therefore, I developed the *global control-flow graph (GCFG)*, which can be obtained by collapsing the SSTG based on the currently executed thread and ABB, as a more coarse-grained interaction model.

inter-thread control flows

For the GCFG, we lift the CFG concept from the function-local and thread-local level to the system level. In Section 3.3.1, we defined a control-flow graph as a directed graph of basic blocks whose edges represent possible execution paths between code blocks. In other words, if there is an edge between two blocks the last instruction of the predecessor block can immediately be followed by the execution of the first instruction of the successor block. However, this notion of immediate execution succession is always tied to a certain execution context. For the function-local CFG, instructions from the same function body can follow each other. For a thread-local CFG, the whole thread becomes the execution context and control flows between function bodies are possible as long as they do not switch between threads.

Generalized to the system level, the GCFG includes all possible control-flow transitions between code blocks, even if they are executed by different threads. Thereby, the GCFG covers all possible scheduling decisions of the RTOS as well as function- and thread-local control-flow decisions with its edges. In Figure 3.12, we see the GCFG for our running example (Figure 3.7). While the control flow follows the application logic in the conditional (ABB_1), the interaction with the RTOS becomes visible when a system call is invoked (ABB_3): Since the activated thread High has a higher priority than the currently-running thread Low, the scheduler immediately dispatches to ABB_9. Therefore, the control-flow edge between ABB_3 and ABB_4 is not present in the GCFG (marked as gray) as High must terminate (ABB_{10}) before Low gets resumed ($ABB_{10} \rightarrow ABB_4$).

interrupt edges

Special consideration in the GCFG must be put on the processing of ISRs. While all GCFG edges in Figure 3.12 are synchronous to the current execution flow (conditionals, synchronous system calls), interrupts can occur after each executed machine instruction. A strict enforcement of the possible-execution-path semantic would require that all GCFG blocks become minimal blocks, which contain only a single instruction. This would diminish the utility of the GCFG as the number of nodes and edges would explode drastically. Therefore, I use a relaxation for interrupt-induced GCFG edges that still captures the IRQ influences but keeps the GCFG at a reasonable size.

For each interrupt source and for each ABB, we draw only one *interrupt edge* from the computation ABB to the ISR's entry ABB. This edge indicates that a specific IRQ can occur at any given time during the execution of the ABB. However, it does not indicate that some instructions are more likely to be interrupted than others. This relaxation is possible as computation ABBs do not modify the RTOS state during their execution.

If the user is only interested in thread–thread and thread–OS interaction, we can condense the GCFG even further by cutting out the ISR execution. The precondition for this even denser view is

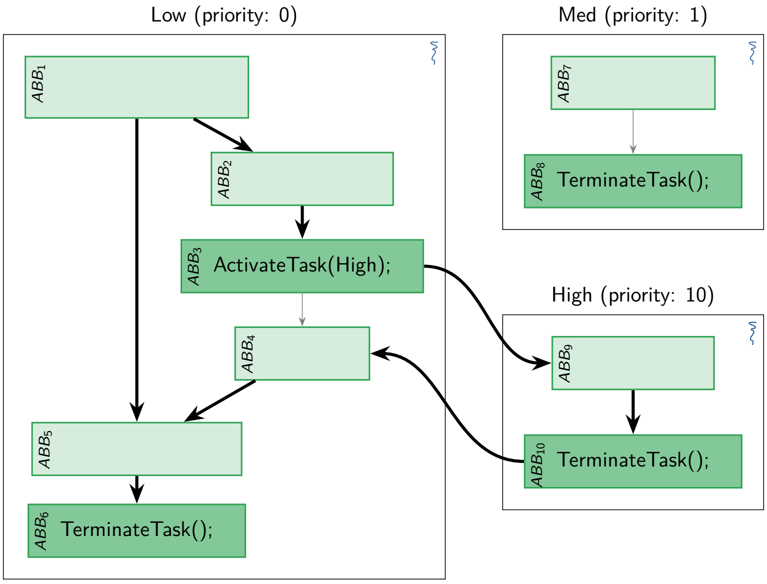


Figure 3.12 – Example Global Control-Flow Graph. Global control-flow graph for the example system from Figure 3.7. Adapted from [▷Die14].

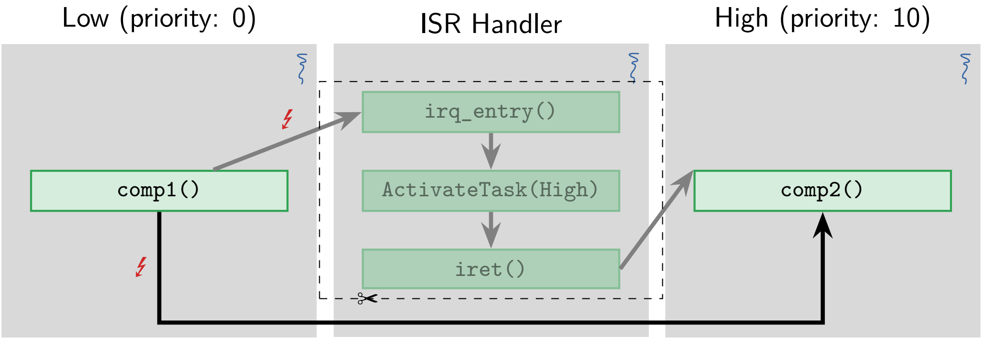


Figure 3.13 – GCFG without ISR Regions. For some GCFG users only thread code is of interest and we cut out IRQ regions. However, the interruption edge still can occur at any point in `comp1()`.

that the RTOS executes ISRs non-preemptively in a run-to-completion fashion. In such a system, each ISR execution forms a connected region of the GCFG (see Figure 3.13). While the activation of thread High is actually conveyed by the ISR, we can collapse the ISR code into one GCFG edge from `comp1()` to `comp2()`. In this dense view, we grasp ISRs as complex system calls that follow a computation block.

3.5.1 Code-Section–Based State Collapsing

We can derive the GCFG directly from the more fine-grained SSTG by state merging. For this, we define an equality operator $eq(a, b)$ that partitions the SSTG states into equivalence classes; each class gets collapsed into one GCFG node. The simplest $eq()$ operator distinguishes two states iff they differ in their next ABB field.

$$eq_{\text{simple}}(a,b) := (a.\text{next\_ABB}()) \equiv (b.\text{next\_ABB}())$$

With the equality operator, we partition the SSTG state space V_{SSTG} and create one GCFG node V_i per partition p_i. Thereby, the common next-ABB field becomes the referenced ABB block for V_i. We draw edges (V_i, V_j) between nodes if at least one transition between two states from the corresponding partitions exists.

$$\begin{aligned} P &= partition(V_{SSTG}, eq_{\text{simple}}) \\ V_{GCFG} &= \{V_i \mid p_i \in P\} \\ E_{GCFG} &= \{(V_i, V_j) \mid S_i \in p_i, S_j \in p_j : \exists (S_i, S_j) \in E_{SSTG}\} \end{aligned}$$

With eq_{simple}, we get the smallest but least detailed graph that is a GCFG as it captures all possible control flows between all code blocks. Every ABB occurs only once in the GCFG, even if it is executed in the different thread contexts. As this most simple GCFG form is rarely useful, we use a more specific equality operator that also takes the currently running foreground thread into account:

$$\begin{aligned} eq_{\text{std}}(a,b) := {} & (a.\text{next\_ABB}(), a.\text{running\_thread}()) \\ & \equiv (b.\text{next\_ABB}(), b.\text{running\_thread}()) \end{aligned}$$

For this thesis, I will consider the GCFG produced by the eq_{std} equality operation as the standard GCFG. However, other even more fine-grained equality operators that take more AbSS fields into account are possible and have useful applications. For example, an operator that takes the dynamic priority of the currently running task into account produces an GCFG that additionally distinguishes between ABBs with SRP-induced priority changes. It is important to note that multiple GCFG nodes can reference the same ABB if we choose another operator than eq_simple.

family of interaction-models

In Section 3.4.3.2, we already got the choice between different SSTG "flavors" by executing the SSE with the CFGs or the ASMs. With the choice of the equality operator for the state-collapsing GCFG construction, we got a second degree of freedom and the capability to construct a multitude of GCFG flavors for the same system. Cutting out IRQ regions, on the SSTG *or* the GCFG level, adds even more freedom in the construction. In total, there is not only one SSTG or one GCFG, but a whole *family* of interaction models; each member having a different degree of specificity and a different focus, making them useful for different usage scenarios.

3.5.2 System-State–Flow Analysis

The GCFG merge construction is based on the SSE analysis, which is potentially of exponential run time. We used the result of the expensive SSE analysis, collapse states into a flow graph, and, thereby, loose information and precision. Therefore, a faster GCFG construction method that directly produces the (imprecise) GCFG without the (precise) SSTG detour is desirable.

With the *system-state flow (SSF)* analysis, I developed a regular fixpoint data-flow analysis that progressively uncovers GCFG nodes and edges in polynomial time. Thereby, the SSF does not produce the same GCFG as the merging approach, but trades in analysis time for a GCFG that contains some infeasible edges, which cannot occur in the actual application. Furthermore, the SSF produces a GCFG with cutout IRQ regions and uses the thread-local CFGs instead of the function-local ones.

For the SSF analysis, we have put two more restrictions on the thread-local CFGs:

Restriction 3 (SSF Analysis)

1. *Each ABB must be unique in the entire system such that the nodes of the thread-local CFGs are disjoint.*
2. *The dynamic priority of an ABB must be unambiguous.*

In combination, both restrictions result in thread-local CFGs where the currently running thread (`abb.running_thread()`) and the dynamic priority (`abb.priority()`) are static properties of the respective ABB instead of dynamic properties of the system state.

We can fulfill both restrictions for any system by virtual function inlining and (virtual) duplication of ABBs in a graph preprocessing step: For example, if a system-relevant function is called by two threads, we duplicate its ABBs and call each instance only from one thread. If an ABB is executed once with and once without a resource taken, we duplicate it to have two ABBs with unambiguous priority.

imprecise states

In a nutshell, the SSF propagates imprecise system states on the already discovered GCFG edges while it considers the thread-local CFGs in the handling of computation blocks. These imprecise system states are the basic data structure of the SSF, and they capture our often inconclusive knowledge about the RTOS state at a certain point. In essence, we introduce [*] values for the AbSS field (see Figure 3.8) to indicate our lack of precision. For example, where the normal AbSS always makes a precise statement about the ready state of a thread, the imprecise AbSS can also state that we do not know whether the thread is ready or suspended. Furthermore, the resume ABB of a thread becomes a set of ABBs that indicates that the thread will continue in one of the given blocks when resumed. Only for the currently running thread, the resume-ABB field will always hold exactly one ABB.

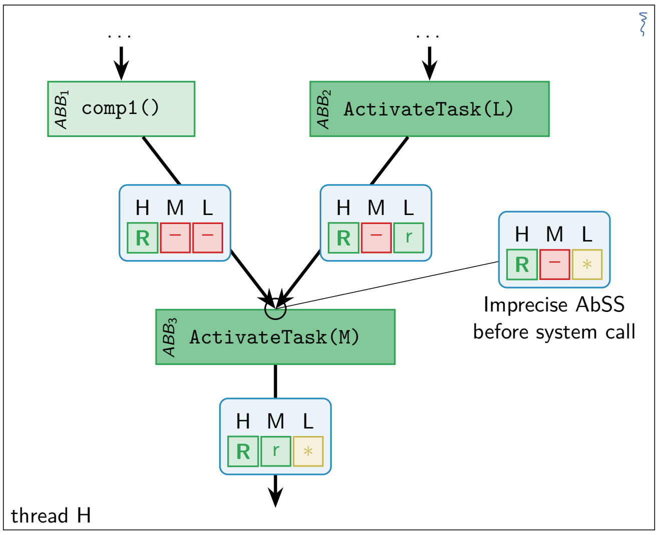


Figure 3.14 – Data-Flow of Imprecise AbSSs in the SSF Analysis. For a system with three threads (H, M, L) with respective priorities, the imprecise AbSSs (blue) "flow" on the already discovered GCFG edges into ABB_3. Since the ready state of L differs in both incoming edges it becomes unknown ([*]) in the outgoing AbSS.

Before I describe the SSF more formally, I want to give an intuition about imprecise AbSSs and their propagation along GCFG edges. In Figure 3.14, we see an intermediate step of the SSF analysis where imprecise are manipulated and merged. The considered system consists of three threads (H, M, L) and we see blocks from the currently-running (R) high-priority thread H. On each discovered GCFG edge, the SSF assigns one AbSS that becomes progressively more imprecise as the fixpoint iteration carries on. In the example, the ready state of thread L differs in the incoming edges as `ActivateTask()` set the thread L to ready (r) in the right branch. When the SSF processes an ABB, all incoming states get merged into a single imprecise AbSS, we apply the influence of the ABB, and the result is assigned to the outgoing edge(s). In the example, the state of L is [*] right before the execution of ABB_3 and thread M becomes ready (r) by the system call. The resulting AbSS will continue to flow (not shown) on the GCFG edges and thread L will remain [*] until thread H and M finish their execution. In this situation (not shown), the SSF will insert two GCFG edges, where one edge proceeds to the idle thread and the other jumps to the entry ABB of thread L.

Listing 3.2 The System-State Flow Analysis. The SSF is a data-flow analysis and adds the GCFG edges used for the traversal of the graph during the fixpoint analysis.

```
1  void SSF(AbSS initial_state) {
2    // System states are stored for blocks and for edges
3    map<Edge, AbSS> edge_states;
4    stack<ABB> working_stack;
5    // Set up the working stack and fake the inputs for the initial block
6    initial_abb = initial_state.next_ABB();
7    working_stack.push(initial_abb);
8    edge_states[initial_abb→initial_abb)] = initial_state;
9
10   // Run the fixpoint iteration until the working stack is empty
11   while (! working_stack.isEmpty()) {
12       ABB abb = working_stack.pop();
13       AbSS state_before = merge_states(edge_states[*→abb])
14
15       list<AbSS> followup_states = system_semantic(state_before);
16       for (AbSS next_state : followup_states) {
17           ABB next_abb = next_state.next_ABB();
18           if (gcfg_edges.find(abb→next_abb)) {
19              new_gcfg_edge(abb→next_abb);
20           }
21           if (next_state != edge_states[abb→next_abb]) {
22              edge_states[abb→next_abb] = next_state;
23              working_stack.push(next_abb);
24           }
25       }
26   }
27 }
```

The SSF analysis is a fixpoint data-flow analysis that can be described as a work-list algorithm (see Listing 3.2). It calculates the GCFG from the initial system state and uses an auxiliary data structure (`edge_states`) to assign an imprecise AbSSs to each discovered GCFG edge.

In the initialization (line 6f), we push the initial ABB onto the `working_stack` and assign the initial state as the incoming state flow for the first executed ABB. The analysis runs until the work list becomes empty and it processes one ABB from the stack in each iteration. For the examined ABB, we merge all AbSSs on the incoming GCFG edges (line 13) and apply an adapted `system_semantic()` function on the result. As the followup states point to other currently running ABBs, we add new GCFG edges if the followup ABB is not already present in the GCFG (line 19). Afterwards (line 22), we update the AbSS on the outgoing edge and push the followup ABB onto the working stack if the edge state has changed. During the algorithm, we gradually discover all GCFG edges, the recorded AbSSs become more and more imprecise, and we terminate the SSF if no edge state changes.

The lower precision of the SSF analysis but also its run-time benefits stem from the early merging of system states during the GCFG discovery. While the SSE analysis first discovers all states precisely, and we collapse them afterwards, the SSF performs the merge operation in each iteration without considering every state–state transition individually.

As already hinted, we need an adapted `system_semantic()` version that applies the semantic on imprecise AbSSs. First, we have to modify the `execute()` for system calls only slightly. For example, `ActivateTask()` adds the entry ABB of the thread to the set of resume ABBs if the thread is not already surely ready (i.e., [*] or suspended). For the synchronous semantic of computation blocks, we use the thread-local CFG successor instead of the function-local successor, whereby we eliminate the need for the abstract call stack.

The other adaption takes place in the `schedule()` operation to handle imprecise system states correctly. Listing 3.3 shows the schedule operation for a single imprecise AbSS. It emits a list of possible followup states and works in three distinct phases:

dispatch targets

First (line 2ff), we generate a list of candidate ABBs that are possible targets for the scheduling. As ABBs are unique to each thread, an ABB is statically associated with a thread. Alongside of each candidate ABB, we store if the associated thread is surely ready. For a [*] thread, all resume ABBs are not surely ready (line 8) as the thread could be suspended as well. For a ready or the currently running thread, only the blocks with the lowest dynamic priority are surely ready (line 15).

In the second phase (line 21f), we sort the list of possible followup ABBs according to their priority (`abb.priority`). Like the static association with the currently running thread, we know the dynamic priority statically due to the SSF requirements.

fuzzy scheduling

In the third phase (line 21ff), we start dispatching from the highest priority block downwards and emit one followup AbSS for each virtual dispatch. We cut off the dispatching at the first ABB that is surely ready as all lower priority blocks cannot be selected by the scheduler at this point. For an emitted AbSS, the target thread is set to be surely running (line 30) and it has only the target ABB as resume point. Furthermore, we remove the ABB from resume-ABB fields for all subsequently emitted AbSS as we already considered it in the current AbSS (line 35). If we remove the last resume ABB of a thread, it is known to be surely suspended in all lower-priority AbSSs. Intuitively, if we produce a followup state that dispatches to a low-priority thread, we know for sure that all higher-priority threads could not have been ready. The `schedule()` operation returns with a list of possible followup AbSSs (`return_abss`).

The special case in phase 1 for ready and running threads solves a problem involving two threads: For a surely running thread (A) with a high and a low priority resume ABB and a [*] thread with medium priority (M), we get a list of three possible ABBs: $[ABB_{A,H}, ABB_M, ABB_{A,L}]$. However, if we stop considering blocks after we have encountered the first surely ready block (line 41), we may only mark the blocks of the lowest priority as surely running for our scheduler:

$$[(ABB_{A,H}, \text{false}), (ABB_M, \text{false}), (ABB_{A,L}, \text{true})]$$

With the additional flag, we emit three followup states: one for each involved block since it is not sure that the first task will resume in the high-priority block.

IRQ handling

For the IRQ handling in computation blocks, the SSF takes a partitioned approach: For each ISR activation, we spawn a separate SSF analysis with the AbSS of the interrupted computation block as input state. The subordinate SSF analysis propagates the state on the ISR edges until an ISR exit block is reached, and we return the system state at the ISR exit block as the followup state of the interrupted computation ABB. Thereby, we automatically produce a GCFG with cutout

Listing 3.3 Scheduler for SSF analysis. The adapted schedule operation processes imprecise system states and returns a list of followup states.

```
list<AbSS> schedule(AbSS in_abss) {
  // Phase 1: Collect possible blocks
  list<(ABB, bool)> possible_blocks;
  for (Thread thread : all_threads) {
     if (in_abss.thread_state(thread) == [*] ) {
        for (ABB abb : in_abss.resume_ABBs[thread]) {
             // No [*] -thread can be surely ready
             possible_blocks.append( (abb, false) );
        }
     } else if (   in_abss.thread_state[thread] == READY
                || in_abss.thread_state[thread] == RUNNING) {
        min_prio = min({abb.priority | abb ∈ in_abss.resume_ABBs[thread]});
        for (ABB abb : in_abss.resume_ABBs[thread]) {
            // Only the block with the lowest dynamic priority is surely ready.
            surely_ready = (abb.priority == min_prio);
            possible_blocks.append( (abb, surely_ready) );
        }
     }
  }

  // Phase 2: Sort blocks by priority; highest priority to the front.
  possible_blocks = sort(possible_blocks, sort_by = abb.priority);

  // Phase 3: Dispatch to each ABB until the first surely running block}\label{line:phase3}
  list<AbSS> return_abss;
  for ((abb, surely_ready) : possible_blocks) {
      // Dispatch virtually to each possible ABB
      Thread th = abb.thread;
      AbSS next_abss = in_abss.copy();
      next_abss.thread_state[th] = RUNNING;
      next_abss.resume_ABBs[th] = set([abb]);
      return_abss.append(next_abss);

      // All lower-priority possibilities lack this resume point
      in_abss.resume_ABBs[th].remove(abb);
      if (in_abss.resume_ABBs[th].isEmpty()) {
         in_abss.thread_state[th] = SUSPENDED;
      }

      // Abort Dispatching, if we hit the first surely ready ABB
      if (surely_ready) {
          return return_abss;
      }
   }
}
```

IRQ regions. Before we start a subordinate SSF, we also check the interrupted system state for the required constraints to trigger the ISR (see Section 3.4.3.1).

If the interaction-model user requires a GCFG with IRQ regions, we re-insert the IRQ regions into the GCFG: During the subordinate SSF analyses we record all interruption points, ISR-GCFG edges, and the resumption ABBs and replace the interruption edge by the GCFG of the ISR body.

For the SSF, I choose this partitioned approach to avoid the problem of unnecessary imprecision due to superfluous state-merge points in the GCFG. To illustrate this problem, let us assume that we would execute the SSF in a non-partitioned fashion: As an IRQ can occur in every computation block, all computation blocks get an GCFG edge to the ISR entry block. As we propagate the states on already discovered GCFG edges, we would merge all system states at the ISR entry block. Even worse, since most ISRs resume eventually to their interrupted block, the ISR exit block would redistribute this completely-imprecise AbSS to all computation blocks. Thereby, all state precision would be lost and the SSF would become useless.

SSF complexity

As the exponential complexity of the SSE analysis was the starting point to develop the SSF analysis, I will give a coarse assessment of the SSF complexity: In the SSF, we have at most $\#\#ABBs^2$ system states, one for each possible GCFG edge. Each system state has $\mathcal{O}(\#threads)$ variables with a fixed domain and $\#threads$ resume-ABB sets, each set having a maximum of $\#ABBs$ items. In each monotonic analysis step, we change, in the worst case, at least one single fixed-domain variable to [*] or add a single ABB to a resume-ABB set. Therefore, we need at most $\mathcal{O}((\#ABBs \cdot \#threads) \cdot \#ABBs^2)$ iteration steps with polynomial complexity itself. In total, the SSF analysis is polynomial in the product of #threads and #ABBs.

3.6 Limitations of the Interaction Analysis

In the last sections, I presented two methods to calculate a family of interaction models, which all have some kind of control-flow semantic as, both, SSTG and GCFG edges constraint the possible execution paths through the whole system. In this section, I will discuss the limitations of the presented interaction analyses regarding the application and the system model of the analyzed real-time systems. As SSE and SSF are both static analyses and derive their results from system's inherent determinism, these restrictions are all related to the allowed degree of dynamic behavior.

function pointers

For the application logic, I put limitations on the flexibility of the used code constructs: First, I forbid the invocation of system-relevant functions through function pointers (Restriction 1) in order to have a statically inferrable call graph from the thread's entry function downward. This restriction allows us to statically determine caller–callee relations and construct the thread-local CFG.

However, we could lift this restriction by using an over-approximation of the call graph that references multiple callable functions at indirect call sites. This problem of extracting a call graph in the presence of function pointers is a well-studied static analysis problem [MRR04; Mur+98] that is closely related to the problem of points-to analysis [Ste96; SH97]. Nevertheless, as this problem also hinders other areas of static RT analysis (e.g., WCET analysis), the relevant standards (e.g., [04; ISO11]) for safety-critical systems already discourage the usage of function pointers.

system-call arguments

In Restriction 2, I forbid the usage of dynamically calculated system-call arguments if they reference a system object. By this restriction, the influence of a system call on the AbSS becomes deterministic and the `execute()` function emits only one followup state for every incoming precise system state.

We could loosen this restriction by allowing sets of possible values as system-call arguments and by invoking the `execute()` for every possible argument assignment. In order to restrict the set of possible values, we could apply value-range analysis [Har77] as another long-standing static analysis technique. Nevertheless, similar to the function-pointer restriction, the degree of freedom for system-call arguments is often already restricted by the RT domain. For example, for the SRP the set of claimable resources must be fixed for each thread in order to calculate the ceiling priorities. Therefore, also the system-call arguments for `GetResource()` and `ReleaseResource()` are limited to these sets in the implementation.

EDF

The other class of restriction applies to the system and scheduling model. The presented analyses only consider fixed-priority on-line scheduling mechanisms, which are, for example, parameterized by a rate-monotonic scheduling strategy. Systems that offer significantly less determinism, such as an

RTOS with an EDF scheduler or any kind of scheduler that performs time-based online acceptance tests, are much harder to grasp statically in their application–RTOS interaction. While a fixed-priority scheduling decision only depends on the enumerable RTOS state, an EDF decision depends on the progress of the physical time, which is an unbounded real-valued number. Thereby, the interaction model would become the cross product of the application control flow and the unbounded physical time axis. Nevertheless, for the domain of safety-critical embedded control systems, the focus on fixed-priority systems imposes little impact in practice as the relevant industry standards (such as OSEK/AUTOSAR, ARINC 653, μITRON, but also POSIX.4) employ fixed-priority scheduling.

multiple processors

Another limitation of the presented methods is their focus on single-processor systems. On the first view, it seems that this focus restricts the applicability to class of systems that will continuously lose importance in the real world. However, many, including the AUTOSAR [AUT13] standard, propose the usage of partitioned multi-processor systems. In such systems, which spawn a single-core RTOS instance on every processor, threads cannot migrate between cores and the core–core interaction is very limited. Thereby, they show less thread–thread interference and are more predictable and analyzable.

For such systems, which often provide only cross-core thread activations and event signaling, we can calculate one SSTG/GCFG for each processor and model the influence of all other RTOS instances similar to external interrupts. For a more integrated view, we could place the processor-local interaction models next to each other and express their reciprocal influence with a formalism that supports true parallelism, like Petri-nets [Pet63] or communicating-sequential processes [Hoa78]. However, the extraction of such a cross-core interaction model and its restrictive power on the processor-local interaction analysis are a topic of further research.

3.7 dOSEK: A Framework for Whole-System Analysis

For the implementation of my approach, I extended the dOSEK analysis and generator framework and integrated the presented interaction analyses. Originally, the dOSEK framework was developed for the "dependable OSEK" project [▷Hof+15], which was part of DanceOS (DFG SPP1500). In this research program, we investigated on software-based counter measures against transient hardware faults (bit flips) on COTS platforms. For this, we generated application-specific OSEK kernels that were several orders of magnitude more resilient against memory bit flips. Here, I will give a short overview (see Figure 3.15) about the operation of the generator framework, which we structured similar to a regular compiler.

Before we invoke the framework, the user compiles the application source with all dependent libraries to LLVM IR *immediate representation (IR)* (step 0). Thereby, the dOSEK implementation assumes that the source code is available. However, for the interaction analysis we only need the CFG structure, the system-call locations, and their arguments, which could also be achieved by analyzing the machine-code binary [The00].

system graph

In the first step, dOSEK splits the basic blocks (step 1) and finds ABB regions in the function-local CFGs. The front end stores information about the application structure (i.e., functions, ABBs, and basic blocks) in the *system graph*. This system-graph structure holds the static information about the currently processed application. Therefore, dOSEK enriches the system graph in step 3 with the real-time properties and information about other system objects (i.e., SRP resources, alarms,...).

In the middle end, dOSEK combines the information from the system graph in the interaction analysis (step 4), which I described in this chapter. As a result, we have the SSTG and the GCFG as interaction models of different accuracy.

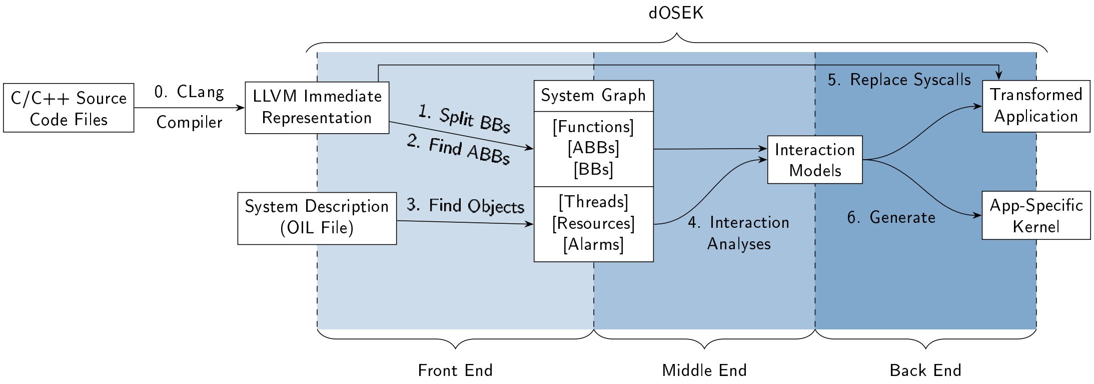


Figure 3.15 – dOSEK Generator Framework

In the back end, the kernel generator replaces all system-call sites in the application code by calls to placeholder functions (step 6). The implementations for these place-holder functions are filled by dOSEK as part of the application-specific kernel implementation. For example, dOSEK replaces `ActivateTask(T1)` system call with `ActivateTask_BB3(T1)` if the system call is located in the third basic block. By this isolation of the system-call sites, dOSEK can generate a specialized system-call variant at every call site. In the end, the generated kernel-source code and transformed application code are combined and optimized by a final whole-system compilation step.

While we started dOSEK to ease the run-time overheads of the bit-flip protection of the RTOS [▷Hof+15], it turned out to be a good substrate for whole-system analyses of static systems. With dOSEK, I have the possibility to perform elaborated analyses on the whole system and influence the kernel implementation, as well as the application code at the system-call sites.

3.8 Chapter Summary

In this chapter, I provided the techniques and means to statically capture the potential interaction between the RTOS and one concrete application into a family of control-flow sensitive interaction models. These techniques start out with the application code and use the *control-flow graph (CFG)* to capture the logic of the implementation. Since most parts of the application do not interact with the RTOS, I coarsen the CFG with the *atomic basic block (ABB)* abstraction, which subsumes single-entry–single-exit regions of code that run atomically from the RTOS' point of view.

Based on the coarsened CFG structure, the foundational interaction analyses capture the potential system behavior: The *system-state enumeration (SSE)* analysis enumerates all possible system states and produces the exponentially-large *static state-transition graph (SSTG)*, which can be collapsed into a *global control-flow graph (GCFG)* by different state-equality operators. In contrast, the *system-state flow (SSF)* analysis directly calculates the GCFG in polynomial time via a fixpoint data-flow analysis that progressively uncovers the GCFG by propagating imprecise system states on the already discovered GCFG edges. Both analyses are parametrizable with additional constraints from the RT domain, produce a family of SSTGs and GCFGs, and capture the RTCS on the implementation and post-mapping level with different degrees of precision.

Part I

Analysis

4

SysWCET and SysWCEC

Whole-System Analyses

> I have always been a quarter of an hour before my time, and it has made a man of me.
>
> in *The Gentleman's Magazine*, 1848, LORD NELSON

In the previous chapter, we have seen how the SSTG and GCFG capture the system-wide control flow that crosses, mediated by the RTOS, the thread boundaries. It is along these control-flow paths, which are sequentially processed by the CPU, that a system consumes important resources that are inherently tied to the application execution: time and energy. For real-time and resource-constrained systems, these dimensions are crucial and we must be able to give upper bounds for their consumption in order to give guarantees about the logical correctness of the system. However, many analysis methods stop at the thread boundary, analyze each thread in isolation, and accumulate results (and pessimism) in a post-processing step.

With SysWCET, I will present an integrated method that exploits the results of the interaction analysis to give tighter bounds for the *worst-case response time (WCRT)*. For this, I combine the execution-time analysis and the response-time analysis into a single problem formulation, instead of performing both steps separately. Applied to the *worst-case response energy (WCRE)* problem for systems with switchable peripheral devices, an adapted version (SysWCEC) drastically reduces the overestimation of the system's energy consumption.

Related Publications

[▷Die+17] **Christian Dietrich**, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. “SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems.” In: *Proceedings of the 23rd IEEE International Symposium on Real-Time and Embedded Technology and Applications*. RTAS ’17 (Pittsburgh, Pennsylvania, USA). Washington, DC, USA: IEEE Computer Society Press, 2017, pp. 37–48. ISBN: 978-1-5090-5269-1. DOI: `10.1109/RTAS.2017.37`.

[▷Wäg+18] Peter Wägemann, **Christian Dietrich**, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. “Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems.” In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems*. ECRTS ’18 (Barcelona, Spain). Ed. by Sebastian Altmeyer. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. DOI: `10.4230/LIPIcs.ECRTS.2018.24`.

4.1 Whole-System View on Consumable Physical Resources

With each executed instruction, a computing system consumes time and energy, which either passes and is lost forever or dissipates into the heat sink. For RTCSs, the logical correct operation does not only rely on the bug-free application code but also on the timely reaction to system-external events. For energy-constrained systems, the power drained during the calculation of this reaction elevates from a non-functional to a functional property, as a system cannot be logically correct if it has not enough energy to complete its work. Both resources, time and energy, are consumed on the system level as their consumption does not end at the thread boundary and threads can influence each other. In the context of real-time systems, it is desirable, for both problems, to identify one *worst-case path* through the whole system that becomes an upper bound for the respective resource consumption. Thereby, we can guarantee that a system will execute a task within a given time and energy bound.

For a safe *worst-case response time (WCRT)* – an upper bound for distance on the physical time axis between job release and completion – we must not only consider the mere computational requirements of the task, but also all other influencing system factors, like interruptions and kernel overheads. In contrast to the regularly used WCRT methods, which analyze tasks pessimistically in isolation and add the system overheads in retrospect, I propose SysWCET, an approach that exploits the whole-system view of the SSTG interaction model for an integrated response-time analysis. SysWCET does not only integrate all relevant system components in one problem formulation, but also allows us to consider inter-thread dependencies in the execution-path analysis.

Besides calculating upper bounds for response times, the SysWCET methodology proved to be also applicable to assess the energy consumption for systems with switchable peripheral devices that execute a task set. SysWCEC, which I will present in Section 4.7, is the adaptation of the SysWCET principle to the energy-consumption problem for systems with switchable devices. It was developed by Peter Wägemann [▷Wäg+18] and I had the pleasure to work with him on the adaptation.

Since Peter did most of the work, I do *not* consider SysWCEC as a main contribution of this thesis. However, it demonstrates the flexibility of the SysWCET approach and the benefits of a context-sensitive interaction-aware analysis for other consumable resources. The application to a second important resource by another researcher validates the SysWCET method and gives further insights into the properties of my interaction-aware analysis.

In the following, I will first explain problem with traditional worst-case response-time analyses and the SysWCET method, before I briefly dive into the application to the energy-consumption problem.

4.2 Compositional WCRT Analysis Considered Harmful

The WCRT analysis of individual real-time tasks is closely related to the timing, or scheduling, analysis of a whole RTCS. For a hard real-time system, a successful scheduling test states that all jobs finish on time between their release and the respective absolute deadline. For example, the scheduling analysis for a uniprocessor EDF system decides upon the processor utilization of a given task set whether the execution is feasible or not [LL73]. Many scheduling tests have a binary result for a given task set: it is either schedulable with the given resources or it is not. However, often it is useful to know how early before its deadline a job will finish; for example, to give lower bounds for the *slack time* of the response. For this, the WCRT is a quantitative upper bound for the response time of an individual task that is executed in the context of a whole system.

Most WCRT analysis methods divide the problem into two subproblems [JP86]: execution-time analysis and response-time analysis. First, they calculate an upper bound for the execution time (= the WCET C_i) of the task as if it would run in isolation on the given hardware. For this, they analyze the task's code structure and the underlying machine, and find the longest-running path through the task body. Afterwards, they accumulate the individual WCETs according to the given scheduling model and other real-time system parameters, like the period of each task. Thereby, the accumulation formula reflects the preemption and interruption situation for the *critical instant*, which denotes the worst chain of events that is possible during the execution of the task.

In order to exemplify this, I will explain a simple but representative WCRT analysis [JP86] for task sets of sporadic, preemptable, and independent tasks under a fixed-priority scheduling policy. Each task τ_i has a minimum interarrival time I_i, a given WCET C_i, and a fixed priority. Without loss of generality, I assume that a lower task index i indicates a higher priority and τ_0 has the highest priority in the system. Clearly, for task τ_0 the response time RT_0 is equal to its WCET C_0 as no other task is scheduled during its execution. For a low-priority task, the critical instant occurs when all higher-priority tasks occur as often as possible during the processing of the low-priority task. Therefore, we can give an iterative formula that converges to RT_i:

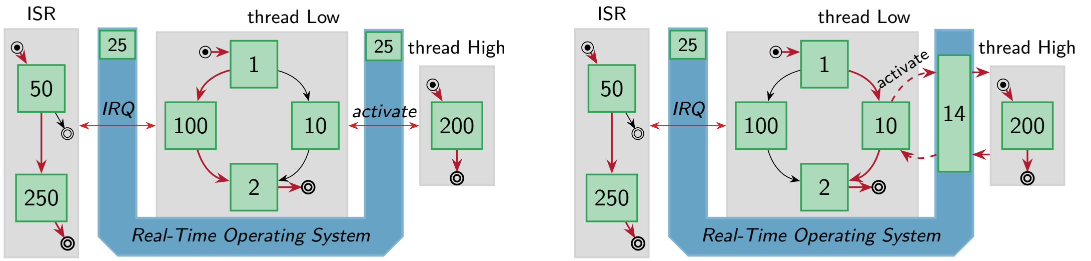
$$RT_i^n = C_i + \left(\sum_{j=0}^{j<(i-1)} \lceil RT_i^{n-1} / I_j \rceil \cdot C_j \right)$$

In each iteration, the response time is the sum of the task's own execution time C_i and the time that is spent in the execution of higher-priority tasks. For the latter part, we calculate the maximal number of occurrences for each higher-priority task τ_j from the previous response time RT_i^{n-1} and the minimum interarrival time I_j of the preempting task. For each possible preemption, we account for one full execution time C_j.

For more complex system models, the WCRT formula is extended by additional additive terms to account for other system interferences. For example, for systems with coordinated shared-resource accesses, we have to add a blocking term that accounts for the worst-case delay that is induced by the resource acquisition [LZM04]. Other important terms cover RTOS overheads like context switches and time that is spent in the interrupt handling. However, the general structure of WCRT analysis methods remains *compositional* as the accumulated delays are considered to be constant and independent of each other.

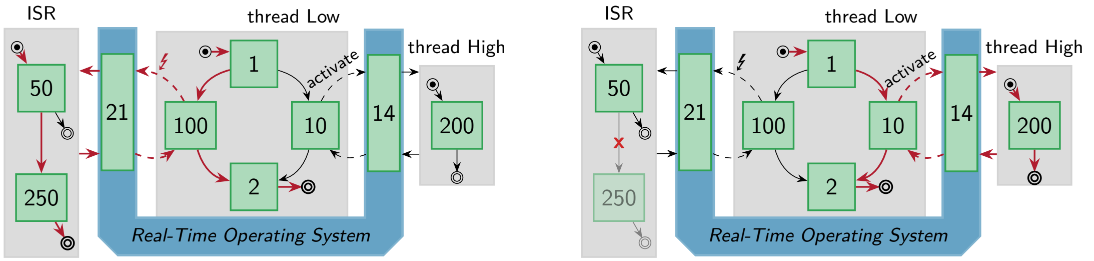
However, these compositional WCRT methods all share a common problem of accumulating pessimism, which leads to overly conservative upper bounds for the response time of individual tasks. The more pessimistic a WCRT estimate becomes, the larger is the distance between actual WCRT and the estimate. Such an overly pessimistic estimate forces the RTCS developers to overprovision the hardware in order to meet the theoretical provable requirements. In the introduction (Chapter 1), I already presented a simple instance of this problem. For illustration purposes, Figure 4.1 shows an extended version thereof that highlights different situations that are, due to their control-flow sensitivity, problematic for every compositional approach. For the flow-sensitive WCRT considerations, we switch from the RT domain (task) to the OS domain (threads) as we are interested in the WCRT of the actual implementation.

All four situations in Figure 4.1 show the same RTCS with different degrees of knowledge about the RTOS–application interaction. The system consists of two threads and one ISR: the low-priority thread has a conditional statement where the branches have a large difference in their execution time. In the faster branch, the low-priority thread activates the high-priority thread by a synchronous system call. An interrupt can occur at one location in thread Low and the activated ISR has two different interrupt-return points; one after 50 cycles and the other after 300 cycles. The overheads



(a) All interruptions and task preemptions are possible. WCRT: $300 + 103 + 200 + 4 \cdot 25 = 703$

(b) Thread activation occurs only in right branch. WCRT: $300 + 13 + 200 + 25 \cdot 2 + 14 \cdot 2 = 591$



(c) Interruption and activation are mutual exclusive. WCRT: $300 + 103 + 0 + 21 \cdot 2 = 445$

(d) ISR execution always exits early. WCRT: $0 + 13 + 200 + 14 \cdot 2 = 241$

Figure 4.1 – Control-Flow Sensitive Response Times. We can lower the upper bound for the response time if different control-flow sensitive constraints on the actual interaction are known. Here, we see a system that consists of two threads and one ISR in four different degrees of used interaction knowledge for the WCRT analysis. Each block contains its timing cost, the RTOS requires 25 time units in the worst case, and the edges in the respective WCRT case is indicated in red. Adapted from [▷Die+17].

of the RTOS are 25 cycles in the worst case for each activation. Furthermore, we are interested in the WCRT of thread Low, and we assume that thread High and the ISR can occur at most once during the execution of thread Low.

Figure 4.1a shows the knowledge that is considered by the compositional WCRT method and I highlighted those CFG edges that are visited in the longest execution path. Since we calculated the WCET of Low (103) and High (200) in isolation, we had to assume the worst-case execution path for both threads and the late return point in ISR (300). For the RTOS overheads, we use the usual pessimistic approach [Lv+09b] and add the longest path through the kernel for every context switch [Bla+11]. Therefore, we account for four full kernel activations (25), two for the synchronous preemption, and two for the ISR activations. Overall, a compositional approach would end up with a WCRT estimate of 703 cycles.

The first improvement (Figure 4.1b) of the estimate shows if we consider information form a GCFG with cutout ISR regions. As this GCFG includes all thread–thread preemptions, we see that the activation of thread High is located in the faster branch of thread Low. Due to this flow-sensitive directed dependency between both threads, their WCETs are no longer independent as C_{High} adds nothing to the response time if C_{Low} reaches is maximum in the left branch. Another aspect is the more precise information about the kernel run time for the thread activation. In Figure 4.1a, we accounted the maximal WCET of the kernel for each activation. However, since we now exactly know the exact operations that is performed in both kernel activations (preemption and resumption), we can give a much lower bound (14) for both transitions, leaving us with an overall WCRT of 591 cycles.

We can improve even more on the analysis, if we consider more information about the possible occurrences of the ISR (Figure 4.1c). For this example, we use the knowledge that the ISR can only trigger in the long-running block (100) of the left Low branch (e.g., due to a logic-of-action constraint). With this knowledge base, which is included in the standard GCFG, the flow-sensitive WCRT analysis will now consider the left branch as the more expensive situation since the ISR outweighs the cost of thread High. Again, as we know the interruption point for the ISR precisely, we can give a tighter estimate on the RTOS overheads. So, if we consider flow-sensitive thread dependencies and flow-sensitive interrupt constraints, we can tighten the WCRT even further to 445 cycles.

Besides system-level constraints that influence the occurrence and scheduling of activities, an application can also have logic constraints on its control flow. For example, if we know that the ISR will always take the early exit path in the examined situation (Figure 4.1c), the ISR's WCET becomes 50 cycles. Thereby, the left branch of Low is no longer the longest execution path as thread dependencies, interrupt constraints, and control-flow constraints interfere with each other. Compared with our initial compositional WCRT estimate of 703 cycles, we end up with an overall WCRT of 241 cycles.

Overall, the compositional WCRT method, which works bottom-up and adds up thread-local WCET estimates according to the RTCS configuration, is easy to implement but afflicted with overly pessimistic estimations. These over estimations increase with rising degree of constrained component interaction as this fine-grained knowledge cannot be considered in the composition. The root problem of the compositional WCRT method is the segregated consideration of threads in the preceding WCET analysis (see RQ2), where thread boundaries also act as boundaries for the exploitation of flow-sensitive constraints. In a nutshell, the classical WCET analysis reaches its limits at the application–RTOS boundary; SysWCET will tear down this boundary.

4.3 Related Work

Before I describe SysWCET as an integrated method to tackle the problem of overly pessimistic WCRT estimates, I want to give an overview about regular WCET and WCRT analysis techniques, as well as methods that already consider more fine-grained information about the system.

The goal of WCET analysis is to estimate the upper bound for the longest execution time among all paths through a given program. For all downstream RT analyses to be sound, this bound must be larger than actually occurring execution time. In the literature, many techniques, which are based on measurements on the actual hardware or which rely on static code analysis, were proposed. In this section, I will concentrate on the three important classes of static-analysis methods for WCET estimation: the structure-based methods, the path-based methods, and the *implicit path-enumeration technique (IPET)*. For a more exhaustive discussion of WCET techniques, which includes processor-behavior, control-flow, and value analysis, please refer to Wilhelm et al. [Wil+08].

Structure-based WCET techniques, like the timing schema [Sha89], work on the hierarchic structure of the program's *abstract syntax tree (AST)*. In a preceding machine-analysis step, every leaf node, which are statements that contain no statements themselves, is assigned a basic timing cost. In a bottom-up traversal, the costs are propagated to the root node according to combination rules for this statement type. One prominent tool that is based on the program structure is HEPTANE [CP00; CB02]. Control-flow sensitivity can be partially achieved by using flow contexts, which require virtual AST modifications like loop unrolling and function inlining. However, more complex control-flow constraints, which are also called *flow facts*, like mutual exclusive paths, cannot be expressed. In their methodology, this class is similar to the compositional approach to WCRT analysis as the execution-time costs are propagated from the bottom up.

The second class of bound-calculation methods are the path-based ones, which explicitly enumerate a large amount of paths through the program. Thereby, a very precise hardware modeling can be done as the exact instruction sequence for each path is known. Therefore, path-based methods work especially well for straight-line (=branch-less) code [SA00]. For more complex control-flow graphs, the explicit enumeration of all paths provokes an exponential blow up and becomes infeasible. However, Stappert, Ermedahl, and Engblom [SEE01] proposes a hybrid approach that partitions the control flow into smaller independent scopes, which are analyzed in a path-based manner. Besides being beneficial for the hardware analysis, path-based methods are inherently able to incorporate complex control-flow constraints as they can check the constraints for every processed path.

The third class of bound calculation are *implicit* as they do not explicitly enumerate all paths but encode the longest-path search on top of another problem formulation. For the IPET [PS97; LM95], which is *the* prominent implicit method, we encode the CFG as an *integer linear programming (ILP)* optimization problem. While the IPET is also able to incorporate complex path constraints and flow facts, its implicitness allows it to handle larger systems than the path-based methods. As IPET is the base for SysWCET, I will the IPET in greater detail in Section 4.4.

While WCET analysis is already complex for regular programs, its application to operating-system code is even more challenging as the OS often contains irregular control flows (e.g., interrupts, context switches) and a mix of a high-level language and assembler [Lv+09b]. For example, the assembler problem struck Colin and Puaut [CP01] in the analysis of the RTEMS operating system with the structure-based HEPTANE tool. Later work mostly relied on the IPET and, for example, Lv et al. [Lv+09a] presented an IPET-based WCET analysis of the μC/OS-II RTOS and gave an upper bound for each synchronously invoked system call.

An important issue for the analysis of RTOS code, is the longest interrupt-blockade time as it determines the interrupt-detection latency of the whole operating system. Along these lines, Carlsson et al. [Car+02] and Sandell et al. [San+04] analyzed the system calls of the commercial OSE operating system and gave an upper bound for the disable-interrupt regions.

Similar, Blackham et al. [Bla+11] extended the guarantees of the functionally verified seL4 [Kle+09] microkernel to the timing domain and provided bounds for the longest uninterruptible kernel path. While the original seL4 kernel disabled interrupts for its whole run time to ease the verification process, Blackham introduced explicit preemption points with a bounded interrupt-detection latency in between. Later on, Blackham, Liffiton, and Heiser [BLH14] improved on the seL4 analysis by systematically removing infeasible paths by additional IPET constraints.

As already mentioned, the compositional WCRT is executed after the WCET bound estimation and uses the resulting WCETs. The first WCRT analysis of a fixed-priority system was presented by Harter Jr [Har87] and already entailed the compositional nature of later analyses. The cumulative analysis, like I presented it in Section 4.2, was developed in parallel in [JP86] and [Aud+91; Aud+93]. Later, these WCRT analyses were extended for other task models, like tasks with offset [Aud93], with release jitter [TBW94], or preemption thresholds [KBL10].

Also, different OS services that block the progress of a task due to synchronization and event signaling were considered. For these delays, we calculate and add a *blocking term* to the WCRT formula. This can be done for OS-mediated synchronization protocols, like the priority-inheritance protocol [SRL90a] or the SRP [Bak91], and for inter-core synchronization primitives, like spinlocks [WB13]. For self-suspending, or blocking, tasks it was recently discovered [Nel18] that many previous methods for calculating a maximal waiting time were flawed and had to be fixed.

Besides all the mentioned extensions, most WCRT analyses remained compositional in nature. A different direction was taken by researchers from the model-checking community that formalized the behavior of OSEK systems. Waszniowski and Hanzálek [WH08] considered a system with non-preemptive tasks and ISRs that influence the RTOS state. They used timed automata to represent tasks, which are similar to application-state machines (see Section 3.4.3.2) besides that they are labeled with execution-time demands. These timed automata were combined with a model of the OSEK semantic in the UPPAAL model checker. For each task, they progressively reduced an initial WCRT estimate until the model checker cannot prove the property "maximal model time < WCRT" anymore. As UPPAAL is a model checker that explicitly searches the state space, the WCRT analysis is similar to a path-based WCET analysis on the system level.

From the related work, we see that the WCET analysis moved from structure-based methods over path-based approaches to the IPET. With the increased path sensitivity, we learned to exclude infeasible paths from our analysis, which results in tighter execution-time bounds for individual tasks.

When we take a look at the WCRT methods, we see that they are largely, besides the model-checking approach, compositional. This reflects the problem that we had with structure-based WCET analysis: we could not easily include complex flow constraints and ended up with more pessimistic WCET estimates. Tracing the development of WCET analyses, I propose SysWCET: an implicit and control-flow–sensitive take on WCRT analysis that relies on the IPET and the SSTG interaction model.

4.4 Implicit Path Enumeration Technique

SysWCET uses the IPET to encode the WCRT problem as an ILP. Therefore, I will give an overview about this basic WCET-analysis technique, which I also apply in Chapter 6 to calculate an upper bound for the call-frame allocations on a shared stack.

The IPET was first proposed by Li and Malik [LM95] and later extended to more complex control flows by Puschner and Schedl [PS97]. It tackles the problem of WCET analysis from the same direction as path-based methods as it searches for the execution time of the longest path on the CFG through the program. Thereby, each basic block and possibly every edge, have assigned execution costs, which we accumulate to get the WCET. Unlike the path-based methods, the IPET captures the worst-case paths only *implicitly*. This means that the IPET does not calculate the longest path, but it only calculates the image, in terms of execution counts for each block and each edge, of the longest path.

Unlike the shortest-path problem for a general directed weighted graph (with non-negative edge weights), which is solvable in polynomial time [Dij59], the longest-path problem is, unfortunately, an NP-hard problem [UU04]. However, the problem is only NP-hard for graphs with loops, as the longest-path problem for DAGs is reducible onto the shortest-path problem for DAGs, which can be solved in linear time [KW59]. Unluckily, most programs contain loops and, therefore, finding the longest execution path through a program is, in general, NP hard. Therefore, the IPET stays in the same complexity class if it reduces the longest-path problem onto another NP-hard problem for which efficient solvers exists: *integer linear programming*.

integer linear programming

Integer linear programming is a mathematical optimization technique where we have to choose a vector of integers $x \in \mathbb{Z}^n$. Together with the integer-typed weight vector c, we have to maximize the value of the weighted sum:

$$\text{maximize} \quad \sum_{i=0}^{|x|} c_i \cdot x_i$$

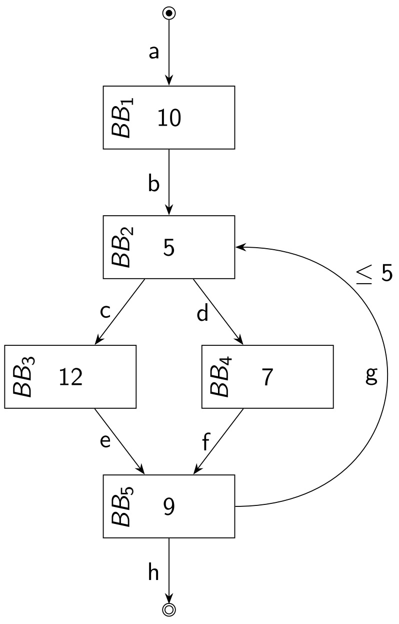
Furthermore, for a solution to be valid, it must also fulfill a set of linear constraints. Each constraint is a less-equal inequality with a constant integer b on one side and another weighted sum (weights: a) on the other side:

$$\text{subject to} \quad \sum_{i=0}^{|x|} a_i \cdot x_i \leq b$$

While this canonical form is restricted to simple less-equal inequalities, standard transformations allow us to express more complex constraints, such as equalities. With these constraints, we can even encode and embed Boolean formulas over x into our ILP that must be fulfilled for the solution to be valid (reduction of SAT onto ILP). It is this flexibility of ILPs that enables the IPET to encode the longest-path search on CFGs alongside with complex flow facts in the same ILP-problem formulation.

execution counts

The key idea of the IPET is to create a *count variable*, which the solver chooses, for each basic block and for every edge. These variables count how often the respective element is visited during the longest path and the WCET becomes the weighted sum of the count variables and the element execution times, which have to be supplied by a preceding machine analysis. Furthermore, we encode the CFG structure as a set of *structural constraints* that ensure that each basic block is left over an outgoing edge as often as it activated by an incoming edge.



(a) CFG with execution times

Structural Constraints:

$$\begin{aligned} a &= BB_1 = b \\ b + g &= BB_2 = c + d \\ c &= BB_3 = e \\ d &= BB_4 = f \\ e + f &= BB_5 = g + h \end{aligned}$$

Entry and Loop Constraints:

$$\begin{aligned} a &= 1 = h \\ g &\leq 5 \cdot b \end{aligned}$$

Objective Function:

$$\text{maximize} \quad 10 \cdot BB_1 + 5 \cdot BB_2 + 12 \cdot BB_3 + 7 \cdot BB_4 + 9 \cdot BB_5$$

(b) ILP

Figure 4.2 – IPET Example. With the IPET, we can encode the structure of the CFG as control-flow constraints, as well as loop-bound and entry constraints. The objective function becomes the WCET for its maximal valid assignment.

In Figure 4.2, we see an example CFG and its IPET-constructed ILP representation, with one variable per basic block (BB_1) and per edge (b). For the control-flow constraints, we sum up the variables of the incoming edges (e.g., $b+g$) and assert it as equal to the basic-block count (e.g., BB_2) and the sum of the outgoing edges (e.g., $c+d$). Additionally, we insert an artificial entry (a) and exit (h) edge, which are visited exactly once, for the counts to become balanced.

flow facts

On this basic control-flow structure, we can add additional control-flow constraints. For example, for every loop, we have to constraint the number of loop executions (6 iterations, $N=5$); otherwise the ILP would be unbounded. For this, we add a constraint that the sum of all back edges (g) is at most N times larger than the sum of all loop-entering edges (b). With these constraints in place, our example ILP has a maximal objective value of 166. For the WCET, the entry node (BB_1) is visited once, the loop header and footer (BB_2, BB_5) are visited six times, and, in all six loop iterations, the left branch is taken (BB_3) as it executes for longer than the right branch. As we see, the ILP solver does *not* give us the longest path but only the execution counts *on* the longest path.

On top of this basic ILP formulation, we can add further path refinements and add more complex flow constraints to tighten the WCET estimate and bring it closer to the actual WCET. For example, if we know that the loop takes the branches within the loop in an alternating fashion, we use a set of constraints to encode this knowledge: (1st row) the helper variable X becomes 1 iff the branch is visited an uneven amount of times; the variable E captures the number of even branch decisions. (2nd row) The uneven "excess" branch (X) is distributed between both branches with the help of two additional helper variables (X_c,X_d):

$$\begin{aligned} E\cdot 2 &= BB_2 - X & E, X_c, X_d &\in [0,\infty[& X &\in [0,1] \\ X &= X_c + X_d & c &= E + X_c & d &= E + X_d \end{aligned}$$

With these additional constraints, the solver can choose the left branch only three times and must take the less-costly right branch also three times. Therefore, this additional flow fact tightens our WCET estimate to 151. In general, we must transform such path constraint to a constraint over the execution counts to use it with the IPET. However, not every path constraint can be transformed that easily as execution counts have no information about execution order; there is no "before" or "after". For a detailed discussion about more complex flow constraints and their encoding as ILP constraints, please refer to Ermedahl [Erm03].

4.5 Response-Time IPET Construction

With SysWCET, I propose a response-time analysis that combines the IPET with the SSTG interaction model. Into the same ILP formulation, SysWCET integrates the lower-level WCET analysis, the whole application code, *and* the control transfer between threads. By this tight integration, we can express inter-thread flow facts and benefit from control-flow–sensitive information, which is normally only available on the WCET level, in the WCRT analysis. In this section, I will first give a short overview about the layered SysWCET IPET construction, before I describe each layer in more detail.

$$\text{WCRT}(\text{RTCS}, \tau_i) = \text{WCET}\left(\text{RTCS}\left(thread_{\tau_i}\right)\right)$$

The key observation of SysWCET is that the WCRT of a task is the WCET of the whole RTCS while it executes and completes the task-implementing thread. From a code-centered view on the RTCS, the WCRT analysis of a task must answer the question: how long does it take, at maximum, from the first instruction of the triggered ISR until the last instruction of the activated thread. My chosen

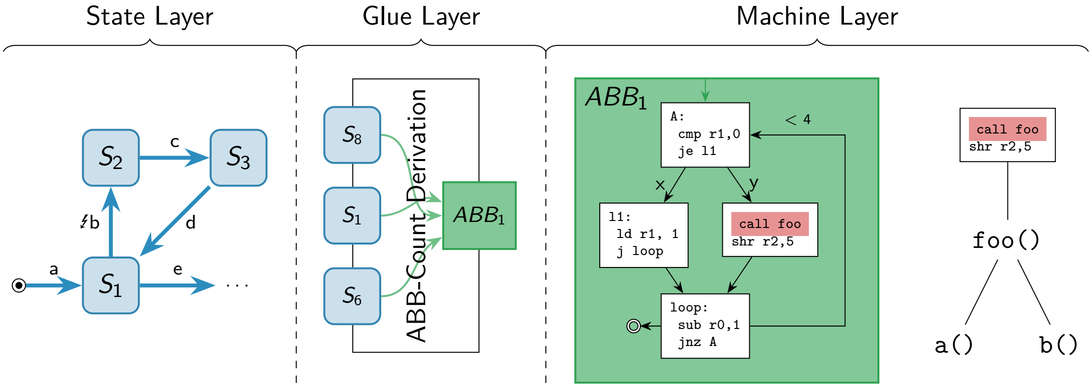


Figure 4.3 – SysWCET IPET Layers

possibility to bound this duration is to find the longest execution path/instruction sequence on the system level. Therefore, SysWCET is similar to (implicit and explicit) path-based WCET analyses methods, but it operates on the whole-system level.

The SSTG includes this longest system-level execution path, since it includes *all* possible execution paths through the system. Therefore, our WCRT problem is equivalent to find the longest path (≅ the longest execution time) in a SSTG subgraph that starts with the ISR entry block and ends with the thread's `TerminateTask()` system call. For the ILP formulation, SysWCET uses a layered construction (see Figure 4.3): (1) in the *state layer*, we interpret the SSTG subgraph as a control-flow graph, encode it with the IPET as an ILP, and expose execution-count variables that capture how often the system visits that state. (2) in the *glue layer*, we derive an ABB execution count from the state counts, since the system can enter and leave a state several times before the linked ABB is completed once. (3) in the *machine layer*, we encode all ABB contents and the referenced non-system-relevant functions with the IPET as ILP fragments and connect them to the ABB counts. On the resulting integrated ILP, we add additional system- and thread-level flow facts.

4.5.1 The SSTG Subgraph of the WCRT Problem

For the normal WCRT analysis, we are confronted with a RTCS and a task, and have to answer how long it takes, at maximum, between job release and job completion. However, as I discussed in Section 2.2, the actual existing RTCS does not execute tasks but threads and ISRs that implement, in their entirety, the task set. Therefore, we have to tie the release and completion of jobs to code points in the system execution. For my implementation, I chose source-code annotations in the form of pseudo function calls, which the developer must supply and which indicate the start and the end of the considered execution paths.

```
TASK(Low) {
  if (...) {
  →timing_start();
    ActivateTask(High)
  } else { ... }
}
```

```
TASK(High) {
  ...
→timing_end();
  TerminateTask();
}
```

Figure 4.4 – Example System with Response-Time Annotations

In Figure 4.4, we see our running example with source-code annotations, where a high-priority thread is activated from a low-priority thread. The user is interested in the WCRT of thread High and puts the start annotation (`timing_start()`) right before the activating system call while the end annotation (`timing_end()`) is located right after the last thread statement. There are two benefits of using function calls for the annotation: (1) we can eradicate these function calls during the system generation to avoid influence on the final system. (2) we can use these function calls for measurement-based response time analysis of the task of interest. During the system-analysis step, we identify the static source-code locations of the annotations and map them to their corresponding ABB such that we end up with two sets of ABBs ABB_{start} and ABB_{end}.

The decision for using sets of ABBs instead of determining a single start/end ABB is a conscious one. As we have discussed earlier Section 2.2.1, the mapping from the RT domain to the OS domain is not unambiguous and, therefore, we can end up with several possible release points for a task implementation. Similar, the mapping can duplicate the task code at two points in the system such that we end up with multiple completion points. As a side effect of this required flexibility, we gain the possibility to determine upper execution-time bounds between arbitrary locations in the system, which do not have to be aligned with task implementations. For example, we can enclose a task implementation like in Figure 4.4 but put an additional completion point in an ISR that cancels the task execution.

From ABB_{start} and ABB_{end}, we will derive corresponding sets of states (S_{start}, S_{end}) from the SSTG state set S and identify the subgraph (S_{sub}) between these sets.

$$\begin{aligned} S_{start} &= \{s \mid s \in S,\ s.\text{next\_ABB}() \in ABB_{start}\} \\ S_{end} &= \{s \mid s \in S,\ s.\text{next\_ABB}() \in ABB_{end}\} \\ S_{sub} &= \{s \mid s_s \xrightarrow{T*} s \xrightarrow{T*} s_e,\ states(T) \cap S_{end} = \emptyset \\ &\qquad s \in S,\ s_s \in S_{start},\ s_e \in S_{end},\} \end{aligned}$$

For the start and end set, we select those states that currently execute one of the ABB blocks from the corresponding ABB set. It is noteworthy that the `next_ABB` field is unambiguous for the whole family of interaction models that I described in Chapter 3. Therefore, SSTG and standard GCFG, but also flow graphs with an intermediate precision (due to a different state-equality operator), can be used as an input graph for SysWCET. However, for this thesis, I will concentrate on the full SSTG as the input graph for the SysWCET analysis.

We select the subgraph S_{sub} between both sets with a depth-first search beginning from the start set. We include all states (and transitions) on the paths between a start and an end state that do not visit an end state in between. While the annotation-based subgraph selection is very accessible for the developer, we also can use other filters and constraints to generate SSTG subgraphs to reflect other WCRT problems. For example, it is often useful to exclude the execution of the start block from the problem but only include states that are direct followup states.

As result of the preprocessing step, we end up with a SSTG subgraph that reflects out WCRT problem. It has potentially multiple entry states, multiple exit states, and it can consist of several disconnected components.

4.5.2 The State Layer: Constraining State Counts

The first layer of the SysWCET IPET construction is the *state layer*. It introduces one count variable for each state and each transition in the SSTG subgraph, which we will denote with S for the following IPET construction. As nomenclature, I will use $||\langle\text{object}\rangle||$ for count variables, which are positive integer-typed ILP variables, of arbitrary objects. Later, we will encounter ILP variables that serve different purposes and have other value ranges.

$$\forall_{s \in S} \left(\sum_{t \in \{* \to s\}} ||t|| = ||s|| = \sum_{t \in \{s \to *\}} ||t|| \right)$$

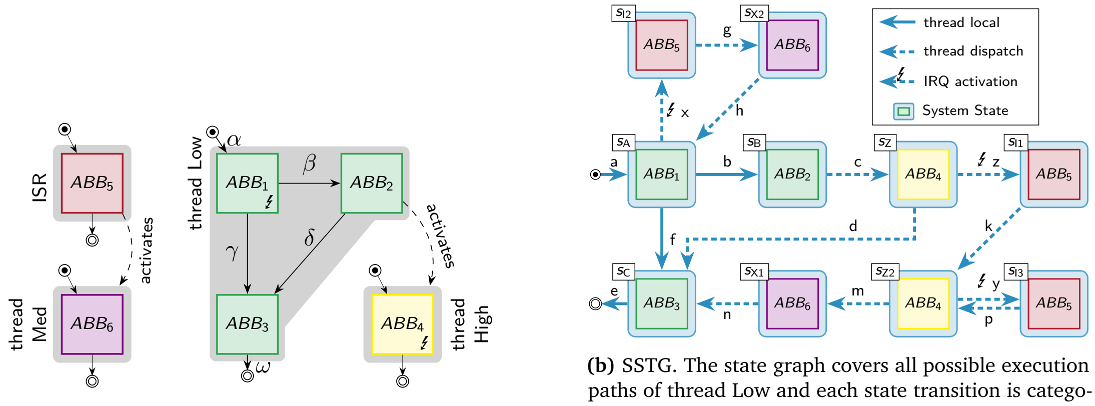
Analog to the WCET variant, we introduce structural constraints on the state layer to encode the control-flow structure of the SSTG. The sum of the count of the incoming transitions, the count of the state itself, and the sum of all outgoing transitions must be equal. However, we have to pay special attention at the start/end states, since we have multiple entries and exits. Therefore, we insert artificial dangling start ($\to s_s$) and end ($s_e \to$) transitions for the count flow to become balanced. Furthermore, we have to assert that exactly one start transition and exactly one end transition can be taken.

$$\sum_{s_s \in S_{start}} || \to s_s || = 1 \qquad \sum_{s_e \in S_{end}} || s_e \to || = 1$$

To illustrate the SysWCET construction, I will extend the example from Figure 3.7 with an ISR that activates the medium-priority thread and that can only be triggered at two locations in the system. Figure 4.5a shows, in a condensed form that is similar to ASMs, the extended example system. For the WCRT analysis of thread Low, we will use the SSTG subgraph in Figure 4.5b as input and encode it as an ILP. Furthermore, we categorize the state transitions into three classes: (1) thread-local transitions (T_{local}) invoke no thread switch and follow the thread-local CFG. (2) thread-dispatch edges ($T_{dispatch}$) represent thread switches that are synchronously invoked (e.g., with a system call). (3) IRQ-activation edges (T_{irq})are the transitions that are the forced activation of an ISR by the processor's interrupt circuitry.

Coming back to the IPET construction, we would add the following structural constraints for the state s_A, which executes ABB_1, to our ILP:

$$||a|| + ||h|| = ||s_A|| = ||x|| + ||b|| + ||f|| \qquad ||a|| = 1$$



(a) ABB Graphs. The ISR activates thread Med and can itself be triggered in two locations.

(b) SSTG. The state graph covers all possible execution paths of thread Low and each state transition is categorized into classes. Each state is labeled (e.g., s_A) and refers its ABB (e.g., ABB_1)from its `next_ABB` field.

Figure 4.5 – Extended Example System and its SSTG. The extended example system has an additional ISR that activates thread Med. Adapted from [▷Die+17]

While it looks superficial to introduce $||a||$ as variable that is constrained to be constant, I chose this explicit and verbose formulation to improve readability. For the ILP-solving time, this redundancy is no impediment as solvers normally preprocesses the problem [CPL16, cha. 13], before any attempt is made to find an optimal solution.

The SSTG (Figure 4.5b) exhibits another issue that we have to tackle on the state layer that is not present in this form for WCET IPETs: *state loops*. As RTCS run potentially forever, the system will revisit previously visited states on a regular basis. For example, the edges (x→g→h) and (y→p) are loops in the SSTG, which are provoked by interrupt activations. Unlike loops in regular WCET problems, an explicit loop bound can often not be given for a state loop. For example, the upper bound for an IRQ–iret loop must be derived from the WCRT and the minimal interarrival time and coordinated with all other interrupt edges of the same source. However, for the structural constraints to become complete, we use an artificial upper bound M, which must be definitely larger than the actual loop bound. We express these structural constraints by stating that no back edge can be taken if the loop-entry count is zero:

$$\sum_{t \in \text{back edges}} ||t|| \leq \sum_{t \in \text{loop entries}} M \cdot ||t||$$

With these structural constraints, the solver will only produce state-count vectors that are derived from valid paths through the SSTG. However, we still lack proper constraints that limit the number of state-loop iterations. A problem that I will discuss in Section 4.5.5.

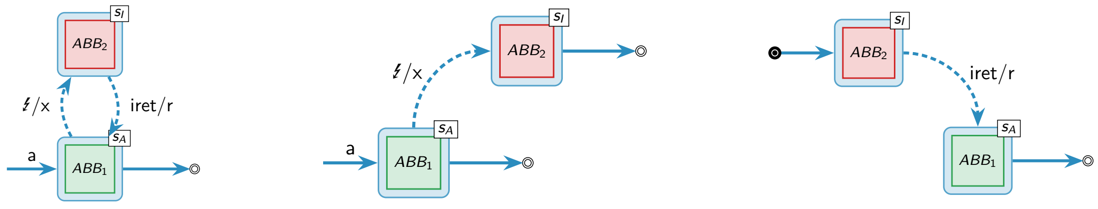
4.5.3 The Glue Layer: Deriving ABB Counts

The state layer provides state-count variables, which indicate how often the system will visit a specific state in the worst-case scenario. However, the ABB that is linked with a state is not necessarily executed as often as the state is visited. For example, if ABB_1 in Figure 4.5b is interrupted three times ($||x|| = 3$) during its execution, we enter the corresponding state 4 times ($||a|| = 1, ||h|| = 3$) as we resume three times to this block after thread Med has completed. However, ABB_1 is executed and completed exactly once and not four times. In general, we want to derive the actual ABB-execution counts from the state counts, instead of using the state counts as an over approximation, to get a tighter WCRT estimate.

The naïve approach to the ABB-count problem would be to subtract the outgoing IRQ-edge counts from the state count to derive the ABB count. For example, for the IRQ–iret loop in Figure 4.6a, we would end up with the following constraints for $||ABB_1||$:

$$||s_A|| = ||a|| + ||r|| \qquad\qquad ||ABB_1|| = ||s_A|| - ||x||$$

While this approach looks promising on the first sight, it is fatally flawed as the WCRT gets underestimated in some cases. It is only sound in situations where the interrupt eventually resumes



(a) Full IRQ-iret Loop **(b)** Reach Completion Point by IRQ **(c)** Enter State only via iret

Figure 4.6 – Problems with naïve ABB-Count Derivation

to the interrupted state. However, as we have to allow arbitrary SSTG subgraphs, we also have to handle interrupts that never resume but reach a completion point beforehand. In Figure 4.6b, we see a distilled version of this problem. If we apply the naïve approach, we end up with the following two constraints:

$$\|ABB_1\| = \|a\| - \|x\| \qquad\qquad \|ABB_2\| = \|x\|$$

If the solver takes the interrupt edge ($\|x\| = 1$), state s_A is correctly visited once but the ABB count for ABB_1 becomes zero, since we subtract one state visit for the interruption. In this case, the solver would only account for one execution of ABB_2 but not for the execution of ABB_1, although the interrupt could occur right after the last instruction of ABB_1. Thereby, the naïve approach leads to underestimation of the WCRT.

Another possible, but flawed, approach to this problem would be to subtract the IRQ-resume edges for the ABB count. Again, for situations with a full IRQ–iret loop, this will lead to a correct derivation of the ABB count from the state count. However, another flawed situation occurs if we encounter a state graph where only the iret but not the IRQ edge is present (see Figure 4.6c). Here, the modified naïve approach would lead to an unfulfillable constraint, as state counts cannot be smaller than zero:

$$\|ABB_1\| = 0 - \|r\|$$

From these corner cases, we see that we are only allowed to subtract full IRQ–iret loops from the state count to tighten up the ABB counts. However, we cannot even detect such full loops statically in all cases from the structure of the SSTG, since combinations of Figure 4.6a+(b), where S_i is also a stop state, can occur. In such a combination, the solver decides dynamically whether the ISR resumes (to s_A) or if the WCRT path ends before (in s_i). For such cases, where we cannot statically guarantee that interrupts and resumptions balance out, we can derive the ABB count without provoking a WCRT underestimation by the combination of static analysis of the SSTG and some additional ILP constraints.

For the generalized ABB-count derivation, we let the ILP solver calculate the number of fully completed IRQ–iret loops and subtract it from the state count. We analyze each ABB_a in isolation and find all states S_a in the subgraph that execute it. For these states, we identify all interrupt ($T_{a,i}$) and resume transitions ($T_{a,r}$) that start, respectively end, in a state from S_a. While the interrupt set is easy to derive from the transition classification, we only include transitions that are reachable in a depth-first search from an interrupt transition:

$$\begin{aligned} S_a &= \{s \mid s \in S,\ s.\text{next\_ABB}() = a\} \\ T_{a,i} &= \{t \mid t \in T_{irq},\ t.\text{from} \in S_a\} \\ T_{a,r} &= \{t \mid t \in T_{dispatch},\ t.\text{to} \in S_a,\ t_i \in T_{a,i},\ t_i \xrightarrow{T*} t\ \} \end{aligned}$$

The sum of all count variables in $T_{a,i}$ is the total count of interruptions in ABB_a; the sum of $T_{a,r}$ is the total count of resumptions. Depending on the scenario (Figure 4.6a-(c)), their delta becomes either zero or indicates “overhanging” interruptions or resumptions in this block. With the help of a *special-order set* (SOS), we can calculate the number of “incomplete” interruptions that never resume. Special-order sets are a feature that most ILP solvers provide to ease the formulation of ILP problems. However, they are just an optimization and can also be implemented with regular ILP constraints. An SOS of class N is a set of ILP variables that constraints that only N variables in the set can be larger than zero; all other variables must be zero. For this ILP construction, we need additional helper ILP variables, which I will denote with H_i.

$$\begin{aligned} \mathrm{SOS1}: H_{a,i} > 0 \quad &\veebar \quad H_{a,r} > 0 \\ -1 \cdot H_{a,i} + H_{a,r} &= \sum_{t_r \in T_{a,r}} \|t_r\| - \sum_{t_i \in T_{a,i}} \|t_i\| \\ H_{ABB_a} &= \sum_{s_a \in S_a} \|s_a\| - \sum_{i \in T_{a,i}} \|t_i\| + H_{a,i} \end{aligned}$$

First, we define an SOS of class 1 such that either $H_{a,i}$ (overhang interruptions) or $H_{a,r}$ (overhang resumptions) can be greater than zero. Afterwards, we calculate the delta between interruptions and resumptions and let, depending on the sign of the result, either $H_{a,i}$ or $H_{a,r}$ absorb the absolute value. For the ABB count, we accumulate the state counts, subtract all interruptions, and re-add all incomplete interruptions. To illustrate this approach, let us have a look at the ABB-count derivation of ABB_4 from Figure 4.5b. It is noteworthy that we could, in this case, statically derive that interruptions and resumptions will balance out and avoid the SOS construction.

$$\begin{aligned} -H_{4,i} + H_{4,r} &= \overbrace{(\|p\| + \|k\|)}^{\text{resumes}} - \overbrace{(\|z\| + \|y\|)}^{\text{IRQs}} \\ \|ABB_4\| &= \underbrace{(\|s_Z\| + \|s_{Z2}\|)}_{\text{state count}} - \underbrace{(\|z\| + \|y\|)}_{\text{IRQs}} + \underbrace{H_{4,i}}_{\text{unresumed IRQs}} \end{aligned}$$

It is important to emphasize that the IRQ–iret detection works on the granularity of ABBs and not states. Therefore, an interruption on edge z and a resume on edge k balance out, although we never resume to the interrupted state s_Z but to the state s_{Z2}, which resumes the interrupted ABB_4 execution.

4.5.4 The Machine Layer: Integrating the Application and Kernel Code

In the glue layer, we have derived ABB counts that indicate how often each ABB is executed in the WCRT scenario. However, we have to derive a worst-case execution time for each ABB, which itself is a single-entry–single-exit region of basic blocks. For this, we formulate the ABB's WCET calculation as an ILP with the IPET, which I already discussed in Section 4.4, and integrate it as a subproblem to the SysWCET ILP.

For the formulation of these WCET-ILP fragments, we start by collecting all ABBs that are referenced from any state in our SSTG subgraph. Furthermore, we also have to account for all non–system-relevant functions that are called from the ABB structure but are not directly covered by their basic blocks. While these non–system-relevant functions do not guide the execution paths of the system, they contribute large parts of the execution time as they cover large parts of the application code and the called libraries.

As both ABBs and functions have a unique entry block, the execution count of the ILP fragment, which is generated by the IPET, can be controlled through one basic-block count variable $\|BB_{\text{a,entry}}\|$ (resp. $\|BB_{\text{f,entry}}\|$). Unlike the simplistic WCET problem (Section 4.4), we do not constraint these entry variables to one, but connect them via constraints to other parts of the ILP. For ABBs, we constraint the ABB count $\|ABB_a\|$ and its respective entry count $\|a, entry\|$ to be equal, which links the machine-layer ILP to the glue-layer ILP.

For functions, we bound their activation count by the sum of all basic-block counts that contain call sites to the function, like it was also already proposed in the initial proposition of the IPET [LM95]. In order to handle function pointers correctly, where the call target is ambiguous, this approach requires helper variables $H_{b \to f}$ for each function that can be called form a basic block b. Thereby, the call-site count $||h||$ is distributed among all possible call targets:

$$||b|| = \sum_{f \in call\_targets(b)} H_{b \to f} \qquad\qquad ||f, entry|| = \sum_{b \in callsites(f)} H_{b \to f}$$

Besides the structural constraints, we also add loop constraints into the integrated ILP, like we would do it for a regular WCET problem. Thereby, we have to pay special attention for loops that cross the ABB boundaries: In the glue layer, we have only derived ABB counts but no ABB–ABB transition counts, which would be required in the loop constraint in terms of loop-entry and loop-back edge counts. We solve this problem, by summing up the corresponding state–state transitions and use them as ABB–ABB transition counts if needed.

kernel code

One side effect of our machine-layer construction is that it already includes the kernel code, as non-computational ABBs invoke system calls. Due to our restrictions on the static CFG structure, we know for every system call exactly which RTOS function is invoked and reference the actual system-call implementation, even if it crosses protection-domain boundaries. For example, ABB_2 in Figure 4.5 is a system-call ABB and it references the `ActivateTask()` execution path through the kernel. Thereby, we account every system-call site exactly with the execution cost for the specific system call instead of adding the maximal kernel execution time. In Section 4.5.5, I will take a closer look at additional constraints that I had to introduce to handle the dOSEK kernel code correctly.

As the last step of our ILP construction, we have to assign timing costs to all elements of the combined ILP problem, like basic blocks and interrupt edges. In general, we assign a constant cost to each count variable in the ILP and formulate the objective function as the weighted sum of count variables and timing costs.

For the machine-code basic blocks, I rely on the normal machine-code analysis that produces a worst-case timing for a given instruction sequence. As I focus on the path analysis part of the WCRT problem, I use the *basic processor model* [Roc14], where each machine instruction is counted as a single cycle without inter-instruction effects or caches. In Section 4.6.4, I will discuss the incorporation of more complex hardware analyses into the SysWCET approach.

IRQ delays

However, at this point, I want to point out one specific hardware-induced delay that is closely related to the structure of the SSTG. On most platforms, interrupt handling induces a (varying) timing penalty if the ISR interrupts the execution at some point within an ABB. These costs can, for example, stem from a forced pipeline flush or communication with the interrupt controller. As an IRQ edge in the SSTG does not specify the exact location of the interruption, I require the machine-code analysis to calculate an ABB-specific worst-case interruption delay for any code location that is covered by the ABB. We account for this interruption delay by adding it as an additional cost to all interrupt transitions that originate from the ABB. For example, if an interruption in our example system Figure 4.5 takes 5/7 cycles for the ABBs ABB_1/ABB_4, we add the term $5 \cdot ||x|| + 7 \cdot ||z|| + 7 \cdot ||y||$ to our objective function.

If our system contains no IRQs, we can already give the ILP problem to a solver, like `CPLEX` [CPL16] or `gurobi` [Gur19], and we will receive a WCRT estimate for our SSTG subgraph. The solver will choose, in compliance with the ILP constraints, those count-variable values that maximize the WCRT. Thereby, the solver will find the image of the costliest state–state path through the SSTG, calculate maximal ABB counts, and find the longest execution path through each ABB and all referenced functions.

However, despite our initial goal of fully integrating WCET and WCRT analysis, I want to point out that the SysWCET problem formulation is also capable of a more compositional WCRT analysis that still inherits most of the flow-sensitive properties. For this, we analyze the WCET of each ABB in the system individually, skip the machine-layer generation, and assign the ABB execution cost to the ABB-count variables. Thereby, we still have the possibility to formulate inter-thread constraints on the (coarser) granularity of the ABB structure, but lose the ability to tie together machine-code basic blocks from different ABBs by constraints.

4.5.5 System- and Thread-Level Flow-Fact Constraints

While we already added loop bounds as one type of flow facts on the machine layer, we can introduce, now that the ILP structure and all variables are in place, more complex constraints to tighten the WCRT bounds. These ILP constraints can put count variables from all three layers in relation to each other.

mapping problem

One problem for flow-fact handling is the (possible) gap between the ABB level, which is directly derived from the compiler's *immediate representation (IR)* level, and the actual machine-code level. In principle, the compiler back end is allowed to change the IR-level CFG structure in the machine-code generation. For example, if the processor supports predicated instructions, the back end can inline small conditional blocks into their predecessor blocks [Mah+92]. While this is not a problem for the formation of ABBs as function- and system calls are sequence points, it can be an issue for flow facts that are formulated over the IR-level CFG structure. For example, such flow facts arise if we reuse results from the compiler's control- and data-flow analyses to gain insight into the actual execution paths.

In order to safely use these and other ABB-level flow facts, I used *control-flow relation graphs* (CFRG) [HPP13] to bring all flow facts down to the machine-code level. The CFRG is a meta-CFG that relates the relative progress between the IR-level and the machine-level CFG. It is produced by a modified compiler that tracks all CFG modifications and keeps the CFRG up-to-date with all optimizations. The nodes in the CFRG are tuples and *can* contain one basic block from each side. If a node references two blocks, we call it a *progress node*, as it indicates that the progress of the program at certain location is in sync. Along these progress nodes, we are able to push ABB- and IR-level flow facts down to the machine level [HPP13].

interrupt counts

Another problem, which we have postponed in the state layer (Section 4.5.2), is the global limitation of interrupt occurrences with ILP constraints. As we have seen in the initial discussion of the compositional WCRT analysis (Section 4.2), the number periodic and sporadic activations of high-priority tasks depends on, and prolongs, the WCRT of the low-priority task. For the OS domain, we transfer this observation to ISRs, which implement the periodic and sporadic thread activations (Section 2.1.2.1) and derive interrupt counts from the WCRT estimate.

For this, we capture the value of the objective function, which is the weighted sum of count variables and timing costs, in a (time-unit typed) ILP variable T_{WCRT}. For each interrupt source i, we introduce an interrupt-count variable $\|i_{max}\|$ and constraint it as an upper bound with the WCRT and its minimal interarrival time I_i. Furthermore, I take a possible release jitter J_i [Aud+93] of the source into account:

$$I_i \cdot \|i_{max}\| \leq (T_{WCRT} + I_i + J_i)$$

The intuition behind this constraint is that the worst case occurs if: (1) The first interrupt occurs exactly at $t = 0$ and, subsequently, every I_i and interrupt triggers. To express $\lceil \frac{T_{WCRT}}{\|i_{max}\|} \rceil$ as an ILP constraint, we add I_i to the WCRT. For example, for an $I_i = 30$ and an $T_{WCRT} = 100$, we would get $30 \cdot \|i_{max}\| \leq 130$ and we would encounter up to 4 interruptions over our WCRT execution path. (2) We account for the release jitter by assuming that the last interrupt occurs J_i ticks before its regular time. It is important that we use a less-equal constraint here as the maximal number of interrupts does not necessarily lead to the WCRT in all cases; sometimes, an interruption can shorten the execution time, as we will see in the evaluation. In a second constraint, we allow the solver to distribute the interrupt count over all interrupt transitions $T_{irq,i}$ that are associated with the source:

$$\sum_{t \in T_{irq,i}} \|t\| = \|i_{max}\|$$

With these constraints, the solver is allowed to distribute $\|i_{max}\|$ over all interrupt transitions. Thereby, the solver has no restriction about clustering of IRQs and all interrupts can trigger in the same ABB. Actually, the solver will prefer such clusters if one interrupt–iret loop is more expensive than the others. While such a clustering does not obey the minimal interarrival time in a control-flow sensitive manner, it is still sound as the resulting WCRT is a safe over approximation of the actual worst-case path. The clustering problem is rooted in the basic nature of the IPET, which does not work with explicit paths but only with implicit execution counts. Therefore, I have not addressed this problem any further for this thesis.

implementation idiosyncrasies

During the application of SysWCET to *d*OSEK-generated systems, I encountered a mismatch between the ABB/SSTG level and the actual implementation level.

The concrete problem arises in the ISR activation and it is representative for mismatches that still exist despite the interaction model's code-centric focus: As my model is implementation agnostic and only reflects the standardized semantic but not the mechanisms of the kernel, the ISR dispatch is not covered (see Figure 4.7). In the model, the first block of the ISR starts immediately after the IRQ edge is taken. However, for many architectures, an RTOS must distinguish between different IRQs in a central ISR-dispatching method. For example, the RISC-V *instruction-set architecture (ISA)* [Wat+14] maps all external events to a single interruption vector and passes the identifier of the event as data.

If we would consider only `isr_uart()` in Figure 4.7, we would miss the execution time of the `isr_entry()` and the `dispatch()` function, which could lead to a fatal under-estimation of the WCRT. Therefore, we have to include the top-level IRQ handler and all its child functions, including our `isr_uart()`, in the ILP and use the ISR-level ABBs only to guide the execution flow to the actual handler.

For this, we use the combined state count of all IRQ transitions (T_{irq}) to constraint the execution count of the top-level handler and the ABB count guides the execution flow to the actual ISR:

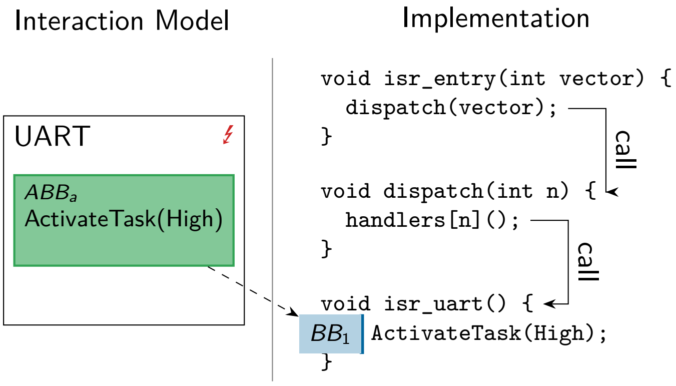
$$\|f_{\text{isr\_entry}}, entry\| = \sum_{t \in T_{irq}} \|t\| \qquad\qquad \|ABB_a\| = \|BB_1\|$$

These constraints, ensure that indirect function call in `dispatch()` will take the function-call edge ($H_{\text{dispatch}\rightarrow\text{isr\_uart}}$) exactly $\|ABB_a\|$ times. With such constraints, we can also handle other implementation details, like the co-location of multiple alarms within one timer ISR.

4.6 Comparison to Compositional WCRT Analysis

For the experimental validation, I integrated the SysWCET approach into *d*OSEK and the `platin` [Hep+15] WCET framework and applied it to real-time systems of different sizes. These systems were compiled for the PATMOS [Sch+15] processor architecture and I compare the compositional WCRT to a WCRT estimation that SysWCET calculated. For validation purposes, I also measure the actual execution times, which must be strict smaller than the calculated bounds, in an instruction-set emulator. With this evaluation, I do not only want to show that SysWCET in particular can provide tighter WCRT bounds, but I also want to demonstrate that the non-compositional view on WCRT analysis is possible (RQ1) and beneficial (RQ2).

Besides the WCRT results, I also provide results for the size of the SSTG subgraphs, the ILP complexity, and the time that is required to solve the optimization problem with a state-of-the-art ILP solver. Furthermore, I will discuss the implications of the results and discuss the generalizability, benefits, and limitations of the SysWCET approach.



```
void isr_entry(int vector) {
  dispatch(vector);
}

void dispatch(int n) {
  handlers[n]();
}

void isr_uart() {
  ActivateTask(High);
}
```

Figure 4.7 – IRQ Handling in Operation Systems. On the level of the interaction model, the ABBs of the ISR simply start to execute if an interrupt occurs. However, on the OS level there is often a dispatching logic in place that precedes the execution of the ISR.

4.6.1 Evaluation Scenario

The SysWCET evaluation is based on the PATMOS [Sch+15] architecture, which was developed in the T-CREST research project as the processor component of a timing predictable computing platform. With PATMOS, the researchers aimed for a better analyzability with regard to (worst-case) timing analysis while maintaining an acceptable average-case performance. However, as SysWCET is only a worst-case path analysis and does not change the hardware analysis, I used the PATMOS instruction-set architecture as a basic processor model [Roc14], where each instruction takes exactly one time unit. Thereby, I consciously neglected the influence of hardware components, like caches or the pipeline, and the machine-code analysis became trivial.

As benchmark scenarios, I use different OSEK applications with a wide range of OS-usage patterns and of different sizes. I analyze these applications with *d*OSEK, calculate the full SSTG, and extract SSTG subgraphs for an individual task, which can be implemented by one or multiple threads. For each application, *d*OSEK generates an OSEK kernel, which is tailored on the instance level (see Section 1.2, [▷Fie+18]) for the given application (i.e., threads are statically allocated and initialized). Furthermore, *d*OSEK exports the SSTG-subgraph structure and other analysis results, like the ABB–basic-block mapping and additional kernel-level flow facts, in `platin`'s PML format [Pus+13]. The generated kernel source and the application were compiled with the PATMOS toolchain, which outputs not only a system binary, but also produces additional PML files with information about the IR-level basic blocks, machine-code basic blocks, CFRGs, and compiler-deduced flow facts.

The deduction of WCET and SysWCET-WCRT bounds is done with `platin` [Hep+15; Pus+13], a WCET analysis tool that was developed in the T-CREST project and proved to be quite accessible for my adaptations, as it is written in the `ruby` scripting language. `platin` combines the various PML fragments and uses the IPET to formulate ILP problems for the WCET analysis of individual functions. For the optimization of the ILP, `platin` provides different solver backends, from which I used the backend for `gurobi` [Gur19] 6.5.2. For the analysis run-time measurements, I ran *d*OSEK, `platin`, and `gurobi` on a 16-core Intel E5-2690 machine with a processor speed of 2.90 GHz.

For the compositional WCRT analysis, I used `platin` to calculate the WCET of individual functions and the thread-entry functions in isolation. For the WCRT analysis, I use the methodology of Audsley et al. [Aud+93] and incorporate the following information, besides the WCET times, into the composition: static thread priority, minimal inter-arrival times, and periods.

In order to validate the deduced upper bounds, I execute the system binaries with the PATMOS instruction-set simulator `pasim` and count the number of executed instructions according to the basic processor model. As observed worst-case execution time, I give the longest observed execution time between the `timing_start()` and `timing_end()` (see Section 4.5.1). However, as it is unlikely that the executing system encounters the actual worst-case event sequence for larger benchmarks, these observed times can only be considered under-approximations of the actual WCRT.

4.6.2 Micro Benchmarks

First, I will apply SysWCET to some small scale benchmarks where I can discuss the application structure and the savings on the WCRT bound in detail. While these benchmarks are, for themselves, unrealistically small for a real-world application, the included patterns of interaction can often be found in larger applications. As SysWCET is context-flow sensitive and covers the RTOS semantic, it is able to exploit such patterns regardless of the application size.

Listing 4.1 Triple Modular Redundant Application. The low-priority thread activates a high-priority thread three times from a bounded loop.

```
TASK(HI_workload) { ... }

TASK(LO_control) {
→timing_start();
  for (i=0; i < 3; i++) {
    ActivateTask(HI_workload);
  }
  // Not shown: Vote over results
→timing_end();
}
```

The first micro benchmark (Listing 4.1) is a *triple modular redundancy (TMR)* application, which can be found in safety-critical systems that employ software-based measures against transient hardware faults [▷Hof+14]. A workload, which is located in HI_workload, is critical for the correct operation of the system but the processor is susceptible to encounter bit flips and other soft errors in its execution. Therefore, the application executes the critical workload three times (LO_control) and performs a majority vote over the results. The relative priorities of both threads results in an interwoven execution of control and workload thread.

For the benchmarks, I replaced the application code by (short) busy-loops where the thread would perform its computation phases. This simplification of the benchmark code allows me to focus on the interaction between the application and the real-time operating system. It is sound in the sense that whole systems has the same (potential) interaction patterns as the replaced code blocks are located in computation ABBs or in non-system relevant functions.

The removal has two more consequences for the evaluation: (1) The ILP problem becomes smaller as the ABBs have no real content, which will reduce the ILP solving time. (2) The reported WCRT savings will be lower than what can be achieved for a full application: For example, if SysWCET avoids a thread activation due to an infeasible path, the WCRT shrinks only by the kernel overhead for the activation and the resumption, and not the WCET of that thread. Since (2) gives the benchmark designer the possibility to increase the savings arbitrarily (by prolonging the excluded thread execution time), I decided to remove the application code to avoid this evaluation pitfall.

The TMR benchmark is challenging for a compositional WCRT as there is a mapping mismatch between the RT domain and the OS domain: As the TMR application performs one chore redundantly

complex control dependencies

	SSTG		ILP Problem			*Worst-Case Response Time [instrs.]*		
	ABBs	States	Vars	Constr.	Run Time	SysWCET	Compositonal	Observed
TMR	9	9	248	458	94 ms	3 319	3 319	2 893
Alarm	5	13	319	594	0.23 s	765 716	766 733	764 785
Aborted Comp.	9	35	601	1 088	0.3 s	56 340	60 425	55 738
Activate w/o Disp.	3	3	146	264	96 ms	318	318	123
Activate w/ Disp.	6	6	271	487	105 ms	596	596	395
Terminate w/ Disp.	6	6	266	478	131 ms	593	601	398
Wait & Wakeup.	6	6	232	416	124 ms	607	610	436
Interrupt w/ Sched.	3	12	155	290	99 ms	464	466	339

Table 4.1 – Results for the Micro Benchmarks.

and it can be considered as one individual task that contains a control part and a workload part. However, the implementation on the OS domain chose to execute the workload in a separate thread with a higher priority, which is controlled by and fully dependent on the lower-priority control thread. This results in a situation, where the workload thread is activated with the same activation spacing as the control thread but it is executed exactly three times as often. Therefore, a general compositional WCRT analysis of this pattern would require a more complex WCRT formula, especially if other threads are involved as well.[8]

However, for this micro benchmark, we can simulate the scenario and all three preemptions manually and end up with a compositional WCRT for the two TMR threads of 3 319 cycles. This upper bound is 15 percent larger than the largest observed execution time. The overestimation stems from kernel paths that the IPET considers but that are not taken in the real execution. A result overview about this, and all other, micro benchmarks can be found in Table 4.1.

For the TMR scenario, SysWCET constructs an ILP problem from a small SSTG subgraph with only 9 states where each state references a different ABB. This SSTG contains a state loop between both threads. We constraint the loop with an upper repetition bound of three, which was automatically deduced by the compiler, given to `platin`, lowered with CFRGs to the machine level, and added to the ILP. However, as these loop constraints spans multiple ABBs, they also constraint the state layer and, therefore, the activation count of the workload thread becomes three. SysWCET is able to handle this pattern automatically since CFG-level flow facts from the machine-layer directly constraint the state layer. In the end, we get the same upper bound for the WCRT but in a fully automated manner and without a manual extraction of the activation count for the workload thread.

The second micro benchmark (Listing 4.2) demonstrates the benefits of having an integrated view on the kernel code and the real-time configuration. This benchmark contains two tasks implemented in two threads and one timer ISR. The low-priority task is sporadic and implemented in thread LO_computation, which computes for over 750 000 instructions. For the periodic task, the kernel triggers a timer ISR with a period of 100 000 instructions, which activates on every third interruption the high-priority thread. The kernel activates the high-priority thread *after* the completion of the interrupt.

The challenging part of this benchmark is hidden in the kernel paths of the timer ISR. For the compositional approach, the WCET of the timer ISR is calculated once and, thereby, the activation of thread HI_urgent extends the WCRT bound for every interruption although it is only triggered on every third IRQ. Although it would be possible to calculate different WCET bounds (i.e., with and

---

[8] For example, if the TMR task with its two threads is executed together with a medium-priority, the WCRT of the medium priority task is surprising around $1 \cdot WCET_{HI} + 1 \cdot WCET_{ME}$ as the medium-priority thread blocks the activation of the two other TMR workload executions.

Listing 4.2 Computation Interrupted by Periodic Alarm

```
TASK(LO_computation)() {
→timing_start();
  /* 760 021 instrs.*/
→timing_end();
}
TASK(HI_urgent)() { ... }

ISR(timer) {
  counter++;
  if (counter % 3 == 0) {
    ActivateTask(HI_urgent);
  }
}
```

without activation) and use it in a specialized WCRT method, this approach becomes infeasible if the number of alarms increases; the number of different bounds would grow exponentially. Even worse, the kernel code that implements the timer ISR does not necessarily look as simple as Listing 4.2 suggests, but probably uses a more dynamic data structures, like linked lists.

Here, SysWCET wins through its integrated nature: the flow graph of the timer-ISR implementation, the maximal number of interruptions, and the number of alarm activations are available in same ILP problem (see Section 4.5.5). With the number of interruptions, we constraint the execution count of the ISR's entry block. With the number of alarm activations, we limit the function-call edges from the ISR to `ActivateTask()`. Thereby, the ILP solver will figure out the correct execution counts between the ISR entry and the activation, regardless of the ISR's control flow. This is similar to the situation described in Figure 4.7.

For the WCRT analysis of the low-priority task, both analyses derive that the timer triggers 8 times. However, SysWCET accounts for only 3 alarm activations and, therefore, derives a 1 017 cycles tighter bound than the compositional approach. If we take into account that the main driver of the WCRT is the computation in the low-priority thread and subtract its constant share of 760 021 cycles, SysWCET derives a 15 percent tighter bound for the remaining parts.

Listing 4.3 Abortable Computation

```
TASK(HI_control) {
→timing_start();
  ActivateTask(thread_LO_computation);
  WaitEvent(sig_done || sig_abort);
→timing_end();
}

TASK(LO_computation) {
   do_computation();  // 55613 instrs.
   SetEvent(HI_control, sig_done);
}

ISR(abort) {
   SetEvent(HI_control, sig_abort);
}
```

The third micro benchmark (Listing 4.3) highlights the benefits of taking the system-call semantic and the induced RTOS–application interaction in the WCRT analysis into account. The benchmark contains one task that performs background computation, which can be aborted by an external event. Our WCRT path starts in the high-priority thread HI_control, which activates the low-priority worker thread and waits passively for the completion or the abortion signal. These events are signaled after the computation completes or in the sporadically occurring abortion ISR, which has a minimum interarrival time of 10 000 cycles.

The challenges of this benchmark are threefold: First, it includes, with HI_control, a self-suspending thread, which are, in general, considered challenging for response-time analyses [Nel18]. Second, the benchmark includes a complex interaction pattern that depends on the concrete semantic of the `WaitEvent()` system call, which waits for at least one signal to arrive. Third, the actual response time decreases if the abortion interrupt occurs too early as it brings the control-flow directly back to the `timing_end()` marker without completing the computation.

The compositional WCRT analysis computes the WCET for each thread and the ISR individually. With the information that both threads belong to the same task, we account for one full execution of each thread in the WCRT. However, for the ISR, the compositional approach must assume, as it uses no control-flow sensitive interaction information, that the interrupt occurs seven times during WCRT time span (minimal IAT=10 000). It is noteworthy that we must add the seventh interruption only because the ISR execution itself prolongs the execution time too much. In total, the compositional approach ends up with a WCRT bound of 60 425 cycles.

On the other hand, the SysWCET approach is able to derive a very tight bound that is close to the maximally observed response time (+ 1.08 % overestimation). From the ILP execution counts, we can even recover and confirm the actual worst-case scenario. For this benchmark the response time becomes maximal if the computation completes without interruption and the ISR triggers exactly before LO_computation invokes the `SetEvent()` system call. The solver selects, rightfully, this path as the combination ISR+`SetEvent(sig_abort)` is more expensive that completing with `SetEvent(sig_done)` alone. So in the worst case, the computation completes, but the sporadic event invalidates it before we know about it.

Besides these application-sized benchmarks, I also want to demonstrate another benefit of the SysWCET approach. Since the chosen SSTG-subgraph-selection strategy uses function-call sites as start and endpoints of the WCRT time span, we can give upper bounds for the execution of different system calls in their context. This includes `ActivateTask()` calls with and without a following dispatch, `TerminateTask()`, `WaitEvent()` with immediate wake up through a `SetEvent()`, and an ISR with following reschedule. The results for these nano benchmarks can be found in the lower half of Table 4.1.

For these micro and nano benchmarks, the complexity of the analysis itself is naturally very low (see Table 4.1) and stays always below 0.5 seconds. However, we can get a first impression about the driving factors of the analysis run time. For this, I show for every benchmark the number of SSTG states and the number of referenced ABBs. While the former gives us an impression of the variability of the *dynamic* execution paths, the latter denotes the *static* complexity of the application. In all cases, there were at most 25 percent more SSTG transitions than states. As all benchmarks include no real computation, the influence of varying application code is eliminated. If we compare the run time for TMR and the aborted computation case, we see that the increased number of SSTG states has a higher impact on the analysis run time than the larger number of application code blocks.

4.6.3 *i4*Copter: A Realistic Task Set of a Safety-Critical Real-Time System

While the micro benchmarks illustrate specific situations where the SysWCET approach outperforms the compositional approach with regard to easier applicability and tighter WCRT bounds, I also apply my approach to a larger RTCS that reflects the structure of a real-world application. For this, I derived

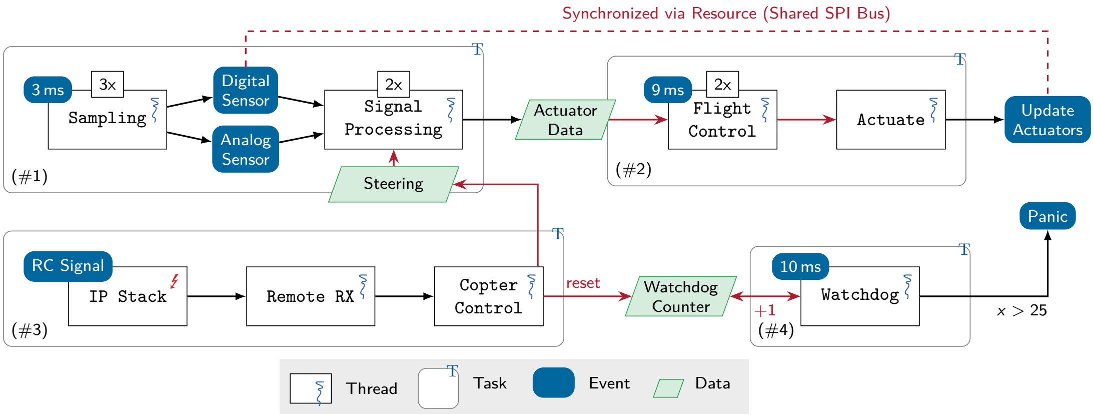


Figure 4.8 – The *i4*Copter. The task set of the *i4*Copter consists 4 tasks that are implemented by 11 OS threads. There are 3 periodic tasks (#1, #2, #4) and one sporadic task (#3), which are implemented on the OS level with one timer ISR and one remote-control ISR. The blue-indicated events are logical events to illustrate the copter functionality, not OSEK events. Derived from [▷Hof+15].

a benchmark from the *i4*Copter [Ulb+11], a safety-critical embedded control system for quad-rotor helicopters, which was developed in cooperation with Siemens Corporate Technology. From the application, I extracted the task set and the thread setup (Figure 4.8) without the computation code in order to focus on the application–RTOS interaction. As this application is also used in later chapters, I will describe its structure in more detail.

From the RT perspective, the *i4*Copter consists of 4 tasks of which 3 tasks (#1, #2, #4) are periodic and one is sporadic (#3). The tasks are prioritized with rate-monotonic scheduling and perform (from the highest priority to lowest) different actions: Task #1 (5 threads) samples the digital and the analog sensors and performs sensor fusion. Task #2 (3 threads) executes the flight controller and updates the actuators. Task #3 (1 ISR, 2 thread) receives and interprets signals from the remote control, and the watchdog task #4 monitors the remote-control channel (1 thread).

On the implementation level, the access to the SPI bus is coordinated with an OSEK resource, since multiple threads require it to communicate with their respective peripheral devices. Furthermore, two threads (Flight Control and Watchdog) are marked as non-preemptable, the periodic alarms are automatically started at boot time and only the watchdog timer is reconfigured at run time. The *d*OSEK RTOS manages these periodic activations with one timer ISR, similar to Listing 4.2. All in all, the *i4*Copter consists of 11 threads, 3 alarms, 1 user-specified ISR, and one OSEK resource.

For the interaction analysis of the *i4*Copter, I use the application's control-flow graphs, the system configuration, and information about the task affiliation of each thread, as well as the assumption that the task's period is equal to its deadline. Together with the task affiliation, this assumption forbids the release of a new job before the last job has finished (see Section 3.4.3.1). On the SSTG level, this constraint reduces the graph size significantly as an IRQ edge is only allowed if all threads from a task have finished.

In Table 4.2, we see the WCRTs analysis of the three top-prioritized tasks (#1, #2, #3), as well as the SSTG characteristics and ILP sizes. For the WCRT bounds, we see that SysWCET consistently provides a tighter bound (—10.5 %,—7.74 %,—7.33 %) than the compositional approach. This saving is rooted in an inter-thread control-flow constraint within task #1: the initially activated sampling thread (3ms) activates (mutually exclusive) one of the two other sampling threads upon a dynamic

	SSTG		ILP Problem			*Worst-Case Response Time [instrs.]*		
	ABBs	States	Vars	Constr.	Run Time	SysWCET	Compositonal	Observed
#1: Signal Gathering	33	9 506	16 269	30 432	14.72 s	5 626	6 286	1 168
#2: Flight Control	55	7 690	16 528	30 666	161.56 s	9 279	10 057	2 261
#3: Remote Control	63	4 608	12 987	26 849	92.57 s	9 768	10 541	790

Table 4.2 – WCRT Results for the *i4*Copter.

decision. While the compositional approach must account for both thread activations and the respective RTOS overheads, SysWCET is able to exclude the thread with the lower execution cost from the worst-case path. Since task #1 has the highest priority in the system, the WCRT savings in this task propagate also to the lower-priority tasks (#2,#3) as their WCRT account for one job release from task #1.

Since the *i4*Copter benchmark is more complex than the micro benchmarks, the actual worst-case event sequence that triggers the WCRT is hard to find: (1) We have to reconstruct the worst-case control-flow path from the execution counts; a problem that is ambiguous in a general control-flow graph. (2) We have to construct an input vector that triggers that control flow. This is related to the path-wise test-data generation problem, which is considered a hard problem in software testing [Kor90; Edv99]. (3) The IPET-determined worst-case path can be infeasible, such that we cannot trigger it in reality [BLH14]. Therefore, my benchmark executions probably did not trigger these worst-case situations, which explains the large difference between calculated and observed response time.

For the SSTG size, we see that the number of states decreases as more tasks and ABBs are covered. While this seems counter-intuitive, it stems from the constraint about the re-occurrence of interrupts: For the WCRT of task #1, the IP-stack ISR (#3) can trigger in every computation block and set the remote-RX thread ready (in the background), which effectively doubles the state graph. For task #3, one of two threads (Remote RX or Copter Control) are ready or running for the whole worst-case path and therefore no interrupt can reoccur; the state space does not fork as often.

For the analysis complexity, we see no clear solving-time trend that can be directly derived from the SSTG or the ILP characteristics. However, the solver finds a solution in reasonable time ($< 180s$) for the three tasks. This is acceptable since if we consider that the WCRT analysis is done ahead of time. Nevertheless, if we consider that these benchmarks excluded the application code, the scalability of could become an issue, which I will discuss in the following section.

4.6.4 Discussion

SysWCET formulates an integrated ILP problem with IPET that combines the RTOS scheduling (state layer) with the application and the machine-code layer. Thereby, we can avoid problems of compositional WCRT analyses but encounter different issues and limitations. In this section, I will shine a light on these limitations and discuss further benefits of the approach for timing analysis. Afterwards, I will describe in Section 4.7 how the SysWCET approach can be applied to worst-case energy consumption.

4.6.4.1 Scalability of SysWCET

In the evaluation, we have seen that the ILP solving takes a considerable amount of time for the *i4*Copter benchmarks. While the *i4*Copter, with its eleven threads, seems like a relative small system, it is not uncommon in some domains, like automotive, to have systems in this size range. For example, the OSEK standard [OSE05] specifies the minimal number of priorities to be at least 16. The popular ERIKA-OS [Evi12], which the standard, provides only 32 priorities on a 32-bit machine and, therefore, can have only 32 threads with distinct priorities (BCC1/ECC1).

The main issue for the SysWCET scalability is the potentially exponential growth of the SSTG with increasing system complexity and size (see Section 3.4.3), as SysWCET searches (implicitly) for a path through the explicitly-enumerated SSTG. Especially, the nondeterminism that is introduced by interrupts is one of the driving factors for the SSTG growth, which directly maps to a growth in ILP variables and constraints. In Section 3.4.3, I already outlined different strategies to cope with this problem by (1) incorporation of more knowledge about the system and (2) use a more condensed and imprecise interaction model.

The incorporation of more knowledge about the system is the preferable variant to cut down on the SSTG size, as it does not influence, or even benefits, the tightness of the WCRT bounds. During the SSTG construction, we can include more information about the environment of the system (e.g., constraints on interrupt occurrences) or the application logic (e.g., avoidance of infeasible paths). On the application-level, these considerations are similar, and we can reuse results from, the infeasible-path analyses that were done for the WCET analysis [BH13; BLH14].

The other possibility to shrink the ILP problem size, without changing the SysWCET method, would be to use a more condensed interaction model, like the GCFG or other merged SSTG variants (see Section 3.4.3.2 and Section 3.5.1). Thereby, the interaction model loses precision as different global paths through the system are considered equivalent although they are triggered by different event sequences. This results in fewer constraints on the execution path and, therefore, in an increased likelihood that the deduced worst-case path is a less tight over-approximation of the actual one.

Another idea, which is a topic of further research, is to replace the SSTG and the state layer with an ILP fragment that grasps the interaction implicitly. For this, we would have to encode the RTOS scheduling rules and the system-call semantics with ILP and bind it to the thread's CFGs. For example, the execution count of an `ActivateTask()` ABB would restrict the execution count of the activated thread's entry block. Here, one problem, which has to be solved for this implicit formulation, is to restrict the system-call effect to the blocks that follow the system call (i.e., the thread dispatch must happen after the `ActivateTask()`). Thereby, we could keep the benefits of SysWCET (i.e., cross-thread flow facts) without the state-explosion problem of the SSTG.

4.6.4.2 Limitations to the Applicability

One problem for the general applicability of SysWCET for the WCRT analysis is its dependence on the SSTG extraction and the, thereby, induced requirements on the system model. As discussed in Section 3.6, my interaction analyses require a static, fixed-priority, real-time computing system with known control-flow graphs. Therefore, SysWCET cannot handle systems that use scheduling strategies with dynamic job-level priorities or systems with dynamic code loading. While the first restriction is an obstacle that a developer can actually encounter, the second one is only hypothetical, as we can never give a strict timing bound for a system that can perform unrestricted code changes at run time.

The other limitation of SysWCET is less obvious and is related to self-suspending tasks that wait on some external event (unlike micro benchmark #3, which waits for the completion of another thread). The problematic pattern arises, if a thread waits for an event variable (`WaitEvent()`), which is signaled via an external sporadic event (i.e., `SetEvent()` in an ISR). In the most trivial case, there is no other thread in the system and after the `WaitEvent()` the system goes to sleep. These sleep states, which are often implemented as an idle loop, are in principle not bounded. Without further constraints, the ILP problem would become unbounded as we have no information about the duration of the sleep.

The first solution would be to require information about the *maximal* interarrival time of the sporadic event that leads to the thread wake up and derive a minimal interrupt count from the idle time. While this is sound for situations where the system enters the idle loop once, it is a harsh over-approximation for the WCRT. In the following, I will explain (1) why the approach is an over-approximation and (2) why the structure of self-suspension is problematic in general.

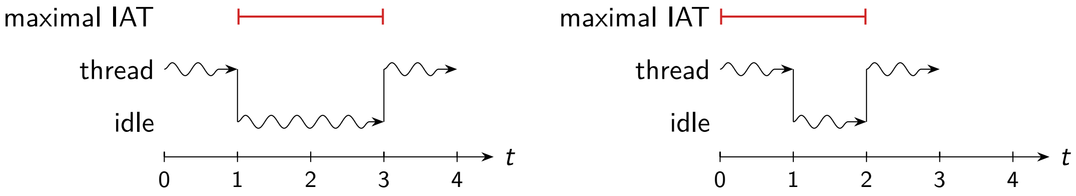
The proposed solution, which bounds the idle time with the maximal IAT, is depicted in Figure 4.9a: After one time unit of execution, the thread issues a `WaitEvent()` system call and enters the idle loop. It is only at this point in time, where the counter for the maximal IAT starts to count down. Two time units later ($t = 3$), the wake up happens, and we end up with an WCRT of 4. However, this is an over approximation of the WCRT with the given program and the given information.

For comparison, the actual WCRT scenario is shown in Figure 4.9. In the worst case, the last signaling of the wake-up event happened at $t = 0 - \varepsilon$ and we have to wait for the full *maximal* interarrival time, which happens after one second of idle time. As the thread executes for one time unit before the sleeping, its first execution phase also accounts for the maximal interarrival time.

The inherent problem for IPET-based WCRT analyses is that this scenario requires a notion of execution *order*. There is a *before* the sleep, which accounts for the maximal IAT, and an *after* the signaling, which, of course, does not account for the maximal IAT. However, IPETs do not have this notion of execution order, as they are implicit, and we could only emulate order by partitioning the application in a before-ILP and an after-ILP, where only the former, together with the idle time, must be smaller than the maximal IAT. With this partition, a modified SysWCET could determine a tight bound for the example system.

However, while this would work for our example with one task and one signal that arrives only once, the situation becomes much harder if the system complexity increases. If there is a loop of computation and sleep time and one wake-up signal that executes N times, we would require $N - 1$ different before-ILPs, as the maximal IAT is reset on every occurrence of the signal. Even if we could determine an upper bound for N beforehand, our approach still handles only one activating signal. In essence, we would have to generate explicit SSTG regions of computation between two idle phases and link them to the maximal IAT.

From a more philosophical point of view, IPET-based timing analyses encounter the described problem as they have only one “clock”: the objective function. While this single clock is enough to handle the sleeping thread that is woken by another thread (micro benchmark #3), which executes on the same clock, it is problematic for external events. In our example, we have two competing



(a) The Idle Over Approximation. If we restrict the idle time with the maximal IAT, the maximal IAT counter starts with the start of the idle loop.

(b) The Actual WCRT Scenario. In the worst case, the last signaling happened right before the thread dispatch and the thread’s execution time counts also on the maximal IAT.

Figure 4.9 – Problems with Self-Suspending Tasks and Idle Times. Example with a self-suspending thread with an execution before and after the sleep of 1 (WCET=2). The maximal interarrival time of the wakeup is 2 time units.

clocks: execution time of the program and the interarrival time of the interrupt. The wake-up (the edge from idle to thread) happens if both clocks expired. Combined with absence of execution order in IPET-ILPs, the problem becomes challenging and is a topic of further research.

Despite all these problems with sleeping threads that are woken by external events, I believe that such idle situations are not typical for the interesting safety-critical paths in a real-time system. Furthermore, with the restriction to the idle time (Figure 4.9a), we have a sound over approximation.

4.6.4.3 More Complex Hardware Models

Another topic for the presented SysWCET approach is the incorporation of more complex hardware models that exhibit hardware-induced delays and inter-dependencies between executed instructions. These effects have been widely studied for WCET analysis and include effects from pipelines [SEE01], data- and instruction caches [FW99], shared buses [CRM10], and DRAM access [WKP13]. Furthermore, besides the inherent hardware complexity, many architectures exhibit timing anomalies [LS; Hec+03], where a locally longer execution path does not result in the actual WCET path. The classical example is a branch mispredict that leads to the accidental prefetching of a later required cache line. As hardware-timing analysis is a problem field of itself, I do not cover it with this thesis and keep its integration with SysWCET as a topic of further research. However, I want to outline directions for this integration.

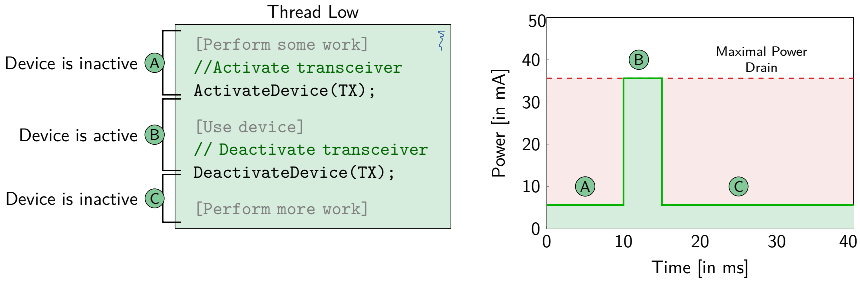
While we could analyze the machine code of each basic block (or even ABB block) in isolation, a more integrated approach is desirable as it promises tighter WCRT results. For example, Li, Malik, and Wolfe [LMW96], Engblom [Eng02] and Theiling [The02a; The02b] presented methods to integrate hardware analysis into IPET-based WCET analyses. As SysWCET is also based on the IPET and uses graphs with control-flow semantic on state- and machine layer, these methods should, in principle, be composable with SysWCET.

Hardware analysis, and especially cache analysis [LMW96], benefits from longer execution paths as a filled cache line can be hit more often. If such a long execution path is split into many small parts, the analysis must assume, at every path beginning, that the cache is empty. Here, the cache analysis could benefit from SysWCET as its whole system view as the SSTG captures all possible execution paths through the system. For example, if a cache line stays in place even if the current thread is only briefly preempted synchronously by another thread, the regular cache analysis would have to assume that the cache set is empty after the preemption. With SysWCET, the longer execution path (through the SSTG) could help to identify the situation correctly and, thereby, SysWCET could refine the cache-related preemption delay [BRA09].

4.7 Application to Worst-Case Energy Consumption

The presented SysWCET approach is able to determine the response time of a thread in the context of a whole RTCS. However, as I already touched, time is not the only physical resource that is consumed during the execution of a program and the RTCS will also consume energy for its operation. While a large energy consumption is a significant financial burden for data-center operators, it becomes mission critical for energy-constrained systems [VHL14]. Energy-harvesting systems, which operate on a small battery that is replenished by extracting energy from the environment (e.g., photovoltaic) [Wäg+15], are examples for such constrained systems. If these systems are safety-critical, it is often not enough to know that the execution will finish in time but it must also be proven that its energy consumption stays within a given energy budget. In a nutshell, without a powered processor, the system will never hold a deadline.

Many authors [Wäg+15; VHL14; CKL00; KE15] focus on the energy consumption of the processor for the *worst-case energy consumption (WCEC)* analysis and assume a continuous and constant power drain for the peripheral devices [LZA09]. However, there are two important aspects that make this CPU-centered view problematic for energy-constrained systems: First, the power consumption of the processor is often much smaller than the consumption of the peripheral devices. For example, the NXP FRDM KL46z [Fre13], which is an energy-optimized ARM M0+-based development board, drains at 3.3 V around 5.6 mA if it executes instructions [▷Wäg+18]. However, an attached ESP8266 WiFi transceiver [Sys17] drains 87.6 mA at 3.3 V. Second, as some peripheral devices, especially transceivers, have such a large energy consumption, they are switchable and turned on only if the system requires their service. Combined, these problems can lead to significant over estimations of the actual energy demand if the consumption of the peripheral devices is considered to be static.



```
[Perform some work]
//Activate transceiver
ActivateDevice(TX);

[Use device]
// Deactivate transceiver
DeactivateDevice(TX);

[Perform more work]
```

Figure 4.10 – Worst-Case Energy Consumption with Switchable Device. Example system (power drain=5.6 mA) with one thread that activates an external transceiver (power drain=30 mA). Adapted from [▷Wäg+18].

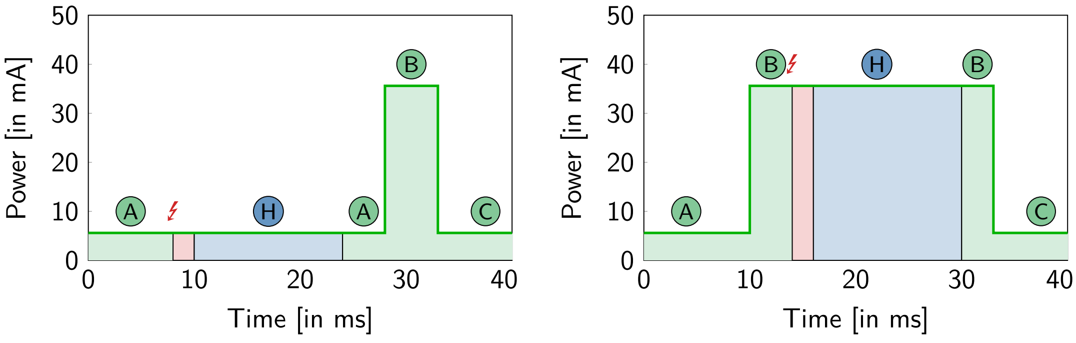
Figure 4.10 exemplifies the situation with one thread, one switchable device, and a typical use-case scenario: The thread performs some preparation (A) before it activates the (energy-hungry) devices (B), transfers data, and receives an answer. After the deactivation of the device, the thread consumes the answer (C), while only the processor consumes energy. If the WCEC analysis assumes the maximal power drain for the whole execution time, we have a huge over estimation (= integral below the red dashed line) over the actual energy consumption (= integral below green line).

4.7.1 System-State Dependent Energy Consumption

power state

For a tight upper energy-consumption bound for this program (Figure 4.10), we multiply and add the WCETs of each section (A, B, C) with the power drain of the system in the respective section. Thereby, the power drain is determined by the *power state* of the system, which consists of the operation mode for each system component. The result is the worst-case energy consumption of the thread; the energy consumption in isolation. However, as we already know from the WCRT analysis, there is normally more than one thread, and we have to calculate the *worst-case response energy (WCRE)*, which considers the WCEC of the thread itself, as well as the consumption and the influence of the other execution threads. Unluckily, with the presence of switchable devices, a compositional approach, which would combine multiple WCECs into a single WCRE, works even worse than for the timing analysis.

In Figure 4.11, we see the energy consumption of the thread from Figure 4.10 if it is collocated with an ISR that activates a high-priority thread H (see Figure 4.12). The two graphs show different paths through the system: In Figure 4.11a, the interrupt triggers before the device is activated (A) and the thread H executes in the low-power mode. Here, the blue area reflects the WCEC of thread



(a) IRQ Before Device Activation

(b) IRQ After Device Activation

Figure 4.11 – Energy Consumption on Different System Paths. Depending on path through the threads and the kernel, the energy consumption can vary significantly as the device power state is carried into the higher-priority thread (H). From [▷Wäg+18].

H as it does not use the device itself. However, if the ISR activates thread H while the peripheral device is active (Figure 4.11b), we encounter the actual system-wide worst-case scenario and the blue area is the WCRE consumption of thread H.

From the example, we see that the WCEC and the WCRE can differ significantly, even for high-priority threads. The WCRE does not only depend on its own execution time and all higher-priority threads but also on those lower-priority threads that manipulate the peripheral power state. Furthermore, as the power drain of peripheral devices can determine the overall energy consumption to such a large degree, a shorter execution path can outweigh the energy consumption on the WCET path. All in all, for a tight WCRE bound, we have to consider the local control flow, interrupts, the scheduling decisions on very fine-grained level, and different power states as they propagate between the threads.

With the SSTG and SysWCET, I already developed a method that combines the thread-local control flows, interrupts, and the scheduling decisions in a worst-case analysis. However, for energy, we have to modify the approach such that it also considers different power states and calculates the WCRE instead of the WCRT.

4.7.2 SysWCEC – SysWCET for Energy Consumption

For SysWCEC [▷Wäg+18], which is the application of the interaction-aware SysWCET analysis to energy consumption, we modify the approach at three different locations: (1) In the ABB construction, we make smaller CFG partitions such that the power state is constant throughout each block. (2) The power state of the system becomes part of the AbSS and is considered in the *system-state enumeration (SSE)* analysis such that each SSTG state executes under a definite power state. (3) In the ILP construction, the energy consumption becomes the objective function, while we still require the worst-case path's execution time for deriving an interrupt count.

First, we modify the formation of ABBs such that they are not only atomic from the RTOS' point of view, but also for the power-drain perspective. For this, we demand that each change in the system's power state is explicitly noted in the source code. This can either be achieved by using explicit function calls (e.g., `ActivateDevice()`, `DeactivateDevice()`) or by other fixed search patters that are matched against the application code. With this requirement, we also restrict the usage of SysWCEC to problems where the power state of the peripheral devices is synchronously *PABB*

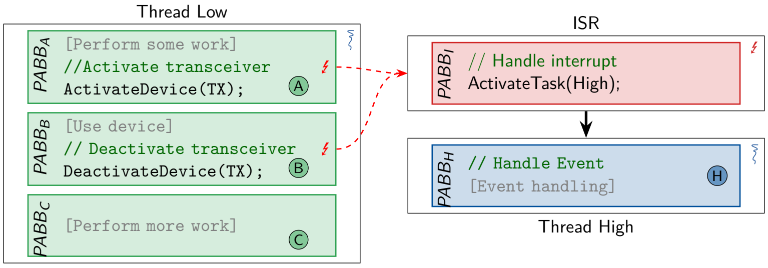


Figure 4.12 – Power Atomic Basic Blocks. Explicit power-state change functions partition the ABB-CFG graph into a PABB graph, where the system's power state is constant during a block's execution. Adapted from [▷Wäg+18].

controlled by the application. We explicitly excluded systems, where other system components, like an independently-operating power-management chip, or the device itself changes the power state of the system.

With these explicit points in the application code that change the drain, we further partition the system into more fine-grained *power atomic basic blocks (PABBs)*. The normal ABB formation (see Section 3.3.2), splits the CFGs, at the system-call sites, into computation blocks and system-call blocks. For PABBs, we further partition the computation blocks into smaller single-entry single-exit regions at the explicit power-state changes. Basically, every point where the power-state changes becomes a pseudo system-call site and the location is split out into its own PABB. Without loss of generality, I will use `ActivateDevice()` and `DeactivateDevice()` as power-state pseudo system calls. With this construction, the system's power state is constant for one execution of a PABB but not necessarily the same for all PABB executions.

In Figure 4.12, we see the (condensed) PABB partitions for our extended example system, where the pseudo system calls were merged into their preceding PABB for a more concise presentation. The activation and deactivation of the peripheral device splits up the single computation ABB into more fine-grained PABBs. It is noteworthy that we still handle interrupts in the same way as for ABBs: PABBs can be interrupted in general, but we make no assumption about the exact location.

power-aware static state-transition graph

With the power-state aware PABB-CFG, we execute a modified SSE analysis that generates a power-state–aware SSTG. For this, we extend each AbSS with a power-state field, which indicates the power drain of the considered peripheral devices. In the simplest case, each device is associated with its idle or its active drain. However, more complex consumption patterns are possible as long as the number of different device states remains discrete. In our example with one processor and transceiver (see Figure 4.10), there are two possible power states (CPU, TX), which we can directly derive the system's power drain in a given AbSS.

$$\text{stateA.power\_state} = (5.6\,\text{mA}, 0\,\text{mA}) \qquad \text{stateB.power\_state} = (5.6\,\text{mA}, 30\,\text{mA})$$

Furthermore, we interpret the pseudo systems calls in the SSE to manipulate these power states. For example, the `ActivateDevice(TX)` generates a followup AbSS that includes the right power state with an overall power drain of 35.6 mA. This combination of power state and RTOS state in the explicit SSE gives us a differentiated picture on the power drain in each state. For example, if a device is enabled only conditionally, the SSTG contains different nodes for the following code blocks; one with and one without the device's consumption. The cost of this detailed picture is a further increase in the SSTG size as, potentially, the number of consumption levels for each device is multiplied with the already huge state space. However, the same arguments and size-reduction strategies that I already discussed for the regular SSTG (see Section 4.6.4.1) also apply here equally.

power consumption in ILP

As the last step of the adaptation, we have to integrate the power-aware SSTG into the ILP formulation such that the maximization objective becomes the WCRE instead of the WCRT. Basically, we still have to calculate the WCET for each ABB and multiply it with the overall power drain of the respective system state in order to get the energy consumption. So, in principle, the formulation of the ILP is the same as for the WCRT problem (see Section 4.4) and we use different costs in the maximization objective.

However, the WCRE ILP has the problem of mixing up execution counts of different power states. As we see in Figure 4.3, all states that reference the same ABB lead to an increased execution count, regardless of the AbSS's power-state. Therefore, we have to derive multiple ABB execution counts $||ABB_n, P||$, one for each power state P in which the block ABB_n gets executed. For this, we derive the overall $||ABB_n||$ count in the glue layer, as described in Section 4.5.3 and distribute it among all power-state specific counts with an ILP constraint:

$$||ABB_n|| = \sum ||ABB_n, P||$$

In order to prevent the solver from choosing only the most energy-consuming $||ABB_n, P||$ in this summation, we restrict its execution count with the respective AbSS counts:

$$||ABB_n, P|| \quad \leq \quad \sum \{\ ||s|| \mid s \in S,\ (\text{s.next\_ABB},\ s.\text{power\_state}) = (ABB_n,\ P)\}$$

Similar to the WCRT formulation, this avoids the duplicated execution of one ABB if it is interrupted and resumed multiple times. However, it also gives the solver the freedom to shift ABB executions to higher power states if the state counts allow for it. This becomes important if an ABB is interrupted in a low-power state, the interrupt activates an external device, and the execution resumes to the same ABB but in a high-power state. In this case, $||ABB_n||$ is one and the solver has the freedom to set that $||ABB_n, P||$ to 1, which maximizes the WCRE.

The problem of mixing up different power states reproduces itself also on the machine layer. There, we solve the problem by duplicating the variables for basic blocks, flow edges, and function calls for each power state they can execute on. We connect the power-state specific ABB counts to the power-state specific ABB-entry basic block. Alternatively to the duplication, we can calculate the WCET of each ABB beforehand and use the result in the SysWCEC ILP.

For the WCRE objective function, we multiply and add the power-state specific basic-block counts with the product of basic-block WCET and the per-time-unit power drain of power state. Furthermore, we can add additional energy budgets at state–state transitions to reflect energy costs that arise when a device is activated or deactivated (e.g., loading of capacitors):

$$E_{WCRE} = \sum ||BB, P|| \cdot \underbrace{WCET_{BB} \cdot POWER_P}_{\text{constants}} + \overbrace{\sum_{s_1, s_2 \in S} ||s_1 \rightarrow s_2|| \cdot E_{switching}}^{\text{Device de-/activation costs}}$$

While we make E_{WCRE} the maximization objective, we still have to calculate the response time T_{RT} of the energy–worst-case path for the global restriction of interrupt counts (see Section 4.5.5). All in all, the SysWCEC method for calculating the WCRE is able to provide control-flow sensitive energy-consumption bounds.

4.7.3 Qualitative Insights from SysWCEC

For [▷Wäg+18], Peter Wägemann performed an extensive quantitative evaluation of SysWCEC's benefits and I had the pleasure to assist him in the setup of the experimental tooling and the validation of the results. In order to keep the here presented description of SysWCEC concise, I deliberate omit these quantitative results and discuss only qualitatively the insights that we gained about the benefits of the interaction-aware approach. For the quantitative evaluation of SysWCEC, please refer to [▷Wäg+18] or to Peter's PhD thesis as soon as it gets published.

flow-sensitive WCRE bounds

Since SysWCEC uses a power-state–based SSTG, the resulting ILP is not only aware of differing power-drain levels but it is also control-flow sensitive. Therefore, it can handle situations, where the energy-consumption rate changes synchronously to the program execution, like it is the case in Figure 4.10. Due to the structural constraints from the SSTG, the solver must choose the lower power state for ABB_A and ABB_C, which avoids the costly always-on approximation for the actual WCRE.

This control-flow sensitivity also extends to RTOS-managed applications (see Figure 4.12): Since the SSE virtually triggers interrupts in every computation block, the SysWCEC ILP allows not only for the actual WCRE path (Figure 4.11b) but also for those paths where the low-priority thread gets interrupted before or after the peripheral is active (see Figure 4.11a). Therefore, the solver is able to choose from both paths and will, as no constraint is violated, select the most expensive path. Thereby, SysWCEC is able to identify the correct WCRE for the low-priority thread.

RTOS services

Even more, SysWCEC is also able to benefit from counter measures against this energetic priority-inversion problem, where a low-priority thread forces a high drain onto a high-priority thread. A developer could create a SRP resource that is owned by both threads and forbid the problematic preemptions by acquiring the resource in the low-priority thread while the device is activated. In this case, only the ISR execution prolongs the device's activation time.

It is noteworthy that this is another instance of the RT–OS mapping problem (see Section 2.2.1) as the usage of the SRP resource does not stem from timing requirements in the RT domain but from energy-conservation considerations on the implementation level. Again, the interaction-aware and implementation-oriented approach that I propose is able to bridge the gap between RT and the implementation domain.

WCRE vs. WCRT

Another insight from the integrated nature of the SysWCEC ILP is the independence of WCRT and WCRE paths. For the same task, both ILPs (SysWCET and SysWCET) calculate the execution time on the inferred execution path. For SysWCET, this value is the maximization objective; for SysWCEC, it is only used to globally bound the number of interruptions. However, since peripheral devices can have a high power drain, the weights in the objective functions can differ significantly, which can lead to different paths for WCRT and WCRE. For example, if an ISR *de*activates a peripheral device, it will not be triggered in the WCRE case while it is surely occurring in the WCRT scenario. Therefore, the WCRE execution path can be shorter than the WCRT path, and the WCRT path can consume less energy than the WCRE path.

4.8 Summary

The logical correctness of a resource-constrained real-time system does not only rely on the correct application code, but also the system's time and energy budgets are sufficient to perform the computation. These two resources are closely related and are consumed along the stream of executed processor instructions. However, it is not the function-local or thread-local instruction stream, but the system-wide execution trace that determines if a task executes, in the environment of a whole RTCS, within a given response-time and energy budget.

For the physical time, the *worst-case response time (WCRT)* analysis estimates an upper bound for the execution duration of a task if it is embedded in a complete RTCS. However, the state-of-the-art method is inherently compositional: We calculate WCETs for each task pessimistically and accumulate them pessimistically according to their mutual preemptions, blockades, interruptions, and dependencies. Thereby, we accumulate not only times but also pessimism as we have to assume the worst-case scenario in every step, even if the concrete combination is an infeasible worst-case path in the actual system. With the compositional approach, we cannot, by its very principle, express constraints about the worst-case path if the condition spans multiple threads; the RTOS becomes an insurmountable border between threads.

SysWCET overcomes this limitation of compositional analyses and provides a method to describe the WCRT estimation as a single, integrated problem formulation. In this integrated formulation, which covers the RTOS, all threads, all ISRs, and their interaction, we can formulate system-wide constraints, avoid more infeasible paths in the analysis, and get tighter WCRT bounds. In addition, SysWCET performs the response-time analysis on the implementation level, after the RTCS is mapped to OS primitives, and, therefore, has a more realistic picture of the RTCS than an analysis that is performed beforehand.

SysWCET starts with the SSTG interaction model and combines it with the machine code of application and RTOS in a single, layered *integer linear programming (ILP)* with the *implicit path-enumeration technique (IPET)*: In the state layer, ILP variables capture how often the system visits an SSTG state in the worst-case scenario; it captures the system semantic, interactions, and all interactions between RTOS and application. The glue layer derives execution count for individual ABB regions of application and kernel code and reduces over estimations by handling of interrupt–resume loops. The machine-layer consists of IPET-ILP fragments that cover the implementation of ABBs and all non–system-relevant functions that are only indirectly referenced by the SSTG. Besides the high automation grade, I could also show that the SysWCET approach is able to provide up to 10.5 percent tighter timing bounds than the compositional approach for the *i4*Copter.

With SysWCEC, I could show that the interaction-aware SysWCET approach is also beneficial for other consumable goods, in this case energy, that is expended during the system's execution. SysWCEC provides tight upper bounds for the *worst-case response energy (WCRE)* of threads even if they execute within RTCSs that include power-hungry but switchable peripheral devices. Thereby, SysWCEC can achieve lower bounds than the always-on approach and exhibits a fine-grained, control-flow sensitive, interaction- and power-state aware, picture on the energy consumption. Furthermore, it is able to benefit from interaction patterns that were intentionally included in the application to conserve energy. This detailed view is achieved by lifting the current power-state configuration to the system-state level and by propagating it all the way down to the SysWCET-based WCRE analysis.

research questions

With this chapter, I could show that a control-flow sensitive view on the interaction in a whole system is possible and beneficial, which gives an answer to my first research question (RQ1). Furthermore, the problem analysis in Section 4.2 and the tighter bounds of SysWCET and SysWCEC also give a (constructive) answer to my second research question (RQ2): The segregation of worst-case analyses along the thread boundaries leads to overly pessimistic worst-case estimations, since the flow of information and constraints about the actual usage patterns stops at these boundaries. Breaking up these boundaries, provides significantly tighter bounds on a system's resource consumption.

5

Automated Kernel Verification

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> Poem by James Whitcomb Riley, around 1885

The logical correctness of a RTCS is dependent on the production of a *correct* calculation result within a given time (and energy) budget. In the last chapter, we have dealt with the timeliness of a system by providing an integrated WCRT analysis that spans multiple threads and that considers the control-flow sensitive interaction between application and RTOS. In this chapter, we will investigate on the correct execution of the system. As my focus is on the boundary between RTOS and application, I will only look at the correctness of this interaction. Therefore, I will present a method to verify that a given kernel–application combination behaves in concordance with the OSEK specification.

While many efforts were done to verify the general correctness of a given kernel, I restrict myself to verifying that the kernel works correctly in presence of a specific application. Thereby, I avoid to grasp the RTOS code semantically, which is especially helpful with generated RTOS instances that are highly optimized for a single application. In a nutshell, the verification is split into two steps that meet at the *static state-transition graph (SSTG)*: First, I formulate constraints about scheduling and interrupt handling in OSEK systems and use a model checker to ensure that the SSTG is in conformance with the standard. Second, I explore the possible state space of the kernel binary dynamically by executing it with an externally-controlled dummy version of the application, which results in the *dynamic state-transition graph (DSTG)*. If SSTG and DSTG match, we can conclude that the properties that we have proven on the SSTG also hold for the DSTG and, therefore, also for the kernel itself.

Related Publications

[▷Dei+17a] Hans-Peter Deifel, **Christian Dietrich**, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. “Automatic Verification of Application-Tailored OSEK Kernels.” In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. FMCAD ’17 (Vienna, Austria). New York, NY, USA: ACM Press, Nov. 2017, pp. 1–8. DOI: `10.23919/FMCAD.2017.8102260`.

[▷Dei+17b] Hans-Peter Deifel, **Christian Dietrich**, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. *Automatic Verification of Application-Tailored OSEK Kernels*. Tech. rep. Extended version of [▷Dei+17a]. 2017. DOI: `10.15488/1761`.

5.1 Problem Field and Related Work

A program or system is *functionally* correct, if its input–output behavior adheres to its specified behavior [DB82]. For every *valid* input, a functional correct program will produce the *expected* output. So, before we make any statement about (partial) functional correctness, we have to define: (1) What is the program/system? (2) What are the valid inputs? (3) What is the expected output?

Directly after the correctness of the hardware implementation, the operating system is the most fundamental component that determines the correctness of the whole system; a misbehaving OS nullifies all correctness proofs of executed applications. For example, an RTOS that does not adhere to the scheduling policy can not only provoke data races in the application, but it will also nullify any statement about the WCRT. Driven by system calls (from the application) and interrupts (from the hardware), the RTOS manipulates its internal state, schedules and dispatches threads, transfers data between protection domains, and controls the I/O channels. Over the observable behavior, we can prove some typical properties: deadlock freedom, correct rescheduling sequences, or task isolation. In order to achieve a solid proof of full functional correctness, multiple properties must be guaranteed. However, also partial proofs over individual properties increase a system's trustworthiness.

Since verification of general-purpose operating systems has a long-standing tradition, I will only highlight a few important milestones. For a more complete overview, please refer to Klein [Kle09].

One of the earliest attempts to verify properties of an OS was done in the UCLA Secure Unix [WKP80] project. Starting with the OS' Pascal code, they showed, by step-wise refinement, that their kernel provides secure and restricted access to data only to authorized processes. Their verification proofed that the kernel will ensure data security for arbitrary user programs.

While UCLA Secure Unix and subsequent projects[FN79; HT05; Bev89] provided partial correctness guarantees, it was only a decade ago [Kle+09] that full functional correctness with respect to any input–output behavior was proven. In a 25 person-year effort, they verified the correctness of the seL4 micro kernel on the C level; Sewell, Myreen, and Klein [SMK13] extended this verification from the C-Code level to the binary level. With translation of a verified Haskell prototype to C, they could check the correctness of the 10 000 lines of C code against 200 000 lines of formal specification.

More focused on RTOS requirements, several attempts were made to formalize the OSEK standard and subsequently verify an implementation. Huang et al. [Hua+11] modeled an OSEK in CSP and verified various properties of the resource protocol, such as mutual exclusion, deadlock freedom, and freedom of uncontrolled priority inversion. While they claim to verify the RTOS on the code level, they verified only a manually derived high-level model of an (unnamed) RTOS implementation.

Vu et al. [VA12; Vu+16] formalized the OSEK standard in Event-B and then verified *designs* of full RTOSs against the specification. Where they looked at fine-grained application models [ZAC15; Zha+13], they verified them only in connection with an RTOS model instead of an actual RTOS implementation.

Waszniowski and Hanzálek [WH08] modeled OSEK using timed automata within the UPPAAL model checker. They include fine-grained application models, but extracted them manually from the application. For a specific combination of application and RTOS model, they verified application properties, like the freedom from deadlocks and performed schedulability analysis of a gear-box controller.

While the previously mentioned attempts to verify OSEK systems only consider high-level application models, Zhang, Choi, and Ogata [ZCO14] consider applications that are written in a simple C-like imperative programming language. They formalize the semantics of the OSEK standard and the OIL language in the K framework, which is a rewrite-based framework to capture language semantics formally. Together with the application, they generate test cases for the RTOS and verify applications by symbolic execution. However, they do neither consider interrupts nor applications that are written in a real-world programming language.

Tigori et al. [Tig+17] show another benefit of formally grasping the RTOS behavior. Similar to my system-call specialization[▷DHL15b; ▷DHL17], they remove dead code from OSEK systems to tailor them towards application requirements. For this, they manually extract the application behavior as *extended finite automata* and perform a reachability analysis with UPPAAL. Unreachable code paths are removed from the kernel. Instead of verifying their specialization procedure or the underlying OSEK implementation, they only validate their approach with the standardized OSEK tests.

All mentioned methods [Hua+11; Vu+16; WH08; ZCO14; Tig+17] for OSEK leave a verification gap between the verified RTOS model and the RTOS implementation. This gap is especially urgent for static systems, like OSEK and AUTOSAR, as they can profit significantly from automated system tailoring, like [Hof+09; Tig+17; Ber+06] or the methods that are described in the second part of this thesis. In cases, where such tailoring is done on the implementation level, it is unclear if the system-generation will preserve the verified properties of the RTOS model, or if its decisions nullify the higher-level proofs. Therefore, there are two options to close the verification gap: Either we verify the system generator and all its analysis and transformation steps, or we verify every generated RTOS instance that is of our interest.

While the verification of the generator would solve the problem one and for all, it would also require a huge effort and it would slow down the incorporation of new optimization techniques in the future. On the other hand, the verification of generation results allows us to focus on the actually required RTOS features, which is sufficient: an application developer does not need a kernel that behaves correctly in all imaginable situations, she needs a kernel that behaves correctly for her specific application. If her application does not use resources, she will indifferent whether the RTOS' resource protocol is faulty in some corner case. However, per-instance verification requires an effort every time the artifact is regenerated, which makes it crucial to automate the verification process.

In this chapter, I provide an automated verification process for generated RTOS instances that focuses on the part of the RTOS behavior that is actually relevant for the application at hand, instead of attempting a general RTOS verification. The verification process is based on model checking and dynamic exploration of the kernel's state space. Thereby, it is possible to (a) work on the actual kernel binary as it is produced by the generation and compilation toolchain, and (b) fully model check the entire application/RTOS system including interrupts, which is not covered by any of the cited literature. Since adherence to the scheduling and interrupt policy is crucial for the timeliness of the whole system, I will focus on this RTOS property.

This verification effort was joined work with the group of Lutz Schröder and two brilliant Master students: Hans-Peter Deifel, who worked on the dynamic state exploration, and Merlin Göttlinger, who formalized key aspects of the OSEK standard and tamed the NuSMV model checker.

5.2 Automated Application-Specific Verification

The per-instance verification uses the *static state-transition graph (SSTG)* as the crystallization point of the verification (see Figure 5.1): The SSTG is calculated according to the OSEK specification by the dOSEK framework and it is used for all kinds of system tailoring, like proposed in [▷DHL15b] or in Chapter 6. It captures the expected behavior of the kernel binary in presence of the given application. To ensure that the generator really adhered to the OSEK specification, we use a model checker to statically verify that the SSTG is correct with respect to the standard. For this, we formalized key aspects of the OSEK standard in *computation-tree logic (CTL)* and model check the SSTG against this

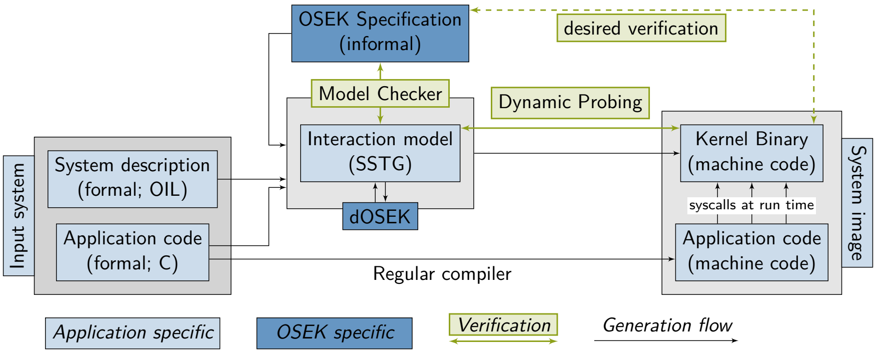


Figure 5.1 – dOSEK System Generation and Behavioral Verification. The SSTG acts as an auxiliary to split the verification of the specialized kernel binary into two components. Adapted from [▷Dei+17a].

specification. In a second step, we dynamically execute and probe a surrogate system, which consists of the real kernel binary and a mock-up application that exhibits the same system-call ordering as the real application. The resulting DSTG is checked for isomorphism, which implies structural and behavioral equivalence, with the SSTG. If both graphs are isomorphic, the verified properties on the SSTG holds for the kernel binary, even if it gets combined with the actual application.

5.2.1 Verification of the Interaction Model

For the verification of the SSTG, we use the NuSMV model checker to ensure that the SSE analysis produced a SSTG that is in compliance with the OSEK specification. In general, model checking ensures, by exhaustive and automated checking, that a given model adheres to some *formal* specification. For an overview about model checking, please refer to Jhala and Majumdar [JM09]. For the per-instance verification, we use NuSMV [Cim+02], which is a symbolic model checker for *finite-state machines (FSMs)* that supports specifications in *computation-tree logic (CTL)* or *linear temporal logic (LTL)*. We derive a NuSMV model from the SSTG and check constraints from OSEK specification against it.

model

First, we generate a NuSMV model from the SSTG and information from the OIL file. As starting point, we use the normal SSTG variant that is derived from the thread's ABB-CFG (see Section 3.4.2). However, we additionally label the edges with the ABB or the external event that causes the transition, which results in a deterministic labeled transition system, if we put it into process-theoretic terms. Since many of the transitions are induced by computation ABBs, which cannot influence the RTOS state, we replace them by ε transitions and perform standard ε-elimination. This preprocessing step, which reduces the number of states, is similar but distinct from the usage of ASMs (see Section 3.4.3.2): There, the ε-elimination was done on the CFG before the SSE analysis; Here, we perform the elimination *after* the state enumeration. This results in a transition system that is not only trace equivalent with the real system (actions are performed in the same order) but even bisimilar [Mil80] (each state has one or multiple corresponding states in the other graph).

This minimized and labeled SSTG already resembles a model that NuSMV understands with the exception that states in NuSMV models do not have edge labels, but transitions are taken nondeterministically in the checking process. Therefore, we convert the edge labels into state labels by pushing them into the followup state, such that each SSTG state results in as many model states as the SSTG states have incoming edges. For example, if state Z is reached via `ActivateTask()`

and `ChainTask()`, the NuSMV model has two states: (`ActivateTask()`,Z) and (`ChainTask()`,Z). These states are hold in global state variables in the NuSMV model: (a) the variable syscall contains the name of the system call (i.e. the label of the edge) that took the system into the current state, and (b) the variable state carries the current state in form of the node id from the SSTG. As an example, Listing 5.1 shows a fragment of the NuSMV model that contains these variables and their value domains, which are statically extracted from the SSTG.

Listing 5.1 NuSMV Model - Global Variables. The state and syscall variables identify the current SSTG state and the system call that lead to the current state. From state, we will derive other variables that are part of the AbSS.

```
// state and syscall variables with their domain
state   : { ABB_67_0, ABB_4_0, ABB_23_0, ABB_63_0, ... };
syscall : { StartOS, TerminateTask, ActivateTask_High, interrupt_37, ... };

// Derived from state
running : { Idle, Low, Med, High, ISR };
```

Besides syscall and state, we introduce global variables for all fields of the AbSS. The values for these fields are deterministically derived from state and updated on each transition with constant values from the SSTG. For example, the running variable in Listing 5.1 can be directly derived from state if the SSTG is given (Listing 5.2, line 21-29). We introduce these variables to make the formulas and the counterexample traces easier to comprehend. As another measure to increase the comprehensibility of the model, we instantiate a NuSMV module for each system object (i.e., threads, ISRs) and use these instances as name spaces. For example, the state and (dynamic) priority of a thread t are referred to by t.state and t.priority (see Listing 5.2, line 37-39).

Listing 5.2 NuSMV Model - Transitions. The model checker can choose the next state nondeterministically. From the current and the next state, the system call that leads to this transition is chosen and we derive the values for the AbSS fields from the next state.

```
 1 // Choose the next state
 2 init(state) := ABB_67_0;
 3 next(state) := case
 4   ...
 5   (state = ABB_67_0) : {ABB_4_0, ABB_23_0, ABB_63_0};
 6   ...
 7 esac;
 8
 9 // Derive system call that has to be invoked for
10 // the transition between state and next(state)
11 init(syscall) := Start;
12 next(syscall) := case
13   ((state = ABB_67_0) & next((state = ABB_4_0))) :
14      ActivateTask_High;
15   ((state = ABB_67_0) & next((state = ABB_23_0))) :
16      interrupt_37;
17   ((state = ABB_67_0) & next((state = ABB_63_0))) :
18      TerminateTask;
19   ...
20 esac;
21 // Derive the other AbSS field from the current state
22 init(running) := Low;
23 next(running) := case
24   next((... | (state = ABB_67_0))) : Low;
25   next((... | (state = ABB_4_0))) : High;
26   next((... | (state = ABB_23_0)) : ISR;
27   next((state = ABB_63_0)) : Idle;
28   ....
29 esac;
30
31 init(state_task_ISR) := Suspended;
32 next(state_task_ISR) := case
33   next(((state = ABB_67_0) | ABB_4_0 ) : Suspended;
34   next(((state = ABB_23_0) | ... )     : Running;
35 esac;
36
37 // Semantic Namespaces for System Objects
38 // (e.g., task_ISR.state)
39 task_ISR : Task(..., state_task_ISR, ...);
```

With these global variables in place, we encode the transitions of the SSTG in the NuSMV model. The only degree of freedom for the model checker is the state variable, which is chosen nondeterministically from the followup states of the currently running state (Listing 5.2, line 3-5). From the current state and next(state), the value of next(syscall) is uniquely determined (line 12-20).

Furthermore, we also update the value of all other variables according to `next(state)` for the next step in the model (line 23-35). In the example (Listing 5.1 and Listing 5.2), we see a slice of an NuSMV model that was derived from a system that starts in Low-priority thread in ABB_{67}. There, the thread can either call `ActivateTask()`, which brings it to the state ABB_4_0 and the high-priority thread, or it can call `TerminateTask()`, which brings the system to the Idle state. Additionally, an interrupt can occur, which results in an activation of ISR.

specification

Since the OSEK standard describes the system behavior only informal, we had to formalize parts of the OSEK standard by ourselves. Currently, the formalization covers the standard roughly up to conformance class ECC1 (see Section 2.1.3). While interrupts are supported in general, we did not cover alarm objects to reduce the formalization effort. However, they can be emulated by modeling alarms with a manually constructed timer ISR. Here, I only want to highlight a single property of this specification that regulates the scheduling of threads. A detailed discussion of the specification can be found in [▷Dei+17b], which is an extended technical report of our FMCAD paper [▷Dei+17a], that includes the full description of the OSEK specification, as well as a detailed discussion of individual evaluation results.

The specification is written as CTL-formula templates that we instantiate according to the OIL file of the current system and check with NuSMV against the SSTG-derived model. Figure 5.2a shows an example template that expresses that the scheduling for a thread t is correct. It specifies that

$$
\begin{aligned}
&\mathsf{AG}((t.\mathsf{state}=\mathsf{Suspended}\rightarrow\psi_1)\wedge(t.\mathsf{state}=\mathsf{Running}\rightarrow\psi_2)\wedge(t.\mathsf{state}=\mathsf{Ready}\rightarrow\psi_3)\wedge(t.\mathsf{state}=\mathsf{Waiting}\rightarrow\psi_4)),\text{ where}\\
&\psi_1\equiv\mathsf{AX}((t.\mathsf{state}=\mathsf{Ready}\vee t.\mathsf{state}=\mathsf{Running})\rightarrow\mathsf{syscall}\in\mathbb{ACT}(t))\\
&\qquad\wedge\mathsf{allOthersPreemptible}(t)\rightarrow\mathsf{AX}(\mathsf{syscall}\in\mathbb{ACT}(t)\rightarrow(t.\mathsf{state}=\mathsf{Running}\leftrightarrow t.\mathsf{isHighestPriority}))\\
&\qquad\wedge(\neg\mathsf{allOthersPreemptible}\rightarrow\mathsf{AX}(\mathsf{syscall}\in\mathbb{ACT}(t)\leftrightarrow t.\mathsf{state}=\mathsf{Ready}))\\
&\qquad\wedge\mathsf{AX}((t.\mathsf{state}=\mathsf{Suspended}\rightarrow\neg\mathsf{syscall}\in\mathbb{ACT}(t)))\\
&\psi_2\equiv\mathsf{AX}((t.\mathsf{state}=\mathsf{Ready}\leftrightarrow\mathsf{othersWillPreempt}(t))\wedge(t.\mathsf{state}=\mathsf{Suspended}\leftrightarrow\mathsf{syscall}\in\mathbb{TSC}(t))\wedge(t.\mathsf{state}=\mathsf{Waiting}\leftrightarrow\mathsf{waitSc}(t)))\\
&\psi_3\equiv(\mathsf{allOthersPreemptible}(t)\rightarrow\mathsf{AX}(\mathsf{syscall}\in\mathbb{SC}\rightarrow(t.\mathsf{state}=\mathsf{Running}\leftrightarrow t.\mathsf{isHighestPriority})))\wedge\mathsf{AX}(t.\mathsf{state}=\mathsf{Running}\vee t.\mathsf{state}=\mathsf{Ready})\\
&\psi_4\equiv\bigwedge_{e\in\mathbb{E}(t)}\big(t.\mathsf{isWaitingFor}(e)\rightarrow((\mathsf{allOthersPreemptible}(t)\rightarrow\mathsf{AX}(e.\mathsf{set}\rightarrow(t.\mathsf{state}=\mathsf{Running}\leftrightarrow t.\mathsf{isHighestPriority})))\\
&\qquad\wedge(\neg\mathsf{allOthersPreemptible}(t)\rightarrow\mathsf{AX}(e.\mathsf{set}\rightarrow t.\mathsf{state}=\mathsf{Ready}))\\
&\qquad\wedge\mathsf{AX}((t.\mathsf{state}=\mathsf{Waiting}\wedge\neg e.\mathsf{set})\vee t.\mathsf{state}=\mathsf{Running}\vee t.\mathsf{state}=\mathsf{Ready}))\big)\\
&\mathsf{allOthersPreemptible}(t)\equiv\textstyle\bigwedge_{ot\in\mathbb{NPT}\setminus\{t\}}ot.\mathsf{state}\neq\mathsf{Running}\\
&\mathsf{othersWillPreempt}(t)\equiv\mathsf{syscall}\in\mathbb{SC}\setminus(\mathbb{TSC}(t)\cup\mathbb{WC})\wedge\textstyle\bigvee_{O\in\mathbb{PRE}(t)\setminus\{t\}}O.\mathsf{isHighestPriority}\\
&\mathsf{waitSc}(t)\equiv\textstyle\bigvee_{e\in\mathbb{E}(t)}\neg e.\mathsf{set}\wedge t.\mathsf{isWaitingFor}(e)
\end{aligned}
$$

(a) CTL Formula for Thread-State Transitions

Set	Name/Description
$\mathbb{SC}$	Scheduling calls, i.e. system calls that lead to a (re-)scheduling of threads
$\mathbb{WC}$	WaitEvent System calls
$\mathbb{ACT}(r)$	System calls that activate the control flow (i.e. thread or ISR) r
$\mathbb{E}(t)$	Events of thread t
$\mathbb{TSC}(r)$	Terminating system calls, i.e. system calls that would lead to the termination of r
$\mathbb{NPT}$	Nonpreemptible threads, i.e. threads that cannot be preempted by higher priority threads
$\mathbb{PRE}(r)$	Control flows that could preempt r

(b) Constant Finite Sets

Operation	Name/Description
$\mathsf{AG}(X)$	The proposition X holds for **A**ll states **G**lobally.
$\mathsf{AX}(X)$	The proposition X holds for **A**ll ne**X**t states.

In CTL, the development of a system's states over time is expressed as a tree: each node is a system state and each child of a state is a possible future. Over this branch tree, which is build up by the model checker, we can formulate propositions that refer to a state and its (recursive) children.

For example, the $\mathsf{AX}(X)$ operator states that the proposition X is true for all direct children of the current state. The $\mathsf{AG}(X)$ operator states that the proposition X must be true for all states in the branch tree.

(c) CTL Operators

Figure 5.2 – Slice of the OSEK CTL Specification. This slice specifies how the state of thread t develops over time.

every change to a thread's state is legal, and every change that OSEK requires is in fact made by the system. The actual CTL formula is generated by instantiating the template for every thread t, and (slightly modified) for every ISR, in the system. In these formulas, we often have to reference finite sets of system objects or constants (see Figure 5.2b) that are declared by the OSEK standard or declared in the OIL file. These constant finite sets are statically expanded into finite conjunctions or disjunctions.

The formula in Figure 5.2a is structured into four subformulas $\psi_1, \ldots, \psi_4$ that handle the allowed state transitions from one of the four possible starting thread states: Suspended, Running, Ready, Waiting. For easier comprehension of this large formula, we use the following abbreviations for state properties and CTL fragments:

- t.isHighestPriority specifies that control flow t is in the Ready or the Running state and that its dynamic priority is higher than any other runnable control flow.
- allOthersPreemptible(t) states that all runnable threads, besides t, are preemptable. In combination with t.isHighestPriority, this formula ensures that no interrupt is currently running as interrupts are interpreted as non-preemptable threads.
- othersWillPreempt(t) denotes that thread t is preempted by a higher-priority thread O as a result of issuing a system call.
- t.isWaitingFor(e) states that the thread t is waiting for the event e to occur.
- waitSc(t) formalizes that thread t is waiting for at least one event that is not set.

Formula ψ_1 handles the case where t was previously suspended. It becomes runnable (Ready or Running if a system call that activates t ($\mathbb{ACT}$) occurred. Depending on the priority of the thread that issued that system call, we either preempt that thread and become Running or we only become Ready. If no activating system call occurs, the suspended thread stays in its suspended state. When t is currently Running (ψ_2), it only changes its state (a) to Ready if it gets preempted by another higher-priority thread, (b) to Suspended if it terminates, or (c) to Waiting if it waits for an unset event to be set. A preempted thread that is Ready (ψ_3) becomes only Running if all other threads are preemptable and a rescheduling system call that made it the highest priority was issued; otherwise it remains only Ready. Waking up from the Waiting state (ψ_4) only happens in the next state if an event, for which the thread waited, was signaled. These four subformulas reflect the edges in the classical thread-state transition diagram as it is also specified by OSEK [OSE05].

Given to NuSMV, this specification is checked against the SSTG-derived model of the interaction between application a kernel. Thereby, we use the detailed values of the AbSSs to connect the specification to the individual states, while we also ensure that the values of these fields were calculated correctly. Therefore, such a verification ensures that the SSE analysis worked according to the specification and that the SSTG adheres to the OSEK standard for this specific application.

5.2.2 Dynamic Exploration of the State Space

Until now, our verification effort was decoupled from an actual OSEK implementation, and we only verified that the SSTG reflects the desired system behavior. However, it is yet to be answered if a given, potentially specialized, kernel will exhibit the same state transitions if it is executed together with the specific application. A classic approach to ensure such a property would be to grasp the semantics of the kernel code formally and refine this model until we can match it against the specification. With the verified SSTG, however, we can use another technique and explore the state space of the kernel binary exhaustively by *dynamic probing*.

Goal of the dynamic probing is to extract the *dynamic state-transition graph (DSTG)*, whose nodes are opaque without inner structure and whose edges are labeled with the system-call sites that guard the transition. The nodes are identified and disambiguated by a hash (or checksum) over the main memory that holds the relevant RTOS state. Later on, we will check this extracted DSTG against the SSTG for equivalence (namely isomorphism).

Listing 5.3 Mockup Application for Dynamic State Exploration. The mockup application behaves, in its sequence of issued system calls, exactly as the original application. However, in ABB it calculates a hash over the relevant RTOS memory and prints the hash together with the current ABB and the (possibly) invoked system call.

```
TASK(Low) {
ABB1:
  int decision = read_decision(ABB1);
  if (decision == interrupt_37) {
     write_state_hash(at: ABB1, next: interrupt);
     trigger_interrupt();
     goto ABB1;
  }
  write_state_hash(at: ABB1, next: EPSILON);
  if (decision == ABB2) goto ABB2;
  if (decision == ABB3) goto ABB3;

ABB2:
    write_state_hash(at: ABB2, next: ActivateTask);
    ActivateTask(High);
    goto ABB3;

ABB3:
  write_state_hash(at: ABB3, next: TerminateTask);
  TerminateTask();
}
```

In more detail, we extract the DSTG by executing and probing the generated kernel binary with all possible system-call sequences that can originate from the given application. For this probing, we generate a mock-up from the application's ABB graphs that retains the control-flow structure and all system-call– and function-call–block contents. We equip the mockup with an external input and an external output channel. In each ABB, the mockup application calculates a hash over that memory region that holds the relevant RTOS state and writes it to the output channel together with an identifier for the current ABB. In each computation ABB, the mockup reads a probing decision from the input and it (a) triggers an interrupt, or (b) steers the control flow to one of the successor blocks. In a nutshell, the mockup is a replacement for the application that can be controlled from the outside. An example mockup that reflects our running example, which is known from Figure 4.5, is shown in Listing 5.3.

We compile the generated mockup application like the original application and link it with the kernel binary that should be verified into a system image. This system image gets executed on the same execution platform that is supposed to run the final system.

The input and output channel are connected to the *dynamic state explorer (DSE)*, which steers the execution of the kernel through its entire state space. By sending branch and interrupt decisions, the DSE traverses the actual state space of the kernel dynamically in a depth-first search; it also regularly restarts the system to choose different paths from the initial state. The depth-first search finishes after the DSE took every possible branch and triggered every interrupt in every discovered state. Thereby, the set of possible decisions for any given state is statically deduced from the ABB graph of the application. From the outputs, the DSE constructs the DSTG where each distinct state hash becomes an individual node, and edges are drawn between hashes that follow each other in the output stream. If the predecessor hash comes with a system-call–site identifier, we label the edge with a system-call–site identifier; otherwise it is labeled with an ε.

DSTG determinism

Since we omit the processing logic in the mock-up and since we cannot foresee the application's actual input, any form of conditional branching in the original application turns into nondeterminism, and we get a nondeterministic transition system. However, we apply the same ε-elimination on the DSTG as we have applied to the SSTG. The ε-eliminated DSTG, as well as the SSTG, is deterministic, since each edge is labeled with a system-call site, which deterministically indicates its influence on RTOS state. This also holds for interrupts, since every interrupt transition gets labeled with the interrupt number, which also exactly describes its effect on the RTOS state. Summarized, in both STGs, all edges that originate from the same state always have different labels.

Finally, we check the DSTG and the SSTG for isomorphism, which is cheap for deterministic labeled transition systems with a single entry state. With induction, we show for every state that it allows for the same output transitions. In fact, this check is so cheap that we perform it on the fly directly after the state exploration. If the implementation was correct, both graphs are isomorphic, since the DSTG states are identified by a hash over the full RTOS state, which are manifestation of the AbSS fields.

5.3 Experiments

In order to validate the proposed verification method, we performed positive and negative tests with different OSEK applications, which are taken from the test suite of *d*OSEK. For the positive tests, we verified that the kernel generator produced correct kernel binaries that expose the OSEK-conform behavior. For the negative tests, we introduce faults (e.g., a corrupted SSTG) and let the verification process detect the faults.

5.3.1 Positive Tests

For the positive test, we took 56 systems from the *d*OSEK test suite and two derived *i4*Copter (see Section 4.6.3) variants, and verified them, fully automated, with the proposed two-step process. For the *i4*Copter variant (copter), we had to replace the three alarms with one user-defined ISR that manages counters and periodic thread activations like the generated timer ISR would have

	System Objects				NuSMV Model		Verification	
Name	ISRs	Tasks	Events	Res.	Reach. States	Diameter	Run Time	Memory
bcc1-resource1j	0	6	0	4	26	16	0.10 s	27 MiB
bcc1-sse1c	0	6	0	4	24	14	0.08 s	28 MiB
ecc1-bt1g	0	6	2	2	10	10	0.05 s	24 MiB
ecc1-event1e	0	4	4	2	13	13	0.11 s	29 MiB
bcc1-isr2d	1	4	0	2	21	10	0.07 s	23 MiB
timing-abcomp	1	3	2	2	77	13	0.14 s	27 MiB
copter-small	3	11	0	3	1366	29	19.44 s	250 MiB
copter	4	12	0	3	4458	32	147.15 s	829 MiB

Table 5.1 – Verification Effort for Positive Tests

done it. This modification was necessary as the verification toolchain does not yet support alarms. Furthermore, another variant (copter-small) was derived by excluding the remote-communication ISR and the subsequent handling thread. For both *i4*Copter variants, we used additional information about the real-time system to reduce the size of the SSTG (see Section 3.4.3.1).

The verification and the dynamic state exploration was done on a 2.4 GHz Intel Core i7-5500U machine with 8 GB of main memory. As RTOS implementation, we used a user-space variant of the *d*OSEK RTOS that uses a Linux process as its underlying "CPU" abstraction and performs user-level scheduling. By running on top of Linux, the read and the write channels were trivial to implement.

For the model-checking part of the evaluation, we could show the correctness of the SSTG construction for all 58 test systems. In order to indicate the required verification run time, we selected eight systems and show the results in Table 5.1. For these systems, the table shows the number of relevant system objects, the number of reachable states in the NuSMV model, as well as its diameter (i.e., longest loop-free path), to characterize the checked model. In all smaller benchmarks, the verification was completed in under a second. Only for the *i4*Copter variants, it took significantly longer, but the verification always finished in under three minutes.

For the DSTG exploration, we probed the same 58 OSEK systems in two generator configurations: without any optimization and with system-call site specialization as described in [▷DHL15b]. For all 116 systems, we could show full isomorphism between the SSTG and the DSTG. The probing time and the isomorphism check for the smaller test cases took under 1 second, and only the *i4*Copter variants required a slightly longer duration: 2.13s (0.81s for copter-small) for the probing and 0.17s (0.04s) for the isomorphism checking. During the development, we also probed the *i4*Copter without additional constraints from the RT properties and discovered more than 400 000 states in under 4 minutes.

5.3.2 Negative Tests and Fault Injection

From the positive tests, we have shown that our approach provides positive correctness statements about correct kernel implementations. However, it is yet to be shown that it also able to spot a faulty system. Therefore, we take correct systems and introduce faults on different levels in order to perform *negative tests* of our verification toolchain.

fault model

We introduce two different classes of faults at different stages of the system generation: (a) mutations of the SSTG, or (b) modification of the kernel generator's OIL input. For the SSTG-level faults, we randomly merge two states or add a state-transition edge, which gets labeled with a randomly chosen system call. These faults will, most likely, produce an incorrect SSTG with invalid transitions. For the OIL-level faults, we (randomly) exchange the priority of two threads, or toggle the preemptability of a thread, or flip the auto-start flag of a thread. For the OIL faults, the generator will produce a valid SSTG and generate the kernel accordingly. However, as the generator was based on a wrong system configuration, the verification will detect that the SSTG is incorrect if compared to the actual (correct) OIL file.

At this point, it must also be noted that not all introduced faults will necessarily lead to an actual incorrect system. This can happen, for example, if the OIL fault does not change the rescheduling sequence: if two priority-flipped threads are constrained by a directed dependency, where one thread activates the other one at the end, their priority is irrelevant. For the SSTG faults, it can happen that an additional edge was indeed a legal transition and could, therefore, not be detected by the model checker. However, we also encountered additional edges that were not detected since they violated parts of the OSEK specification that we have not formalized yet (i.e., release a resource that is not reserved). In the case of two merged states, it happened that the downstream ε-elimination would have had the same effect.

For the model checking, we injected 188 faults into the test cases, which did not influence the required model-checking time significantly if compared to the unfaulty test cases (Table 5.1). In 177 cases, the injected fault was correctly detected by the model checking. In the other 11 cases, we confirmed by manual verification that the fault did not lead to an error and was indeed benign. For a more detailed discussion about each individual benign faults, please refer to Deifel et al. [▷Dei+17b].

For the dynamic state exploration, we inserted 81 OIL-level faults for the kernel generation and compared the extracted DSTG against the intended SSTG. In 61 cases, the isomorphism check could detect the error. In the other cases, we manually verified that the introduced OIL modification had, in the investigated test case, no influence on the system behavior.

5.3.3 Discussion

isomorphism

The isomorphism between SSTG and DSTG proved to be robust, even in the presence of an optimizing system generator. This isomorphism is tightly coupled to the question if there is a bijection between the inputs of the RTOS-hash function and the AbSS fields. Therefore, there are three cases that explain why we can reach isomorphism (most of the time) for a correct generator: (1) If a *correct* generator removes an RTOS-state field from the implementation, the field was not required and did not hold significant state information; any modification to the field would have been unnecessary, and it could not have had any influence on the RTOS behavior. Hence, the now more condensed state results in a DSTG that is potentially smaller ($\leq$) than before. However, as the generator is correct, the DSTG remains of equal size and isomorphic to the SSTG. (2) If it adds a field without modifying it, all hashes are only concatenated with a constant string and the whole DSTG state space gets shifted by this constant. (3) Only if it adds a field and modifies it, things become more interesting: However, if the field modifications do not influence the rescheduling behavior, isomorphism runs into a problem: For example, if the RTOS flips a bit in its state after each system call, the number of DSTG states doubles as the hashes differ. Nevertheless, in this case, always two states (with the 0 and the 1) from the DSTG are, in combination, equivalent to one SSTG state. Therefore, we can fall back to use the weaker bisimilarity equivalence relation.

RTOS hash

In order to reach isomorphism, we had to perform a somewhat laborious tuning process about the inputs of the RTOS-hash function. If we were conservative and hashed too much memory, we hit case (3) for many test cases. Furthermore, hashing a large part of the main memory, more than necessary, also increases the probing time as the state space multiplies by the possible assignments of every unnecessary variable that gets included into the hash. On the other hand, if our hash did not cover all important state variables, we got a too small DSTG, which cannot distinguish between important state–state transitions of the kernel. Therefore, it might in fact be easier for the maintenance of the tool chain and the applicability of the presented approach to replace isomorphism with bisimilarity.

change sensitivity

Another aspect of the presented approach is the re-usability of its verification results and its sensitivity to application or generator changes. For a modified application, we only have to redo the model checking if the SSTG changes, which happens if the system configuration or the ABB-graph structure changed. In this case, also the DSTG has to be extracted again. If we change the system generator or use a different generator setting, we only have to redo the DSTG part of the verification and probe the old mockup application with the newly created kernel binary. Here, using the SSTG as the mediating data structure of the verification allows for such partial re-verification of the system binary.

5.4 Summary

The correctness of an RTOS implementation is a crucial corner stone for the logical correctness of the whole system. Especially in the presence of optimizing system generators, which use techniques that I propose in this thesis, it becomes hard to gather enough evidence to substantiate a correctness claim for the resulting system image. While verification of the generator is possible, a formal verification of each produced instance, if done in an automated fashion, seems more feasible and keeps the generator flexible for future changes.

One aspect of functional correctness is that a kernel binary must behave, in presence of a given application, in concordance with the specified behavior. The presented approach uses the interaction model as an intermediate data structure to aid this verification goal: Coming from the RTOS specification, we use a model checker to ensure that the application-specific *static state-transition graph (SSTG)* was calculated correctly. In a second step, we proof that the SSTG is equivalent to a dynamically-extracted state-transition graph (DSTG).

On a more technical level, we formalized parts of the OSEK standard as CTL formulas and used the NuSMV model checker on a translated variant of the SSTG to show that the *system-state enumeration (SSE)* worked correctly. In order to show that a given kernel binary, which can be produced by an optimizing system generator, exhibits the specified behavior, we produce a mockup application from the ABB graphs. This mockup, together with the finally-deployed kernel binary, gets executed on the indented computing platform. The *dynamic state explorer (DSE)*, which controls the mockup's branching decisions and interrupt-trigger points, steers the system execution through the entire state space of the kernel binary and constructs the DSTG. In the process, we distinguish between different system states with a hash over the relevant RTOS memory regions.

In the evaluation, we could show that the presented approach allows for an automated application-specific verification of optimized kernel binaries. Furthermore, with fault injection of the generation process, we could show that our formalization was able to spot all errors that would have affected the behavior of the system. All steps of this verification procedure were done automatically and finished in a reasonable amount of time.

research question

In this chapter, I could show how a control-flow sensitive interaction analysis provides the necessary information for an automated per-instance verification of the kernel behavior that keeps close to the actual requested RTOS functionality. With the interaction model as its centerpiece, where both parts of the verification can meet, we can make statements about the functional property "correctness" of a system; even in the presence of optimizing system generators. Thereby, I could provide an answer for my third research questions (RQ3): Only with the whole-system view on the application-requested interaction, we were able to provide a correctness guarantee for the execution-coordinating role of the RTOS for a given application without verifying the kernel code or the generator toolchain.

Part II

Optimization

6

Semi-Extended Tasks

Stack as a Shared Resource

> We must either learn to live together as brothers or we are all going to perish together as fools.
>
> *A Christmas Sermon*, 1967, Martin Luther King Jr.

In the first part of this thesis, I used the static interaction analysis to investigate on the properties of an eventually deployed RTCS. In this and the next chapter, I will constructively improve system properties, namely stack consumption and reschedule overhead, in a post-mapping optimization step that is only possible with the proposed whole-system view on the interaction analysis.

With the in-depth knowledge about the thread activation and preemption patterns, I propose the fine-grained sharing of statically-allocated stack space among different threads. Traditionally, we consider the stack space as *the* private resource of an RTOS thread. With the proposed fine-grained stack sharing, I break up this monolithic view and expose the potentials to reduce the overall memory requirement of the RTCS. Thereby, the proposed *semi-extended task (SET)* approach leaves the application logic in place and does not limit the flexibility of the system; it only exploits inefficiencies in the resource usage that originate from the segregation of the real-time application into multiple threads.

Related Publications

[DL18] **Christian Dietrich** and Daniel Lohmann. “Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads.” In: *Proceedings of the 39th IEEE Real-Time Systems Symposium 2018*. Ed. by Sebastian Altmeyer. Nashville, Tennessee, USA: IEEE Computer Society Press, 2018. DOI: `10.1109/RTSS.2018.00049`.

6.1 Memory Consumption of Statically-Allocated Stacks

Besides processing time on the CPU, volatile memory is the most important resource of a computing platform as it holds the program state and intermediate computation results. Its size defines the upper limit for the application complexity, especially if a system, like many embedded real-time systems [Bro06], has no modifiable background storage to hold the ever-changing data. Therefore, the RAM requirement of a RTA is an essential non-functional property.

Looking at a system's RAM requirement from the perspective of the procurement department, another aspect arises for deeply embedded systems: RAM is expensive and its price does not scale linearly. In Table 6.1, we see the price development for the same processor core if only RAM and the non-volatile flash memory is enlarged. First, we see that the amount of flash memory, which is used on this Harvard architecture to store the code, is available in much larger quantities than the on-chip SRAM cells; an observation that also holds, on a different scale, also for the desktop and server market (i.e., SSD storage vs. DDR RAM). Second, we see that RAM comes, for this MCU series, only in power-of-two quantities, which results in a staircase-shaped cost function for the memory demand. In a nutshell, for a system that already uses 8191 bytes of RAM, the 8192nd byte is already paid for, while the 8193rd byte costs about 12 cents, *per sold unit*. Vice versa, also small RAM savings can result in large cost savings if it allows us to purchase the next smaller MCU.

The memory demand of an application can be divided into two classes of allocations: The static allocations (e.g., global variables) have the same lifetime as the program itself and must kept reserved at all times. For the static RTCS, where the RTA is the only workload, these allocations directly reduce the amount of available memory. As these static allocations block whole areas of RAM, we can downright speak of *memory consumption* instead of memory demand.

memory consumption

On the other hand, dynamic allocations live shorter (e.g., for a single job execution) as they are requested from and returned to the dynamic allocator at run time (i.e., `malloc()` and `free()`). Here, the maximal memory consumption of the system can be smaller than the sum of all allocations: if the lifetimes of two allocations do not overlap, the allocator can hand out the returned memory a second time. For example, if two mutual-exclusive tasks allocate and return 20 and 25 bytes, the worst-case memory demand is not 45 bytes but 25 bytes.

Part	Flash	RAM	Price
ATXMEGA32C3	32 KiB	4 KiB	3.21 USD
ATXMEGA64C3	64 KiB	4 KiB	3.96 USD
ATXMEGA128C3	128 KiB	8 KiB	4.08 USD
ATXMEGA192C3	192 KiB	16 KiB	4.94 USD
ATXMEGA256C3	256 KiB	16 KiB	4.93 USD
ATXMEGA384C3	384 KiB	32 KiB	6.06 USD

Table 6.1 – Market Prices of AVR ATXmega C3 Series. Price and resource comparison of different ATXmega 8-bit processors from the same series in the same packaging (64TQFP) with the same feature set. The prices were obtained on 20th February 2019 from `https://www.digikey.com` for a minimum purchase quantity of 1000 pieces.

One driver of the memory demand, especially of the static memory consumption, in event-triggered systems is the execution stack. First invented by Turing in 1946 [Car93],[9] the execution (or function call) stack allows the recursive invocation of subroutines. For this, the stack data structure provides `push()` and `pop()` primitives that store and retrieve data in last-in–first-out manner. In its simplest form, the execution stack holds return addresses, which a function *caller* pushes onto the stack before he invokes the subroutine. After the *callee* function finished, it pops the top-most return address from the stack and jumps to this program address.

function-call frame

In most run-time environments, the stack does not only hold the return address for each function invocation but also other short-lived data that are only required during the function execution (i.e., local variables, arguments). This data bundle is called the *function-call frame* or simply frame. After invocation, a function pushes a new frame onto the stack and eventually pops it before returning to the caller. Since all call-frame–data accesses are done relativly to the top–of-stack pointer, or another derived pointer (e.g., base pointer), multiple invocations of the same function can exist at the same time; allowing for recursion. Furthermore, with this dynamic allocation strategy, we do not have to allocate space statically for every function-local variable but only reserve it on the stack for the lifetime of the function instance.

On first sight, it seems counter-intuitive that I consider the stack space as static memory, as stack and heap are located, in the classical memory organization, on opposite ends of the address space, grow towards each other, and are able to share memory in between. However, in the context of a RTCS with several threads, we often require multiple stacks, whose start pointers have to be placed such that each stack provides enough space to hold its maximal memory demand without growing into other memory reservations. Therefore, it is crucial for the safe operation of an RTCS to give correct upper bounds for the stack demand.

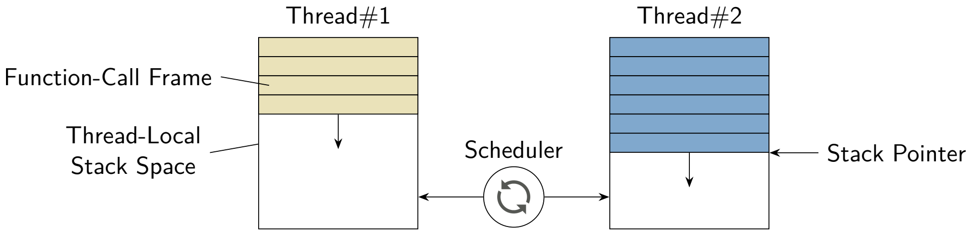


Figure 6.1 – Private Thread Stacks. If each thread has its own private stack space, where the function-call frames (colored boxes) are dynamically allocated from, the scheduler-induced context switch can happen at any time.

In the context of most operating systems, execution stacks are closely connected to threads, and often every thread is equipped with its own *private* stack. This allows the RTOS to suspend and resume every thread at every point in time, since every thread has its own top-of-stack pointer, and therefore the possibility to proceed its execution at function calls and returns. Since this continuability brings so much flexibility, even systems that initially favored a stack-less and event-driven programming model, like Contiki [DGV04] or TinyOS [Lev+05], were later extended with optional thread packages [Dun+06; Klu+09]. With thread-owned stacks (see Figure 6.1), the thread dispatcher saves and restore the list of currently active call frames by including the stack pointer

---

[9] The achievement of Turing was forgotten until the 1970s because the British government kept the ACE report secret. However, his proposal of the BURY and UNBURY instruction predates the proposal (1955) and the patent [BS57] of Bauer and Samelson, who proposed the stack principle for the evaluation of arithmetic expressions. In 1988, Bauer, who popularized the stack concept, was awarded with the IEEE Computer Pioneer Award “for computer stacks” and his other contributions, like coining the term “software engineering” [Awa88].

into the thread context. As the memory of a private stack is exclusively owned by one thread, we need no additional synchronization measures and the OS can switch, *without any restriction* and *at any time*, between threads and their corresponding stacks.

For static real-time systems, private execution stacks have a significant disadvantage: memory consumption. Since the thread allocates call frames from the stack spaces without help of the RTOS, we have to reserve enough space such that all simultaneously-existing call frames fit in and no *stack overflow* can occur. As the other RTOS structures are already allocated statically, it is nearby to allocate stack spaces also statically and dimension them for the thread's *worst-case stack consumption (WCSC)*. While excessive stack-space reservations were also identified as a problem in general-purpose operating systems, like Mach 3[Dra+91], the restricted nature of deeply embedded execution platforms (see Table 6.1) increases the urgency of the topic.

However, the regimented nature of RTCSs opens a possibility for stack-space reduction that is unknown in the desktop world: Not every thread can preempt every other thread at any given point in time, but the priorities, periods, and inter-thread dependencies constraint the possible preemption patterns. From these real-time properties, we can obtain knowledge about threads that never execute at the same time and associate the same stack space to both; allocating only the maximum (instead of the sum) statically. While preemption constraints point out the potential for *stack sharing*, we also require an efficient mechanism to distribute the same stack space to multiple threads without provoking memory corruptions.

stack sharing

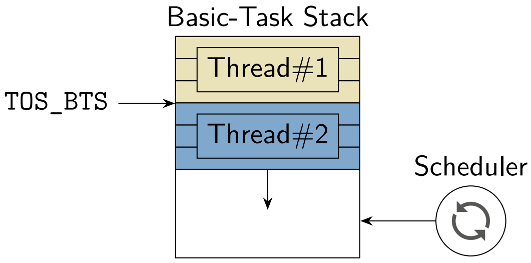


Figure 6.2 – Basic Task Stack. Since basic tasks cannot wait passively, they can only be forcefully preempted, and we can allocate them onto the same stack. Preempted threads (#1)that are closer to the bottom-of-stack will only resume after the preempting threads (#2) complete.

The OSEK world already provides such a stack-sharing mechanism that is tightly coupled with the real-time scheduling: basic tasks. In contrast to *extended tasks (ETs)*, *basic tasks (BTs)* (see Section 2.1.3) are not allowed to suspend themselves; they cannot wait passively for an event. Therefore, a started basic task executes until its self-termination and only the preemption by another higher-priority thread can pause this run to completion. This property allows us to co-locate all basic tasks on the same shared stack, the *basic-task stack* (BTS) (see Figure 6.2). There, the call frames of all basic tasks are tightly stacked onto each other and a newly started basic task begins its call-frame allocations where the last preempted basic task allocated its last call frame (`TOS_BTS`). Due to the no-waiting restriction and the priority structure, only the top-most thread is actually runnable; it (thread #2) must terminate, before the buried threads (thread #1) can continue.

basic-task stack

The BTS mechanism is closely connected to and only possible due to the prohibition of self-suspension in BTs. To get an intuition about this, let us assume the following counterexample situation in Figure 6.2: Thread #1 has a high priority, waits for an event and is currently not on the ready list. We dispatch thread #2, which has a low priority, on top of thread #1. If now thread #1 wakes up, it has the highest priority in the system and gets scheduled immediately. The resumed thread #1 allocates a new call frame, which will overflow into the thread-#2 stack area and provoke a memory corruption. In a nutshell, waking up a thread on the BTS must not happen if its frames are currently buried under another thread's frames. OSEK avoids such situations by generally forbidding `WaitEvent()` system calls in basic tasks.

However, this prohibition restricts the expressive freedom of the developer. Since only whole threads can be marked as BT and transferred to the BTS, we can either have the potential savings of the BTS mechanism *or* the possibility to wait passively for other threads or interrupts. This has an severe impact on the code organization of an application as I/O-driven code, like user-space device drivers, would have to be split into run-to-completion chunks that are activated by interrupts and synchronous system calls. Furthermore, the prohibition of passive waiting also hinders the composability and reusability of code, since we cannot call a library that waits at some point from a basic task.

With *semi-extended tasks (SETs)*, I provide a mechanism that allows for the fine-grained transfer of call frames onto the BTS. Thereby, threads can wait at some point passively on their own private stack, while other parts of its execution are relocated onto the BTS. Furthermore, I provide the necessary fine-grained WCSC analysis that uses the information from the interaction models to identify and ignore infeasible configurations on the shared stack, which leads to a tighter estimation of the memory consumption. Based on the SET mechanism and the WCSC analysis, I provide an optimization strategy to shrink the static stack-space allocations for private *and* shared stack(s).

6.2 Stack-Space Saving in the Literature

Before I describe the SET approach to stack sharing between blocking threads, I want to discuss other proposals that aimed for a reduction of the static stack-space reservation. Thereby, all proposals are based on the observation that the maximum of the system-wide call-frame allocations is often lower than the sum of the WCSC of each thread. In the following, I categorize the related work into three classes: (1) Better mechanisms for call-frame allocation. (2) Optimized real-time parameters to exploit the shared-stack approach. (3) More precise WCSC analysis in real-time systems.

call-frame allocation

Taking a step back, we can describe a statically-allocated stack space as a bump-pointer allocator, with the stack pointer as the bumping pointer. On every call-frame allocation, we move the stack pointer forward;[10] on every function return, we move it backward. While this kind of allocation is efficient and easy to implement, it requires a continuous area of (virtual) memory. For the first class of related work, other call-frame allocation strategies are proposed that co-locate frames of different threads to achieve a denser packing.

Early examples of non-linear stacks and heap-allocated frames can be found in LISP run-time environments [Ste77; BW73]. In order to provide continuations, these systems had to allocate call frames that outlived the call frame of their caller function. Since one caller can generate several continuations, which all link to the caller's frame, a tree-shaped stack structure, the so called *spaghetti stack*, arises.

For the MESA system [Lam82], call frames from different threads were first allocated from a common heap to provide for a compact storage. Each call frame is linked to its dynamic predecessor, forming a single-linked list of call frames for each thread. This mechanism was also applied to wireless sensor nodes [Yi+07], which suffer from strict memory constraints. However, for the memory-saving aspect, this per-call approach has the downside that the heap gets quickly fragmented as frame sizes vary and frames are often short lived [Lam82; Ste77].

Besides fragmentation, per-call mechanisms also impose a run-time overhead for every function call. Therefore, the idea to dynamically allocate larger chunks of memory, which can hold multiple

---

[10] On most architectures, the execution stack grows from the large addresses to the small address; from top to bottom.

frames, came up. Grunwald and Neves [GN96] statically analyze the function-call graph to find regions with a bounded stack consumption. A generated "allocation stub" requests the required memory from specialized fixed-size allocator at the region entrance. Thereby, no run-time check has to be performed within the region. Behren et al. [Beh+03] proposed a more reactive version of this principle, which also segmented the call graph into regions and calculated an upper limit for their stack consumption. However, instead of proactively allocating the exact amount of memory, they allocate larger chunks and the compiler inserts checks at every segment entrance that enough space is available in the current chunk.

A more hybrid approach is MTSS [MSB08], which was proposed particularly for embedded systems. Here, every thread starts in its private static stack-space and run-time checks detect possible stack overflows. If the reserved stack space becomes too tight, the thread steals space, in fix-sized quantities, from the unused memory reservation of other threads. Mauroner and Baunach [MB17] bring a similar reactive stack-space allocation to the hardware level. An OS-aware *memory-management unit (MMU)* provides a linear stack-space virtualization that transparently grows and shrinks the amount of used physical pages according to the task's stack pointer.

Compared to the SET mechanism, the mentioned dynamic call-frame–allocation mechanisms pay continuously for run-time checks and for the memory allocation. SET, on the other hand, transfers the control flow at neuralgic points unconditionally, often with a single instruction, onto the shared stack.

real-time parameters

The second class of related work acts in the RT domain as it adapts the real-time parameters to optimize the stack consumption while keeping the system schedulable. These proposals assume a basic-task–like model with non-blocking threads that run on a single shared stack.

Wang and Saksena [WS99] introduced *preemption-threshold scheduling (PTS)* as a modification to fixed-priority scheduling where each thread has two static priorities. With the preemption priority, a task preempts other tasks; With the preemption-threshold priority, which is higher than the preemption priority, a task defends itself against being preempted. A reschedule only happens if a readied task has a higher preemption priority than the current preemption threshold. By restricting preemptions, PTS results in more mutual-exclusive tasks, which can be exploited in the WCSC analysis. For example, all tasks that share the same threshold cannot preempt each other.

Furthermore, these thresholds do not only decrease the WCSC on the BTS, but they can, if chosen appropriately, increase the schedulability of the system [GD07]. For partitioned and global fixed-priority systems, Wang et al. [Wan+16] and Wang, Gu, and Zeng [WGZ16] proposed an ILP-based strategy for choosing optimal thresholds that minimize the WCSC but still result in schedulable systems. However, all PTS-based proposals view the task as the indivisible source of stack consumption.

A more fine-grained approach was taken by Baker [Bak91], who extended the *priority ceiling protocol (PCP)* [SRL90a] for resource acquisition to the *stack-based resource protocol (SRP)*. As already explained in Section 2.1.3 for OSEK resources, the priority of a resource-requesting task is immediately raised to the ceiling priority of the resource, which prevents any preemption by other tasks that can also acquire the resource. Thereby, we can safely use mutual-exclusive shared resources on the BTS as the `GetResource()` system call can never lead to a self-suspension. As a side effect, SRP locking also reduces the WCSC on the BTS as it restricts preemptions as long as a thread holds a lock. Since the SRP ceiling priority and the PTS thresholds work so similar, Gai, Lipari, and Di Natale [GLD01] could show that preemption thresholds are a special case of the SRP. We can reduce PTS onto SRP if all tasks with the same preemption threshold share an implicit resource, which the RTOS implicitly acquires and releases on thread dispatch and preemption.

Yao and Buttazzo [YB10] used this equivalence to implement preemption thresholds for multiple AUTOSAR runnables, which they mapped sequentially into the same thread body. For this, the assigned threshold was enforced by generated SRP requests that enclosed the runnable's code. Zeng, Di Natale, and Zhu [ZDZ14] used the same SRP-based mechanism and proposed a method to minimize the stack consumption by modifying the runnable-to-thread mapping. Both approaches consider the inner structure of the thread implementation, but only in a much simpler form than the complete call graph.

All [WS99; Bak91; Wan+16; WGZ16; YB10; ZDZ14] these proposals optimize the system by altering its real-time parameters. In contrast, I assume the real-time parameters to be fixed and only exploit the already present preemption constraints that are induced by these RT policies. However, unlike the RT-domain proposals, SET supports (partial) stack sharing among self-suspending tasks and considers the task's micro structure in terms of its call graph.

WCSC analysis

The third class of related work complements the constructive stack-space saving methods in the OS and RT domain by providing tight(er) WCSC analyses. With the WCSC analysis, we calculate an upper bound for combined size of the dynamic call-frame allocations on a given stack. If we dimension the stack space according to the WCSC, we statically ensure that no stack overflow can occur at run time. Therefore, tighter WCSC estimates that are closer to the actual WCSC directly translate into a reduced memory consumption of our system.

The commercial StackAnalyzer [Abs19] tool is developed and sold by AbsInt, and it calculates the WCSC for a regular program (i.e., no context switches). For this, it starts by analyzing the binary code before it derives an upper bound for the stack consumption of each function. In the extracted call-graph it searches for the worst-case path that leads to the WCSC for a given task. This per-task information can, for example, be combined compositionally into the WCSC on the BTS by the tooling that surrounds the commercial RTA-OSEK [Ltd07, cha. 18-2]. However, for both industrial tool chains, the cited sources are not precise enough to determine the employed method they use to determine the WCSC.

On the academic side, Hänninen et al. [Hän+06] presented a WCSC analysis for hybrid (time- and event-triggered) systems that use a shared stack. For this, they incorporated information about task timings (i.e., periods, activation offsets) to get an approximate but safe upper bound on the stack consumption. Later [Boh+08], they extended their approach with an exact analysis that uses a branch–and–bound analysis. However, both analysis consider the stack consumption only on the level of tasks as they neglect the function level.

For interrupt-driven programs, which only have IRQ-induced preemption and no RTOS-mediated interaction, several attempts were made to deduce a WCSC directly from the binary code. Brylow et.al [BDP01; Bry03] used a model-checking algorithm for push-down systems, where each explored system state consisted of the program counter and the interrupt mask of a Z86. Regehr, Reid, and Webb [RRW05a] proposed a similar method based on the abstract interpretation of program for an AVR 8-bit MCU, which can also handle dynamically computed interrupt masks and not only constant values. From a context-sensitive data-flow analysis, which calculates the interrupt mask for every point in the system, they construct an interrupt-preemption graph with the stack consumption at every preemption edge. Chatterjee et al. [Cha+03] showed that the exact stack-bound of interrupt-driven programs with arbitrary-deep nested ISRs and interrupt masks is PSPACE-hard and we can reduce QSAT onto the construction of the interrupt-preemption graph.

The WCSC analysis that accompanies the presented SET approach differs in several aspects from previous methods as it considers the stack consumption on the function-level but in the system-wide context of a full-featured RTOS model. Thereby, the analysis supports complex call graphs with recursive functions, as well as additional function-level preemption constraints. I extract these constraints about mutual exclusive call-frame allocations from the control-flow sensitive the interaction analysis.

6.3 Hybrid Thread Execution on Two Stacks

The basic idea behind *semi-extended tasks (SETs)* is a hybrid approach to call-frame allocation and stack-space reservation that allows a blocking thread to utilize the shared stack space. For all call frames that will never be on the stack when the thread enters the waiting state, we allocate frames on the BTS. For the other frames, when we are unsure whether the thread can sleep during their existence, we use a private stack-space reservation. By this hybrid allocation principle, a SET can never enter the waiting state if one of its call frames exists on the BTS. This avoids the situation where a blocking thread wakes up while one of its call frames is buried in the middle of the BTS.

For this hybrid allocation scheme, we have to decide for every frame whether it is *non-waiting* and can be placed on the BTS, or if it is *waiting* and has to be put on the private stack. It is clear that a function that issues a `WaitEvent()` system call itself must allocate its frame from the private reservoir. However, waiting can happen also indirectly, if a called function or any function deeper down in the calling hierarchy issues a `WaitEvent()`. Therefore, the call frame allocator must predict whether the function itself or any of its child functions will block. However, this prediction cannot be done precisely for an arbitrary function as it reduces to the halting problem.

Therefore, I use an over-approximation and decide statically, on base of the call graph, whether a given function can, directly or indirectly, issue a blocking system call. Thereby, I identify regions in the call graph that can never provoke a waiting state. For the functions in these regions, we can surely allocate their call frames from the BTS. Figure 6.3 shows this operation principle schematically.

The SET (#2) starts its execution on the private stack and allocates frames there until it enters a non-waiting call-graph region. Then we transfer the control-flow to the BTS and allocate frames there. In this state, SET #2 can be preempted by a basic task as this preemption can only be the result of BT #3 having a higher priority. As BT #3 is also not allowed to wait, it runs to its completion and removes its frames form the BTS before SET #2 is scheduled again. If the non-waiting region is left by the destruction of the last frame on the BTS, SET #2 returns its execution back to its private stack, where it is able to issue blocking system calls.

For this basic operation principle, we have to solve three problems that I will answer in the next three sections: (1) How can a thread switch between stacks efficiently? (2) What amount of memory must we reserve for the private and for the shared stack space? (3) Which functions should change the stack for the overall WCSC to become minimal?

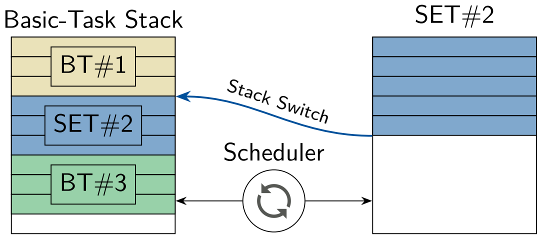


Figure 6.3 – Basic Principle of Semi-Extended Tasks. Each SET starts on its own private stack space, which it can use for waiting and waking. For those code sections that surely never wait, the SET transfers its execution onto the shared stack and draws its call-frame allocations from the common stack-space reservation.

6.3.1 A Mechanism for Intra-Thread Stack Switch

In the following, I will describe the stack-switch mechanism, as well as the required compiler and RTOS modifications that are necessary to implement semi-extended tasks efficiently. For understanding SET's light-weight stack-switch mechanism, we must take a closer look at the technical details of the call-frame allocation [Aho+07, cha. 7.2]. For this, I will discuss, exemplary, the frame layout of the System-V application binary interface [The97, cha. 3.9], which Linux programs use. While calling conventions and compilers might make different decisions, we do not lose generality as the used concepts are similar on most platforms.

Unlike heap allocations, the call-frame allocation is not done at one point in time, but the responsibilities are split between the caller and the called function Figure 6.4. First, the caller pushes the arguments on the stack (line 3-4) and invokes the call instruction to transfer the control flow to the called function. There, the function prologue performs the rest of the frame allocation: first, we save the `ebp` register as a callee-saved register (line 11) as we use it to hold the *frame pointer* for our call frame. From the *stack pointer*, which always points to the *top-of-stack* address, we derive the frame pointer (line 12) and use it in the function body to access the passed arguments. After the new frame pointer is in place, the function allocates slots for the local variables (line 13) and register-spill slots if necessary. In the epilogue, we use the frame pointer pop the callee-created stack frame with two instructions (line 20-21).

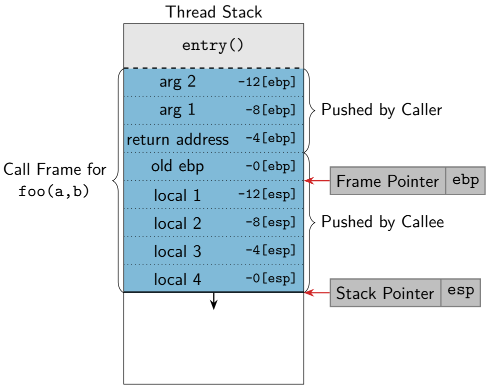
In the standard call frame, the frame pointer seems unnecessary as the frame is fixed in its size, and we can address every stack slot indirectly via the stack pointer with constant offsets. Therefore, compilers often provide the optimization to avoid the usage of a frame pointer,[11] which frees the `ebp` register and makes it available to the register allocation. However, functions that use dynamic stack allocations (i.e., `alloca()` or C99 variable-sized arrays) always require a frame pointer, since the distance between their parameters and their local variables is not statically known.

---

[11] GCC/CLang `-fomit-frame-pointer`

```
1  entry:
2     ...                  ; Push arguments in
3     push b               ;   reverse order for
4     push a               ;   variadic functions
5     call foo             ; Transfer control flow
6     add esp, 8           ; Cleanup arguments
7     ...
8
9  foo:
10    ;; Function - Prologue
11    push   ebp           ; Save old framepointer
12    mov    ebp, esp      ; Load new framepointer
13    sub    esp, 16       ; Allocate local variables
14
15    ;; Function Body
16    ;; - Parameters and local variables can be
17    ;;   accessed register-indirect via esp or ebp
18
19    ;; Function Epilogue
20    mov    esp, ebp      ; Restore old stackpointer
21    pop    ebp           ; Restore old framepointer
22    ret
```

(a) IA-32 Disassembly of `f1()`



(b) Stack Allocations

Figure 6.4 – Regular Intel IA-32 Call Frame. The call frame allocation and deallocation is split between caller and callee. The additional frame pointer allows for efficient stack unwinding and variable-sized arrays on the stack.

stack-switch functions

Coming back to SETs, we want to switch to the BTS in *stack-switch functions* that get statically marked by the system generator. For this, we modify the compiler to produce a slightly different function prologue (see Figure 6.5). Thereby, we split the call frame at the frame pointer into two parts and transfer everything between `ebp` and `esp` onto the BTS. Furthermore, the RTOS has to handle the situation if a SET gets preempted on the BTS.

In the function prologue, after the new frame pointer is in place, we overwrite the stack-pointer register with the value of the `TOS_BTS` variable (Figure 6.5, line 4). As we already discussed in Section 6.1, an OSEK-compatible RTOS already requires this internal `TOS_BTS` variable, which always points below the last preempted thread on the BTS. The RTOS uses it to indicate where newly started basic tasks should begin their frame allocations. For SETs, we have to modify the RTOS to update the `TOS_BTS` variable, if a SET gets preempted while it currently executes on the BTS. Furthermore, we expose the `TOS_BTS` variable as a read-only variable to the application such that it can be used in the switch instruction. Afterwards, all subsequent stack allocations (line 5), draw from the BTS space instead of the private stack; even the invoked child functions draw from the BTS without any further modification.

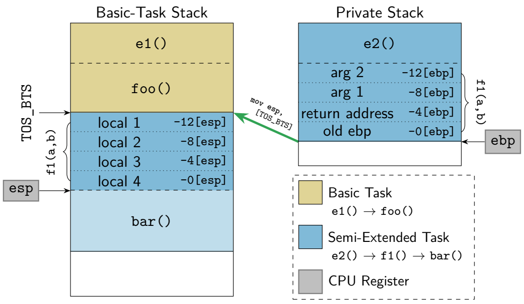
In addition to the single instruction in the function prologue, we also have to modify the compiler's address generation for parameters and function-local variables. As the stack frame is no longer a continuous memory area, we must enforce that the arguments are only addressed via the frame pointer and the local variables only via the stack pointer. Luckily, compilers already must support such constraints since some functions (i.e., with some SIMD instructions) require dynamic stack alignment, which results in gaps of unknown size below the frame pointer. Therefore, I just had to trigger this existing mechanism for switch functions.

The call-frame split at the frame pointer also gives us the benefit that the function epilogue (line 14-15) remains unchanged. With the restoration of the stack pointer from the frame-pointer register (line 14), we implicitly switch back to the private stack. As switching back and forth is done with a single instruction, we do not introduce concurrency problems.

With these modifications, we can switch, at given functions, from the private stack space to the BTS. However, the presented implementation has the drawback of being not reentrant: Let us assume that a SET is currently running on the BTS. In this situation, `TOS_BTS` points below the frames of the last *preempted* thread, which is buried beneath the running SET. If the SET invokes

```
 1 ;; Function - Prologue
 2 push    ebp              ; Save old framepointer
 3 mov     ebp, esp         ; Load new framepointer
 4 mov     esp, [TOS_BTS] ; Switch to shared stack
 5 sub     esp, 16          ; Allocate local variables
 6
 7 ;; Function Body
 8 ;; - Access local variables via esp
 9 ;; - Access parameters via ebp
10 ;; - Access local variable-sized arrays (alloca)
11 ;;   via a third stack pointer esi, if needed
12
13 ;; Function Epilogue
14 mov     esp, ebp         ; Restore old stackpointer
15 pop     ebp              ; Restore old framepointer
16 ret
```

(a) IA-32 Disassembly of `f1()`



(b) Stack Allocations

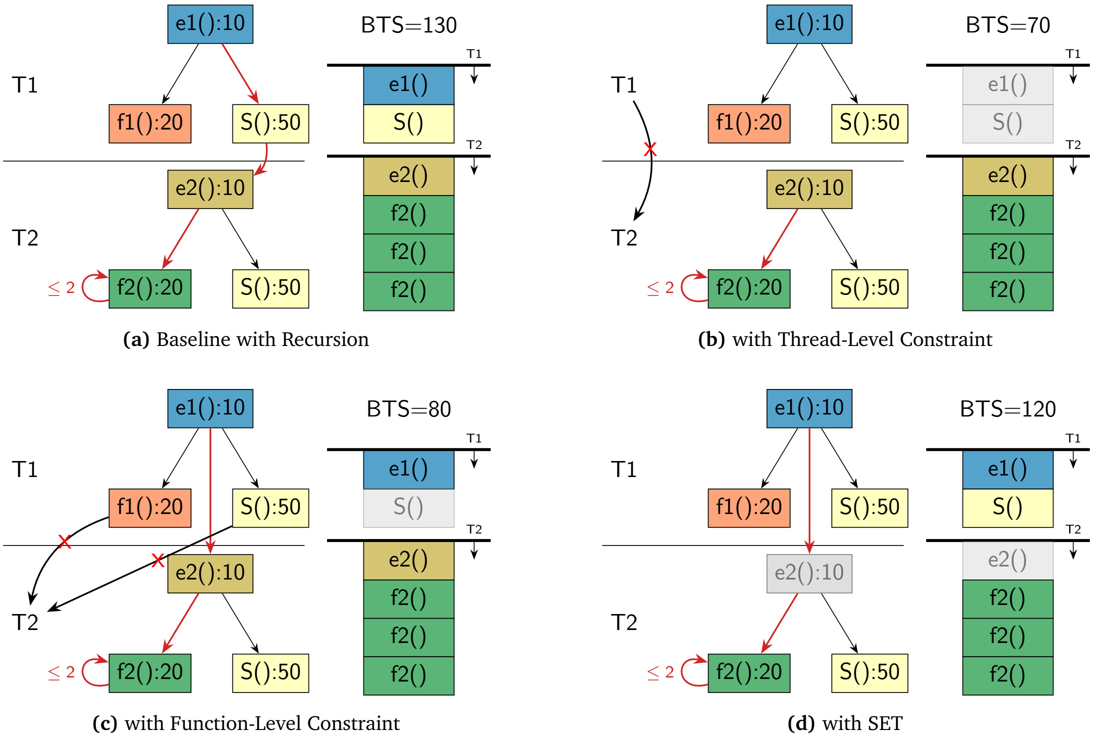
Figure 6.5 – Intra-Thread Stack Switch Mechanism. `f1()` switches from the private stack to the shared stack by loading `TOS_BTS` into `esp`. The stack diagram shows a situation where the SET preempted the basic task and switched with `f1()` to the BTS. Adapted from [DL18].

another switch function, while already being on the BTS, the stack pointer is set to `TOS_BTS` and the SET overrides its own active frames. Later, I will discuss the benefits and problems of a more flexible stack-switch mechanism.

6.3.2 Fine-Grained Worst-Case Stack Consumption Analysis

The co-location of multiple threads on the same stack does not necessarily decrease the combined stack-space demand. For example, if we execute a set of fully independent sporadic tasks, every thread can become ready at any given point in time. Without further knowledge, we have to assume that the worst case manifests: a preemption chain from the lowest- to the highest-priority thread where every thread gets preempted in its most stack-intense call chain. Thereby, the stack must be as large as the sum of the maximal consumption of each thread.

The only chance for shrinking the shared stack-space allocation is to find mutual-exclusive call frames that can never exist at the same time, and account only for the larger one. For finding a safe upper bound, we have to extract static constraints of mutual exclusiveness. Constraint sources can be flow-insensitive scheduling mechanisms like *preemption-threshold scheduling (PTS)* or flow-sensitive directed dependencies, like they stem from a thread activation that happens after a function call. But, wherever the constraints stem from and how we extract them, we have to use a WCSC analysis that can use them to give tighter bounds.



(a) Baseline with Recursion

(b) with Thread-Level Constraint

(c) with Function-Level Constraint

(d) with SET

Figure 6.6 – Stack Consumption and Preemption Constraints. The baseline WCSC can only become smaller if additional preemption constraints are known and used in the analysis. Each function in the call graph is annotated with its maximal stack demand and the worst-case preemption and call chain is indicated in red. Adapted from [DL18].

For understanding the influence of such constraints, Figure 6.6 shows the BTS consumption for the same system of two threads under different preemption constraints. While both threads have their own call graph, they both invoke the shared function `S()`, which can result in two active call frames for `S()`, one in each thread context. Furthermore, T2 calls the recursive function `f2()` for which we know the maximal recursion depth. Without further constraints (Figure 6.6a), the worst case manifests if we call `S()` in T1 and invoke the recursive function in T2 and we have to allocate 130 bytes of BTS space.

preemption constraints

If we know that T1 can never be preempted by T2 (Figure 6.6b), their stack consumption become mutual exclusive and we end up with a much smaller WCSC of 70 bytes. However, such preemption constraint can not only arise on the thread–thread level but also on the function–thread level. In Figure 6.6c, we only know that T2 can preempt T1 in `e1()` but never in `S()` or `f1()`; a constraint that can arise from the usage of SRP locks. Later on, I will explain how we extract such fine-grained constraints from the GCFG, which includes all preemptions edges between threads on the granularity of ABBs.

SET

Since I propose the usage of SETs to reduce the stack consumption, we also want to consider threads that do not start on the BTS but only switch there later on. In Figure 6.6d, we see the situation if T2 becomes a SET (switch functions: `f2()`, `S()`) and we have no further knowledge about preemption constraints. In this scenario `e2()` executes on the T2-private stack and, therefore, does not consume space on the shared stack, which decreases the BTS size to 120 bytes.

From the examples, for our WCSC analysis to become general applicable and tight in its bounds, we must support: arbitrary call-graph structures (with recursion), thread-level preemption constraints, function-level preemption constraints, and information about the SET configuration. As no stack-analysis technique from the literature supports such a fine-grained set of constraints, I had to develop my own WCSC analysis.

For this, we can frame the WCSC problem as a problem of finding the longest path in a weighted two-layer graph: we put the thread's call-graphs with per-thread–duplicated function nodes into the nodes of a thread–thread preemption graph; each function node gets weighted by its maximal call-frame size. On this graph, we search for that thread-thread–preemption and function-function-call path that has the maximal stack demand. Similar to the SysWCET approach (Chapter 4), I use the IPET to formulate this WCSC problem as an ILP.

We start out by introducing a count variable $\|T_i\|$ for each possible thread in the BTS. Since even the multiple activation support of OSEK allows only for one active task incarnation, the range of these variables can be restricted to $[0, 1]$. Furthermore, instead of explicitly modeling preemption edges between threads, I only implicitly constraint thread activations and preemptions implicitly. For example, if the threads $\{T_l, \dots, T_m\}$ have the same preemption threshold, we restrict the sum of their count variables to be no larger than 1:

$$\sum_{i=l}^{m} \|T_i\| \leq 1$$

For each thread, we collect all functions F_i that are reachable from thread's T_i entry function f_e. In $F_{L,i}$, we record all leaf functions that invoke no further function. For each function in the call graph, we introduce an integer-typed count variable $\|f\|^{T_i}$, scoped and prefixed with the surrounding thread. With the scoping, we virtually duplicate all call-graph nodes for every thread. For each caller-callee relationship in the call-graph, we add a call-edge count variable $\|f_s \rightarrow f_t\|^{T_i}$ and for the entry function an artificial call edge $\|E \rightarrow f_e\|^{T_i}$. With these variables and the IPET, we add structural ILP constraint to ensure that only valid paths through the call graph are allowed. Each function is called as often as the incoming call edges are taken (except for the entry function):

$$\begin{aligned} ||f_e||^{T_i} &= \sum_{f_s \in \text{caller}(f)} ||f_s \to f_e||^{T_i} \quad + ||E \to f||^{T_i} \\ \forall f \in F_i \setminus \{f_e\} : \quad ||f||^{T_i} &= \sum_{f_s \in \text{caller}(f)} ||f_s \to f||^{T_i} \end{aligned}$$

Since we search for *one* path through the call graph, every function invocation can lead to at most one additional invocation. However, since constraints could forbid the invocation of leaf functions we have to use a less-equal constraint here to avoid situations where the costliest sub-graph is not considered because its leaf-functions are forbidden. This would lead to an underestimation of the stack consumption. Therefore, the outgoing edges can, in sum, be visited at most as often as the calling function:

$$\forall f \in F_i \setminus F_{L,i} : \sum_{f_t \in \text{callee}(f)} ||f \to f_t||^{T_i} \leq ||f||^{T_i}$$

To complete the structural constraints, we have to add constraints about recursion limits like we would do it for a WCET ILP. These recursion limits have to be supplied by the developer. At last, we complete the basic ILP structure by adding the stack usage as costs to the function variables and formulate a maximization objective.

$$\max \left(\sum_{T_i} \sum_{f \in F_i} ||f||^{T_i} \cdot \text{stackusage}(f) \right)$$

Hereby, I assume that a maximal stack usage is given for each function. This information can often be supplied by the compiler, who anyway defines a function's stack layout. The only difficulty arises from functions with dynamic stack allocations, which might lead from the use of variable-sized stack-allocated arrays or `alloca()`. For these functions, we use the maximal possible stack consumption, which has to be given by the developer if we cannot deduce it with another tool (e.g., AbsInt StackAnalyzer [Abs19]).

We also handle the influence of SETs in the objective function: Since we want to give the WCSC for a system where we have already decided on the switch functions, we can surely identify, with a search from the thread entry to the switch functions, all functions that will never contribute to the shared stack consumption. For these functions, which surely allocate their frames from the private stack, we set the `stackusage(f)` to zero. Thereby, these functions can still play their role in constraining the preemption and call chain, but they do not contribute to the WCSC.

On the so generated ILP, we add further constraints that capture the coarse- and fine-grained preemption information. This information can stem directly from the real-time parameters (e.g., preemption thresholds), but we can also extract preemption information from the interaction analysis. Since we are only interested in the possible preemptions on the thread–thread and function–thread level, the less fine-grained GCFG interaction model and the less costly (and polynomial) SSF analysis (Section 3.5.2) is sufficient.

For extracting preemption constraints from the interaction model, we comprehend the GCFG nodes as triples of thread, function, and executed ABB: (T_i, f, ABB). For each possible (T_i, f) pair, we perform a depth-first search on the GCFG to find the set of all threads that cannot be reached by preemption in f. For this non-preemption set, we start with a full set of all threads and continuously thin out all threads that are reachable on non-resume GCFG edges. For this, we remove all visited T_x from the thread set until we stop the search at nodes that resume back to T_i:

$$(T_i, f, *) \quad \overbrace{\rightarrow (T_x, *, *)}^{0\ldots*} \quad \rightarrow (T_j, *, *) \qquad T_x \neq T_i, T_j \neq T_i$$

By this search, we do not only find all direct preemptions but also all indirect preemptions where a third thread T_j can only become active after some intermediate thread T_x was the first preemptor of T_i. From the resulting non-preemption sets, we can formulate different preemption constraints: If a thread T_j is in all non-preemption sets $(T_i, *)$ of a thread, then we have found a coarse-grained preemption constraint that forbids all preemptions from T_j to T_i. If the thread T_j is only in some non-preemption sets, then we have found a fine-grained preemption constraint, where a thread can only be preempted in some functions.

Since the GCFG contains the complete interaction between application and RTOS scheduling, we cover the influence of real-time parameters like the usage of implicit SRP resource, preemption thresholds, and non-preemptable threads. However, we also get constraints from code-level constructs like explicit SRP resource allocations, interrupt blockades, or the sequential activation of threads. Thereby, we consider the directed and undirected dependencies from the RT domain in exactly that fashion as they influence the actual implementation.

From the GCFG, we get constraints about impossible preemptions ($f \rightarrow T_j$, $T_i \rightarrow T_j$) that we want to introduce into the WCSC ILP in order to tighten up the stack-consumption estimation. On a higher level, we want to force the $\|T_j\|$ variable to become zero if $\|f\|$ (resp. $\|T_i\|$) is larger than zero. We can accomplish this by using the big-M-method [HL01] and a binary-typed helper variable H_x:

$$H_x \in [0, 1] \qquad \|f\|^{T_i} \leq H_x \cdot M \qquad \|T_j\| \leq (1 - H_x)$$

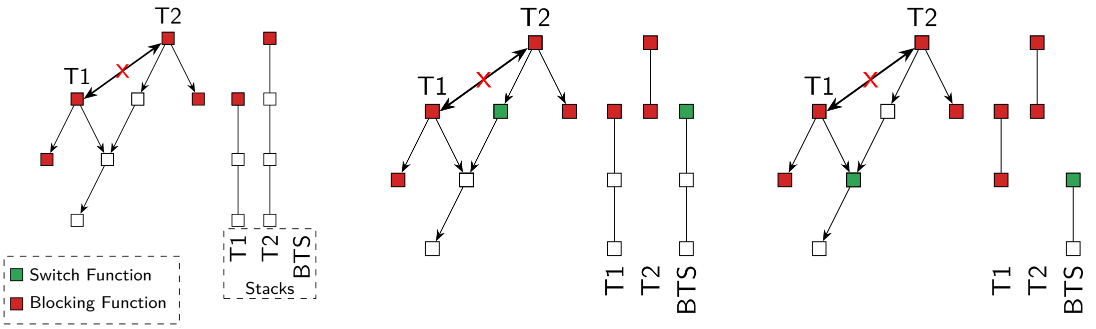
Thereby, M is a “sufficiently” large constant that we use to derive a binary value from $\|f\|^{T_i}$: if f is part of the costliest chain, H_x cannot be zero anymore but must become 1, while the constant M must be larger than any possible value of $\|f\|^{T_i}$. If H_x becomes 1, then the right side of the last constraint becomes zero and $\|T_j\|$ is also forced to zero. If $\|f\|$ is zero, then the third constraint is disabled.

Given to an ILP solver, the constructed problem formulation will yield the WCSC on the shared basic-task stack and the execution counts for all functions and threads on the costliest preemption- and call chains. However, for an honest evaluation of SETs we have to give upper bounds for the combined stack-space allocation of all threads; shared stack as well as private stacks. For this, we use the described ILP construction for each stack space in the system, solve their objectives individually, and add them up to get the total stack-space allocation in the system. We must consider each stack in isolation as the worst-case chains on different stacks are not correlated in time. Therefore, this compositional approach does not suffer the problems of the traditional WCRT analysis that I discussed in Chapter 4.

With the described ILP construction method, we can derive a WCSC from multiple call graphs, preemption constraints, and the set of SET switch function. While call-graph and preemption constraints are given and fixed in my system model, the selection of switch functions is variable.

6.3.3 Selection of Stack-Switch Functions

In the construction of the ILP problem, we have seen that constraints from different sources (call graphs, preemptions, switch functions) influence the WCSC. Due to the interplay of these constraints, an increased co-allocation of switch functions on the BTS does not necessarily lead to an increased



(a) Baseline **(b)** Harmful Selection **(c)** Beneficial Selection

Figure 6.7 – Switch-Function Selection Problem. Call graphs and stack consumption for two different switch-function selections in a system with two threads. Both threads are mutual exclusive and have connected call graphs.

BTS size as mutual exclusivity leads to stack-space reuse. Furthermore, since switch functions transfer their children's frame allocations to the shared stack, we can "hide" whole subgraphs of the call-graph in the shadow of other BTS users. As these subgraphs can no longer consume space on their private stacks, the private WCSC can become smaller. If we want to reduce the system-wide stack consumption, the shrinkage of the private stacks must be larger than the growth of the BTS space.

However, not all switch-functions selections lead to a decreased stack consumption in the system. Even worse, some selections even lead to an increased consumption compared to the unmodified baseline system. Figure 6.7 shows an example of this problem with a beneficial and a harmful switch-function selection. The baseline system contains two mutual-exclusive threads (T1, T2) with their call graphs, where each box is a function and has (for simplicity) the same stack consumption. Both call graphs overlap partially and wait in the most left and the most right child function, which prohibits the usage of OSEK's basic-task technique. In Figure 6.7a, we see also the worst-case call chain for the private and the shared stack. With this system configuration that solely uses private stacks, the system has a total requirement for stack space of 7.

If we use the switch-function selection from Figure 6.7b, we transfer the largest part of T2's call graph onto the BTS, which grows to 3 units. The stack configuration of T1 is not influenced as no switch function is part of its reachable call graph.[12] However, since the blocking functions of T2 still have to be executed on the private stack, it does not shrink to same extend as the BTS grows. Thereby, the overall stack consumption grows to 8, which is worse than the baseline system.

If we push down the switch function by one edge (Figure 6.7c), we get a different picture: Now both threads transfer 2 functions onto the BTS, while a maximal stacking depth of 2 remains on both private stacks. However, as the threads are mutual exclusive, we only have to allocate the BTS for holding two functions. So, in total, we have to use 6 units for the system's stacks.

From the example, we see that it can be better to transfer smaller subgraphs onto the BTS than to transfer the largest possible subgraph; the greedy variant does not produce the optimal solution. While this non-monotonity is already challenging for the WCSC optimization, things get worse as the call graphs are not necessarily trees (as in the example) and preemption constraints also influence the

---

[12] Please note that we cannot simply mark the child of the switch function also as a switch function, since this would break the proposed stack-switch mechanism (see Section 6.3.1). Nevertheless, if we had the possibility to mark parent and child function as switch functions, the stack consumption would remain the same.

WCSC on the BTS. Therefore, it would be desirable to formulate also the switch-selection function inside the ILP problem in order to let the solver do the heavy lifting.

Unfortunately, integrating the switch-function selection into the IPET-generated ILP is not easily possible. The WCSC problem is a maximization problem, where the solver can decide on invocation counts to maximize the overall stack consumption. However, there are many invocation-count vectors that also satisfy our constraints; namely, all other valid stack configurations. These solutions are not chosen, because they do not result in the WCSC but just in a smaller stack consumption.

In comparison, the switch-function selection is a minimization problem. A fictive ILP formulation of this problem would have one binary variable to indicate whether a function is a switch function and would derive a stack consumption as optimization objective from it. Thereby, again, many assignments to these binary variables would be valid solutions but only one would be minimal.

Combining both, WCSC and switch-function selection, our problem would demand from the ILP solver to select those invocation counts that maximize the stack consumption and those switch functions that minimize the stack consumption; an impossible task if no hierarchy between both problems are stated. We would like to achieve this hierarchy by embedding the WCSC problem as *fitness function* into the switch-function–selection problem. However, this results in a bilevel optimization problem [CMS07] which cannot be expressed, in general, as ILP.

genetic algorithm

Therefore, I decided to use a *genetic algorithm* [Rec73] to find switch-function vectors with a small worst-case stack consumption. Thereby, the genetic algorithm will not find the true optimum, as it is only a search heuristic, but we will still find solutions that are beneficial for the memory consumption of the system.

In general, a genetic algorithm works on improving a fixed-sized *population* of possible solutions. These solutions are represented by their *genomes* which are encoded as binary vectors. We can score and rank the *individuals* of the population with the *fitness function*. The genetic algorithm starts with a randomly generated set of initial individuals and tries to improve the population iteratively in *generations*. In each generation, new individuals are breed from one or more old individuals and ranked with the fitness function. From the *intermediate population*, we drop the worst individuals until we are back to our fixed population size. At some point, we abort the genetic algorithm and return the best solution. Please be aware that this brief summary of genetic algorithms is far from complete as many strategies for breeding, selecting individuals, and choosing search parameters were proposed [Whi94].

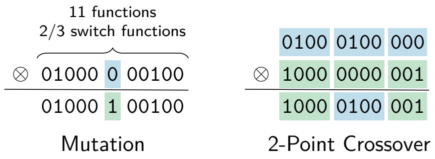


Figure 6.8 – Genetic Operators for Switch-Function Selection.

For the switch-function–selection problem, we have to choose the genetic representation of a solution, as well as to decide on breeding operators. Given a RTCS and all necessary call graphs, we use a genome with one bit for every non-blocking function in the system (see Figure 6.8). If the bit is one, the corresponding function becomes a switch function. For the breeding, we choose (randomly) from two classical operators: *mutation* (p=0.05) and *2-point crossover* (p=0.95). For mutation, we randomly select one old individual, copy it, and flip one random bit in its genome. For the crossover, we randomly select two individuals and randomly choose a start and a stop index in the vector. The new individual is a copy of the first individual where we replace all bits between start and stop index with the corresponding bits from the second individual. After the crossover, we additionally mutate the new individual with a low probability (p=0.1).

However, the newly bred individual is not necessarily a valid selection of switch functions as children of switch functions are not allowed to wait. Therefore, we check this condition on newly generated individuals. Invalid individuals are put in a second population (n=4), which is only used for breeding, in order to allow for larger mutations. The first population of valid individuals is kept at a size of 20, while we breed, in every generation, as long as 6 new valid individuals are generated. To avoid unnecessary recalculations, the algorithm keeps a cache of genome–WCSC pairs. As we have no idea about the lower bound of the WCSC, the optimization process is halted if no improved individual was found for 1000 generations or 60 seconds, whatever comes first.

Please note that I selected the search parameters manually set by trial-and-error with a few example systems, and they worked out well enough in the following evaluation.

6.4 Experimental Evaluation

For the experimental evaluation of the SET concept, I will compare the overall stack consumption for systems that support only private stacks, only basic tasks, or the combination of basic and semi-extended tasks. For this, I generate over 14000 synthetic benchmarks and feed them to the dOSEK framework, which optimizes the switch-function selection and analyses the overall WCSC of the system. Thereby, I use preemption information from SSF-generated GCFGs (Section 3.5.2) in the WCSC analysis, which demonstrates the scalability of this analysis for a wide range of systems.

6.4.1 Benchmark Generation

For the evaluation, I use synthetically generated benchmarks to investigate on the sensitivity of the SET approach with respect to changes in different system parameters. I deliberately chose to use synthetic benchmarks that can cover a wide range of parameters instead of individual realistic benchmarks (i.e., *i4*Copter), since the stack consumption is highly application specific and small changes in the code can manipulate the stack-usage characteristics in both directions. Therefore, synthetic benchmarks provide a more solid evaluation base for the SET method.

For the synthetic benchmarks, I had to write my own benchmark generator as available generators for real-time systems where either focused on the RT domain [BB05], provided only implementations for individual tasks [Wäg+17], or were developed with WCET and scheduling analysis in mind [BB05; Wäg+17; Eic+18]. However, my benchmark generation focuses on producing systems with thread interaction (no independent task set), different call-graph shapes (more than a tree structure), and the usage of RTOS services for synchronization and signaling, while the exact timing of the system is irrelevant.

benchmark generation

I identified five important system parameters that will characterize the generated systems (Table 6.2): Each system has #IRQ sporadic tasks, which are implemented by one or more threads and activated by an ISR. In total, the system contains #threads independently scheduled threads, of which #waiting perform a self-suspending system call at some point in their execution. The whole system contains #resources SRP resources and #functions functions with varying call-frame size. In the following, I will explain the system generation in greater detail.

We initialize a pseudo-number generator with a varying seed, which allows us to get different but reproducible systems. First, we generate a directed acyclic thread dependency graph: We generate #thread threads and give each thread a different static priority, since I target OSEK BCC1/ECC1 (see Section 2.1.3) systems. Additionally, 10 percent of all threads are marked as non preemptable.

Dimension	Description	Range	Stepsize
#threads	RTOS Threads	[20, 50]	5
#IRQs	External, asynchronous thread activations	[1, 10]	1
#waiting	Number/Ratio of blocking threads	[0, 15]	1
#functions	Number of functions in the call graph	[100, 1000]	100
#resources	SRP resource groups with > 2 threads	[1, 10]	1

Table 6.2 – Dimensions of the Generated Benchmarks

The thread set gets partitioned into #IRQ groups, which will be activated by the same external sporadic event by means of executing an ISR. For each task, we select one thread as the root of a randomly-generated dependency tree of the task's threads. Over all threads, we select #waiting threads to have a second dependency on another thread (without considering task affiliations). Later these second dependencies will become `WaitEvent()` system calls (see Figure 2.3b). Furthermore, #resources thread groups with at least 2 members are selected; Each group will synchronize at one point with an SRP resource.

After the thread–thread structure is in place, we generate the call-graph structure. For this, we randomly grow a forest with #thread roots and add #function nodes with a stack consumption that varies uniformly between 90 and 120 bytes. We also allow for more sharing of functions between threads by adding 20 cross-tree call edges. Thereby, the call-forest can become a tree, but we avoid the introduction of recursive functions as the SSF analysis does currently not support recursive functions.

For the function bodies, we first assemble a list of system services and subfunctions that have to be called by a function. From the task dependencies, we derive `ActivateTask()`, `SetEvent()`, and `WaitEvent()` system calls. Furthermore, we add pairs of `GetResource()`–`ReleaseResource()` sections according to the SRP groups and, with 10 percent chance, a thread blocks an IRQ at some point in its execution. The system calls are distributed into the first four functions from each thread's entry. This restriction stems from the current implementation of the SSF analysis, which forbids the sharing of system-relevant functions between threads. Nevertheless, as I want to ensure a minimal call-graph complexity per thread, I use the worst function/thread ratio (200/50) as the number of functions that invoke system calls per thread.

After the system is generated, I serialize the system-object parameters in an OIL. The functions are encoded in a C source file, where each function is equipped with a `volatile char` array that holds its worst-case stack consumption. Therefore, the actual stack consumption of each function is slightly larger as the size that was chosen by the benchmark generator.

6.4.2 Evaluation Scenario and Method

For the benchmark scenario, I choose 49 parameter classes, where each class is identified by the tuple (#threads, #IRQS, #waiting, #functions, #resources). These parameter classes are derived, by scaling each of the five dimension individually, from the base class (20, 1, 10, 200, 1). Table 6.2 lists the dimensions, the explored ranges, and the step size for the scaling. For example, between the class (**20**, 1, 10, 200, 1) and (**50**, 1, 10, 200, 1), we explore the influence of an increasing number of threads in seven steps (20,25,...,50). For each class, I generate 300 systems (PRNG seed 0–299), which results in a total number of 14 700 systems.

The generated applications are translated to LLVM IR [LA04] and lowered to Intel IA-32 machine code. In this first round of code lowering, I extract the maximum size of the call frames, which LLVM lays out for each function. This information is, together with the rest of the static information about the system, fed to the dOSEK framework, which performs the interaction analysis. For the ILP solving, I use `gurobi` [Gur19] in version 8.0 and perform all analyses and the switch-selection optimization on an Intel i5-6400 quad-core system with 32 GiB of main memory. At all points, the main memory was sufficient to hold the necessary data and the optimization process was CPU bound.

To manifest the optimization result, I annotate the selected function in the IR as switch functions and add the symbol name of the `TOS_BTS` variable to the annotation. In the LLVM backend, the body for annotated functions is enriched with the switch-stack instruction and the code generator is instructed to use separated addressing schemes for parameters and local variables. For stack-switch implementation, I had to change 35 source code lines in LLVM 7.0.

In the following, I will investigate on the computation cost and the scalability of the SET-optimization process, which happens before run time, as well as the achieved stack-space savings in comparison to the state of the art.

6.4.3 Run-Time and Scalability of the Optimization

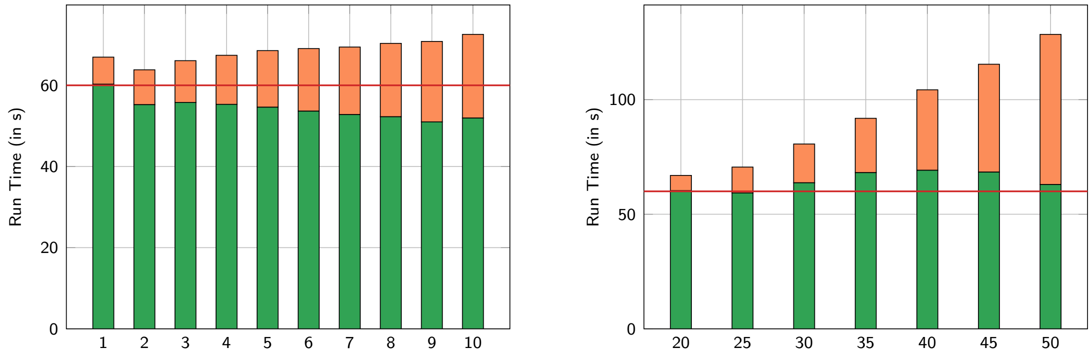
First, I want to investigate on the computation costs that we have to pay for the optimization process itself. These costs mainly consist of the run of the SSF analysis and the time required for the iterated solving of WCSC ILPs in the genetic algorithm. Since the analysis and optimization are done offline, the required computation time is not critical as long as it can be done in reasonable time if we compare to the development cycle of industrial embedded system. Moreover, as SETs are a non-functional optimization to the stack consumption, we could theoretically execute it only once for every deployed system configuration. In reality, the SET optimization should be enabled in the late stages of the product cycle.

Figure 6.9 show the run time of the SSF analysis and the genetic algorithm for each of the five parameter scaling dimensions. Each bar is arithmetic mean of the analysis run time of 300 generated systems. The optimization process gets aborted when the genetic algorithm makes no progress for 60 seconds or for 1000 generations. In order to show the influence of this timeout in the stacked bar plots, the genetic-algorithm time starts at zero and a red horizontal line indicates the 60-second mark. As the run time of the genetic algorithm is, in most cases, only slightly over 60 seconds, we see that the genetic algorithm quickly converges to an, at least local, optimal configuration.

The most influence on the run time of the genetic algorithm has the number of functions in our system (Figure 6.9c). It is the only dimension where the genetic algorithm exceeds the used 60 second-timeout significantly (165 s at 1000 functions). However, this effect is not surprising as the number of functions is directly related to the number of ILP variables which makes the solving of each ILP instance more expensive. For a single ILP invocation the run time increases from 0.009 seconds for 100 functions to 0.17 seconds for 1000 functions. In this interval, the solving time increases exponentially and doubles for every 384 additional functions.

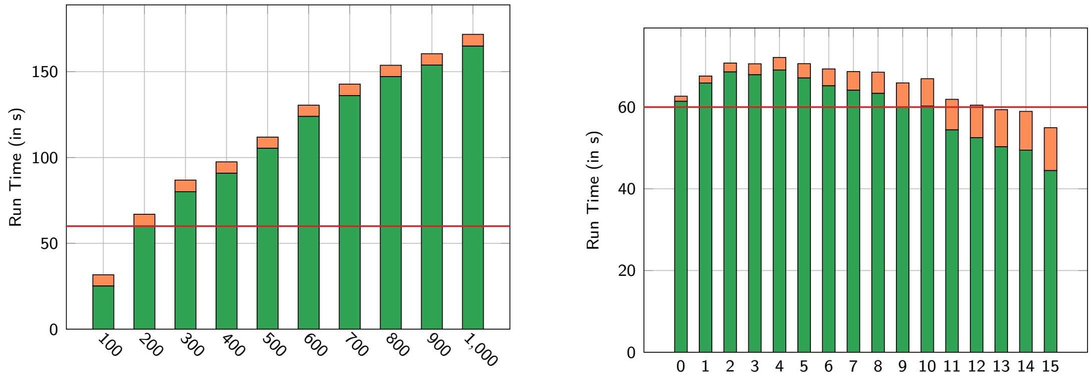
It is only due to the abort condition of the genetic algorithm that the overall run time does not increase with the same exponential factor. In this interval, the number of ILP invocations dropped from 2 719 (100 functions) to 1 128 (1000 functions), which indicates a less intense exploration of possible SET configurations.

For the interaction analysis, we see three factors that significantly increase the run time of the SSF analysis: interrupt sources (Figure 6.9a), threads (Figure 6.9b), and the ratio of self-suspending threads (Figure 6.9d). In all three cases, this increased analysis time stems from the larger state space of the system: For an increased number of threads, the number of system objects actually increases which results in more columns in the AbSS (Section 3.4.1). If more threads are self-suspending, more threads can not only be marked as ready or suspended, but they can also become waiting. If



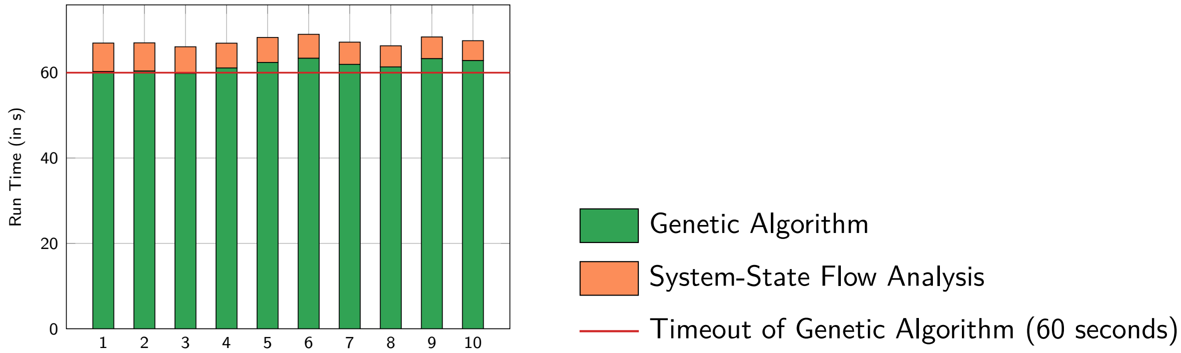
(a) Interrupt Sources (#irqs)

(b) Threads (#threads)



(c) Functions in Callgraph (#functions)

(d) Self-Suspending Threads (#waiting)



(e) SRP Resources (#resources)

(f) Legend

Figure 6.9 – Run-Time of Stack-Optimization Process. For each parameter class, these figures show the averaged (N=300) run times of the genetic algorithm and the SSF analysis. Thereby, the genetic algorithm stopped after 1000 generations of no progress *or* 60 seconds.

we increase the number of interrupts sources, more indeterminism is introduced into the system and more threads can be activated directly in computation block instead of synchronously from a system-call block. In all three cases, the SSTG, if we would calculate it, becomes larger, which results in more GCFG edges. As the SSF analysis performs a fixpoint data flow analysis on these edges, while discovering them on the go, the analysis takes longer.

Furthermore, if the number of GCFG edges increases, also the quality and the quantity of fine-grained preemption constraints decreases. We can already observe this influence in the run-time of the genetic algorithm: if the run-time of the SSF increases, the run-time of the genetic algorithm decreases as less preemption constraints have to be considered by the ILP solver. In the next section, we will see the influence of these diminishing preemption constraints also in the amount of stack saving and in the size of the switch-function set.

Apart from the overhead characteristics of the SET analyses, this evaluation does also demonstrate the scalability of the foundational polynomial SSF analysis. We see that the SSF analysis stays within reasonable bound if we scale important dimensions, like the number of threads or interrupts. At most, the SSF takes, on average, 66 seconds for the #threads axis (#threads=50). The longest SSF analysis of the whole evaluation for an individual system took 143 seconds and was located in the same (#threads=50) parameter class. Over the large and variational benchmark scenario, we see the scalability of the foundational polynomial SSF analysis as an efficient whole-system interaction analysis.

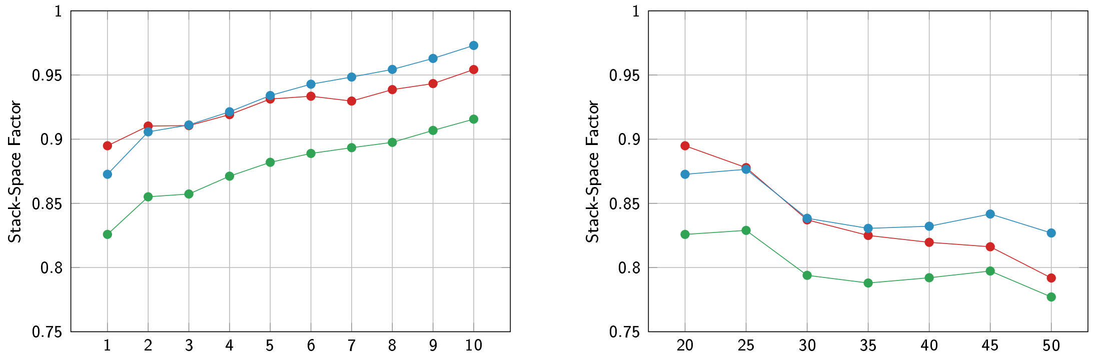
6.4.4 Comparison with Basic-Task Systems

Besides the feasibility for realistically-sized applications, the actual stack-space saving potential, as means to improve non-functional properties, must be quantified. For this, I apply different RTOS configurations (ET system, BTS system, SET system) to the generated benchmark systems and calculate and compare the system-wide WCSC. Thereby, I put the achieved savings into the context of the state of the art, as it is applied to industrial applications. Furthermore, I will dive into more detail of the structure of stack-switch–function sets as SET systems use the stack-switch mechanism alongside the basic-tasks mechanism.

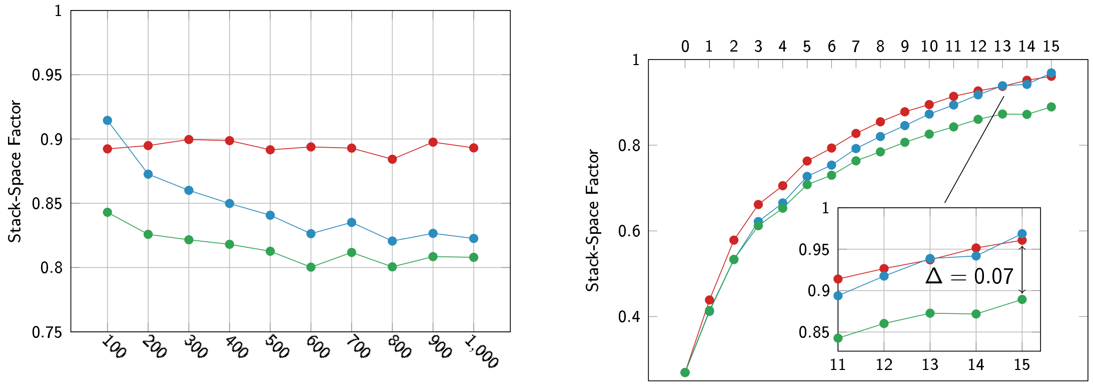
ET system: As baseline variant, I use an RTOS model that supports only private stacks such that no stack-space sharing happens between threads. Only the longest call chain is responsible for the consumption on each stack and all stacks are independent of each other. This configuration reflects the situation for many event-driven real-time systems, since stack sharing requires some restrictions, like the no-blocking rule, on the application code.

BTS systems: The second variant supports the transfer of whole threads onto the shared stack if the thread is not self-suspending. For this, I greedily (without the genetic algorithm) select all threads that never invoke a `WaitEvent()` in their call graph as basic tasks. This second RTOS configuration resembles the mechanisms in OSEK and can therefore be considered the state of the art when it comes to saving stack space in industrial applications. Actually, the greedy marking of tasks as basic tasks results in the optimal system configuration as the stack consumption can only benefit from basic tasks; they do not exhibit the non-monotonic anomaly of SETs (see Figure 6.7).

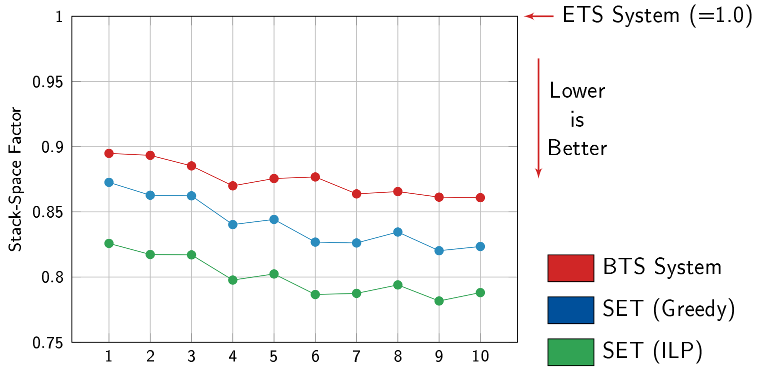
SET system (ILP): The variant is an extension of the BTS-system variant and uses the SET mechanism where it is beneficial for the system-wide stack consumption. The set of stack-switch functions is calculated according to the presented genetic algorithm (Section 6.3.3). Thereby, we still use the basic-task mechanism if it yields a lower stack consumption than switching stacks during the thread execution. For this, we mark a thread as a basic task if the optimization selects the thread's entry function as a switch function. Due to this fallback to the BTS configuration, these systems always show an improvement over pure BTS system when it comes to stack-space allocations.



(a) Interrupt Sources (#irqs) **(b)** Threads (#threads)



(c) Functions in Callgraph (#functions) **(d)** Self-Suspending Threads (#waiting)



(e) SRP Resources (#resources)

Figure 6.10 – Stack-Saving Factors. Comparison of the system-wide stack use for the generated benchmark system. The ET system acts as a baseline for the stack-saving factor calculation. Each dot in the figure represents the average over the factors of 300 systems for systems with only basic tasks (BTS system) and a mix of basic tasks and semi-extended tasks (SET system).

SET system (Greedy): In order to quantify the benefits of switch-set optimization, this variant greedily transfers all functions that can be transferred onto the shared stack. Furthermore, this variant is not constrained by a switch mechanism and switch functions can also execute on the BTS. For this variant, I have *not* implemented the necessary compiler and RTOS modifications, but I show the WCSC results only as a reference point.

For these variants, the WCSC is calculated according to Section 6.3.2 using the same set of fine-grained preemption constraints for each system. Please note that this usage of my own WCSC analysis improves the results for the BTS system, since the interaction-analysis constraints can be exploited. In fact, as I will show at the end of this section, *all* BTS-system savings in this evaluation were only possible due to the exploitation of fine-grained preemption constraints. In the following, I will discuss the savings that can be achieved by the BTS-system and the SET-systems if we compare them with ET systems as the baseline competitor.

The results of the WCSC analysis for ET, BTS, and SET systems are shown in Figure 6.10. For each generated benchmark system, I take the ET system as the baseline and give *stack-space factors* relative to this baseline (lower is better). Thereby, I use the geometric mean over the factors, which are calculated for each generated system individually, for calculating the average (n=300) for each parameter class [FW86]. Here, the geometric mean over the factors avoids the benchmark pitfall that not all systems have the same WCSC in the baseline variant.

Furthermore, for SET systems, Figure 6.11 shows the structure of the stack-switch-function set that is used for the SET systems (ILP). Here, I show (arithmetic) average for each parameter class, which will help us to understand from which mechanism (BT or stack switch) the savings arise and how the different system parameters influence them.

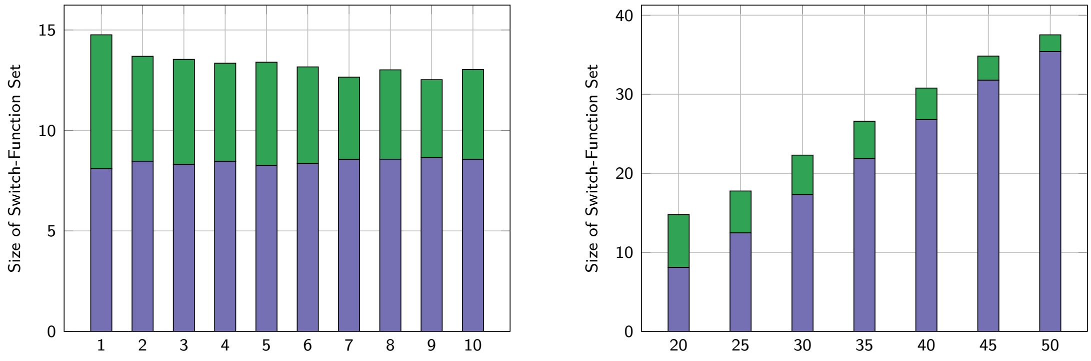
The first observation is that the greedy variant of the SET systems is consistently worse than the variant where the genetic optimization is performed. Due to the non-monotonic SET anaomaly, the results are, for some systems, even worse than for the BTS system. Only for 502 systems, the greedy variant, which is not constrained in its switch-function set, outperforms the optimized SET system (ILP) with its highly-efficient but constrained switch mechanism. Therefore, I will only consider the SET system (ILP) variant in the following.

As expected, the SET savings outperform the BT savings as the first is only an extension of the latter one. Over all systems (n=14 700), the SET systems has a stack-space factor of 0.78, while BTS systems only achieve a factor of 0.83. In 80 percent of all synthetic systems, the SET system had a lower stack consumption than the BTS system.

For the parameter classes, we see the general trend that more IRQs (Figure 6.10a) and more self-suspending threads (Figure 6.10d) as the only parameters that impair the stack saving negatively (for both variants). For IRQs, we have already discussed in the last section that the decreased determinism of additional interrupt sources results in more GCFG edges that originate from computation blocks, which reduces the size of the preemption-constraint set. Therefore, more interrupts harm both BTS system and SET system, while SET systems have a factor that is on average 4.72 percent points lower.

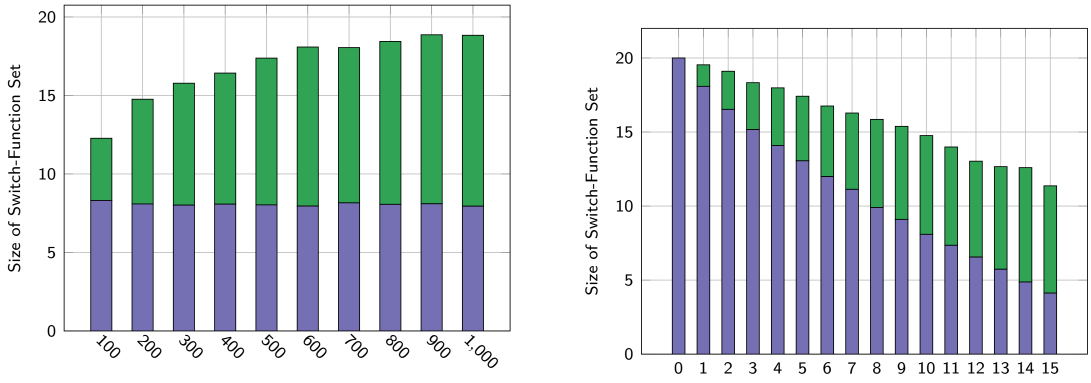
For the #waiting class, the increased number of self-suspending threads shows some interesting features: First, we see that both lines start at the exact same point (0.27) when no thread is marked as waiting as SET systems fall back to using basic tasks, if this is possible. So at #waiting=0, all threads become basic tasks and are allocated onto the same stack (Figure 6.11d). However, if we increase the number of waiting threads from zero to 15 waiting threads, which increases the ratio of waiting threads from 0.00 to 0.75, the stack space cannot be shared as efficiently and the stack-space factors approach the baseline.

While both variants use more stack space, the BT system ends up at a stack factor of 0.96 and does save almost no stack space anymore. On the other hand, the SET systems have at most a



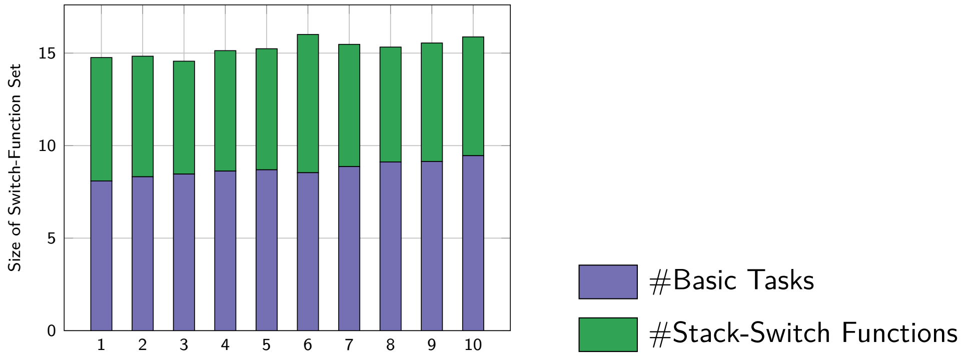
(a) Interrupt Sources (#irqs)

(b) Threads (#threads)



(c) Functions in Callgraph (#functions)

(d) Self-Suspending Threads (#waiting)



(e) SRP Resources (#resources)

(f) Legend

Figure 6.11 – Composition of the Switch-Function Set. We mark threads whose entry function is selected as a switch function as a basic task. Therefore, the stack saving for SET systems is implemented via basic tasks *and* semi-extended tasks.

stack factor of 0.89. We understand these different stack-consumption trends, if we look at the switch-function sets in Figure 6.11d: While a waiting thread can no longer become a basic task, we can (partially) absorb the impact of self-suspension by increasing the number of functions that switch stacks during the thread's execution.

If we look at the #function dimension (Figure 6.10c), we can see that both stack-space factors are relatively stable over the whole parameter range with a maximal delta of 9.35 percent points. However, if we look at the number of switch functions (Figure 6.11c), we have to switch stacks at more and more points in the call graph to achieve the around the same stack-space factor. Nevertheless, even in this dimension, we use on average at most 13 function to become switch functions. In comparison, over all systems, we marked 6 functions as switch functions and 13 threads as basic tasks.

For the #thread dimension (Figure 6.10b), we see that an increased number of threads decreases the stack-space factors and both variants take advantage and achieve factors of under 0.8. However, the gap between both variants shrinks with rising thread numbers as the ratio of waiting threads drops from 0.5 to 0.05 (#waiting is fixed at 10). We can confirm this observation if we look at the number of basic tasks (Figure 6.11b): more and more threads are simply marked as basic tasks and the need for switch functions decreases.

For the #resources domain (Figure 6.10e), we see a slight decrease in stack-space factors for both variants. This stems from the fact that more SRP resources result in more fine-grained preemption constraints that are used in the ILP analysis, which benefits both variants equally. Although basic tasks act on the thread level, a higher number of SRP resources, which is a fine-grained preemption-control mechanism, yields a slight increase in the number of basic tasks (Figure 6.11e).

While these trends indicate moderate savings in the average case, I want to highlight potential of the SET approach by looking at the system where the BTS and the SET approach were most beneficial to the stack-space factor. The best stack saving occurred for a system with no self-suspending threads (#waiting=0), such that the BTS system (and the equivalent SET system) could reduce the stack-space factor to 0.008 by marking all 20 threads as basic tasks. The largest difference between BTS and SET system was seen for a system from the #functions=600 parameter class: The SET system achieved a factor of 0.48, while BTS system only achieved a factor of 0.95.

fine-grained constraints

For the evaluation, I used my own WCSC analysis that supports fine-grained preemption constraints, which I extract from the GCFG. Therefore, it is interesting to see how our results would look without these fine-grained constraints. To give a hint about this matter, I removed all constraints that stem from the GCFG and only used constraints that stem from the static real-time parameters on the task level. For the benchmark systems, this boiled down to the non-preemptability constraint parameter. To make things short, none of the 14 700 systems showed a reduced stack reservation without the fine-grained constraints. Therefore, all savings in Figure 6.10 are only enabled by the interaction analysis.

6.4.5 Discussion

validity of results

As no other WCSC method supports fine-grained preemption constraints and intra-thread stack switching, I was forced to develop and use my own ILP-based analysis method to investigate on the achieved stack-space savings. This reliance on my own method poses a threat to the validity of my evaluation results. However, the derivation of my method from the IPET and the existence of a prototype that produces optimized and working systems gives me confidence in my results.

Furthermore, I used the same set of fine-grained preemption constraints for all variants. While most traditional WCSC analyses only consider task-level constraints, these fine-grained constraints benefit both variants and even lead to an increased use of the BT mechanism (Figure 6.11e). Hence, my WCSC analysis and the fine-grained constraints are also advantageous for BTS systems, even if no SETs support is present.

generalizability

Since the evaluation is purely based on synthetic benchmarks, the generalizability of my conclusion to actual systems might be limited. By looking at individual systems, we have seen that the achieved savings can vary widely between systems of the same parameter class. For example, for the best improvement over BTS systems, I have seen a system with a factor of 0.48 where the average factor was 0.81 for the same parameter class. However, I tried to minimize this potential threat by putting as little restriction on call and dependency graphs in the benchmark generation as possible. Furthermore, since the evaluation covers so many parameter classes, which were organized along five different dimensions, I gained confidence that the observed trends can be reproduced for actual applications.

compiler limitations

For the SET concept, I rely on compiler and RTOS modifications. This might be a problem for the application of SETs in industry, as often only closed-source compilers and operating systems are available. However, in recent years, the trend goes from using closed-source solutions towards using more open-source operating systems [AEe17].

Furthermore, we can also mimic SETs by manually splitting out transferred call-graph subgraphs. For this, each SET gets an additional accompanying BT that gets activated at the call sites of switch functions and the invoked function and we pass all function-call parameters explicitly to the BT, which invokes the switch function on behalf of the original thread. This method requires more interaction with the RTOS and will yield higher overheads as more thread-control structures are necessary in the kernel. Nevertheless, the memory consumption could still decrease in an end-to-end comparison as we still benefit from the optimized switch-function set.

Another technique, which is potentially faster but more invasive, to manifest the switch-function set is to mimic our compiler modifications with inline assembler, binary post processing, and use of RTOS hooks. For GCC 8.2, we can insert the stack-switch instruction at the function entry with inline assembler and clobber the stack pointer to force the usage of base *and* stack pointer.[13]This results in a binary where the switch instruction comes after the prologue (Section 6.3.1, line 6). Therefore, we would have to post process the binary to reorder the instructions in the function prologue, which should be achievable in a rather robust manner. The `TOS_BTS` bookkeeping could be done without RTOS modifications if the RTOS provides hook functions at thread preemption and resumption point. There, we would have to monkey patch the RTOS-internal `TOS_BTS` variable if a SET gets preempted or resumed on the BTS. This would be possible in OSEK, as the standard provides the hooks PreTaskHook and PostTaskHook.

The compiler modifications also have an impact on the optimization stage, since we force the usage of both base and stack pointer for the function body of switch functions. This is similar to disabling the `-fomit-frame-pointer` optimization for the annotated functions, which results in a higher register pressure and potentially more register-spill operations. However, in practice, the effects of this switch-function restriction are negligible as the compiler is still allowed to omit the frame pointer in all un-annotated functions and the number of switch functions is small (6 functions on average).

switch mechanism

Besides the stack-switch mechanism that I described in detail in Section 6.3.1, other similar variants are possible and worth considering.

As a first variant, we could change the location of the switch mechanism from the entry code of the callee to the call-sites.[14] For this, we would have to manipulate the affected call sites such

---

[13] `asm volatile("mov TOS_BTS, %%esp;" ::: "%esp");`

[14] This variant is similar to the difference between call advice and execution advice in aspect oriented programming [SL07; Kic+97]. However, the employed mechanism acts on the ABI and machine-code level instead of the programming-language level.

that the stack-switch instruction is invoked before the call instruction. Furthermore, all stack-passed parameters must be pushed also onto the BTS such that the invoked function body can address all arguments relative to a single register. With this variation, we would have a finer control over allocation of frames onto the different stacks as the same function could be invoked at one place on the private and at another location on the shared stack. This finer control could lead to even lower stack-space allocations and allows us to transfer functions whose compilation we do not control (e.g., closed-source libraries).

On the downside, this variant has some drawbacks that led me to implement the callee-site modification. First, the transfer of all parameters before the actual invocation requires larger compiler modifications as we would have to handle two stack pointers simultaneously during the switch sequence. Furthermore, we would require more instructions as the switch sequence is longer and as we would have to patch several call sites for each transferred function. However, a hybrid approach that makes beneficial use of caller-site switching is thinkable and a topic of further research.

For the second variant, we look at the presented stack-switch implementation within the function body. Here, the stack switch was implemented unconditionally. While this is the most efficient and most minimal implementation, it has the drawback that no child function of a switch function can be marked as a switch function itself as the `TOS_BTS` is not updated in the meantime. We could solve this issue by making the stack switch conditional and override the stack-pointer only in those cases where we are not already on the BTS. However, this also has some drawbacks that prevented me from implementing this variant: First, it makes the compiler modification more complex and inserts more run-time overhead to the application. Second, the switch sequence now has to know the bounds of the BTS statically to become efficient, which fixes the location of the stack in memory. In the evaluation of the greedy SET selection, we have seen that a complex switch mechanism can be beneficial and will also outperform the current mechanism consistently if fully integrated with the genetic algorithm.

protection domains

For security and fault-tolerance, memory protection is an essential technical measure to avoid the spread of attackers and faults throughout the system [▷Hof+15]. Thereby, the RTOS instructs the hardware to prevent a thread from accessing and modifying the memory of another thread. Therefore, we have to put some focus on the issue if we want to use shared stacks without harming the isolation guarantees.

In small embedded systems, the protection is often provided by means of a *memory-protection unit (MPU)*. In contrast to MMUs, an MPU does not provide virtualization but only grants or prevents memory access based on a set of memory-range CPU registers. Furthermore, the granularity is often much finer than the page-based protection that is provided by MMUs (e.g., 8-byte granularity for Infineon TriCore [08]).

For systems that are capable of running whole threads on the shared stack, the RTOS must already configure one MPU register, at thread-dispatch time, to allow access from the last used byte on the BTS (`TOS_BTS` variable) down to the bottom of stack. Thereby, newly started threads cannot access the stack frames of thread that were preempted on the BTS. If we want to implement SETs, we have to configure this BTS range, but we also require a second range to allow access to the thread's private stack. As MPU ranges are a limited resource (Infineon TriCore [08] provides at least 4 ranges up to 16 ranges), this could become an issue for systems with complex protection and sharing patterns.

One solution to this scarcity of MPU ranges is the capability model that is provided by the experimental CHERI processor [Chi+15]. There, the processor supports capabilities as unforgeable fat pointers with memory bounds that can be stored to and loaded from memory without RTOS interaction. We can think of such a capability system as an MPU engine that can safely be reconfigured by the unprivileged user. Xia et al. [Xia+18] could even show that such a capability based processor extension is in the same hardware-cost range as an MPU component. For SETs, we would give a SET two capabilities (one for the private stack and one for the BTS), which are used as stack pointers. At the stack-switch points, the exchange of the stack pointer would automatically change the set of accessible memory addresses.

The other shared data structure is the `TOS_BTS` variable, which communicates the last used byte on the BTS to the threads. However, it is sufficient to have read-only access to this variable since only the RTOS updates it while preempting a thread on the BTS. While it is typically for the target domain to provide read-access to a part or even the complete kernel state to foster efficiency, we can simply copy the contents of the real `TOS_BTS` variable to a thread-local variable during the thread dispatch. Another possibility is the use of global address registers, as they are provided by the Infineon TriCore [08], that can be marked as read-only in the unprivileged user mode.

Summarized, SETs will work well with different memory-protection schemes and only require one additional MPU range.

impact on WCET

Another aspect of the SET method is its impact on the WCET analysis of a thread. Since the call graph(s) and the CFGs remain untouched, all knowledge about preemptions and the data flow remain intact, and we only have to consider the changed machine code and the disruption in the stack-pointer value. However, with the current switch mechanism, at most one switch and one stack-pointer disruption is possible in the (longest) execution path of a SET.

While the additional instruction with one additional memory read (`mov esp, [TOS_BTS]`) will have a minor impact on the WCET, the disruption in the stack pointer can have a larger impact. As the value of `TOS_BTS` is not known exactly, the cache analysis cannot say for sure which cache lines will be evicted by the call-frame allocations after the stack switch. Therefore, the cache analysis becomes more conservative and the WCET bound can increase. As of this reason, the WCET and the WCRT analysis should be done *after* the application of SETs, which fosters my argument that only the implementation carries the truth about the actual system behavior.

6.5 Summary

Starting point for the *semi-extended task (SET)* approach was the observation that stack-space memory is reserved statically for each thread if continuous private stacks are used. This leads to larger-than-necessary stack-space reservations as the dynamic maximum of existing call frames in the whole system is often smaller than the accumulated *worst-case stack consumption (WCSC)* of each thread. While basic tasks and the usage of a *basic-task stack (BTS)* already ease the problem significantly by sharing the stack space between threads, the basic-task concept cannot be applied to self-suspending threads that want to wait passively.

With *semi-extended task (SET)*, I presented a concept that shares the BTS memory among basic tasks and self-suspending threads by partially transferring a thread's stack frames onto the shared stack. For this, a SET, which starts executing on the private stack, transfers parts of its execution at statically determined points to the BTS. There, the SETs and the other basic tasks draw their call-frame allocations from the same stack-space reservation. If some of these allocations are mutually exclusive, the reservation can be smaller as space is shared between the threads.

On a more technical level, SETs switch to the BTS at marked switch functions, which exchange the stack pointer in a modified function prologue with the top-most address of the BTS; at return, these functions restore their private stack pointer. This special prologue is inserted by a modified compiler backed that manifests the switch-function selection, which is carried out by a genetic algorithm that searches for a beneficial balance between private-stack sizes and the BTS reservation. For this selection process, and for the actual dimensioning of the BTS, I developed a new *worst-case stack* *consumption (WCSC)* analysis technique, which is based on the *implicit path-enumeration technique (IPET)*, and that benefits from fine-grained preemption constraints. I extract these constraints from the GCFG, which covers all preemptions between threads on the granularity of ABBs.

In the evaluation, I could demonstrate the benefits of the SET approach in comparison to state-of-the-art by applying the principle to over 14 000 generated benchmark systems. With this systematic evaluation that covers five different dimensions of real-time system parameters, I examined characteristics and limits of the SET approach in comparison to the state of the art. In the process, the polynomial *system-state flow (SSF)* analysis proved to be feasible and scalable, even for large real-time applications. Over all generated benchmarks, the SET concept reduces the stack-space reservation on average by 7 percent if compared to a pure basic-task system; 80 percent of the systems ended up with a smaller reservation. Compared to a system without stack sharing, the reduction was even 22 percent.

research questions

With this chapter, I am able to provide an answer to my first research question (RQ1): Control-flow–sensitive interaction analysis scales to a wide range of real-time systems and provides necessary insights, in terms of preemption constraints, to improve non-functional properties.

Since the SET optimization only provides a different system configuration but keeps the behavior of the real-time application intact, we also see that BT-only, and even more ET-only, systems expose an unnecessarily high static memory consumption. From this inefficiency, I can provide an answer to my second research question (RQ2): The segregation of the application into individual threads of execution leads to the segregated allocation of stack space. If we overcome this strict segregation and use the integrated view on the application–RTOS interaction, we can reduce the system-wide stack consumption significantly over a wide range of systems. This provides an answer to my third research question (RQ3): Only with the fine-grained preemption constraints, the WCSC analysis was able to exploit the saving potential of the SET apporach and to improve the static memory consumption of the whole system.

7

OSEK-V

An Application-Specific Processor Pipeline

> Hardware is just petrified software.
>
> Karen Paneta Lentz

With the SET approach, we saw how we can use interaction-aware optimizations of the whole system to cut down on overly pessimistic memory reservations of the stack space. In this chapter, we use the results of the interaction analysis to tailor the execution platform that runs the RTCS tightly to the requested RTOS behavior. Thereby, I explore the benefits and costs of extensive RTOS specialization that reaches the domain of application-tailored hardware designs.

Using an RTOS provides abstractions and interfaces to the developer that ease the implementation and composition of the real-time application. However, since generic RTOS services introduce latency and memory overheads, developers, especially in a HW/SW co-design setting, try to avoid the usage of a full-blown RTOS. With OSEK-V, I shift this trade-off by pushing the application-specific interaction model into the processor to replace the RTOS by a behavioral-equivalent hardware component, the *system-state machine (SSM)*. This application-specific processor extension, which behaves like the RTOS and controls the hardware-thread scheduling, significantly reduces the event latencies, the interrupt-block times, and the memory footprint at a moderate FPGA-resource consumption.

Related Publications

[▷DHL15] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. "Back to the Roots: Implementing the RTOS as a Specialized State Machine." In: *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. OSPERT '15 (Lund, Sweden). July 2015, pp. 7–12. URL: `http://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p7.pdf`.

[▷DL17] **Christian Dietrich** and Daniel Lohmann. "OSEK-V: Application-Specific RTOS Instantiation in Hardware." In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '17 (Barcelona, Spain). New York, NY, USA: ACM Press, June 2017. DOI: `10.1145/3078633.3078637`.

7.1 Problem Field and Related Work

In the introduction, I compared the specificity of the centrifugal governor with the generality of software-based control-systems. While the first provides exactly the required control of a physical process at high engineering cost, the latter provides a high degree of flexibility and updatability but introduces latencies and jitter in the discretized control algorithm. This tension becomes more evident in hardware/software co-design settings where the hardware is no longer seen as a fixed and unchangeable bedrock for the execution of software, but can be adapted towards the specific requirements of the indented application. In this co-design setting, the question is urgent: what should be put in software and what should be part of the hardware?

With this question in mind, we now take a look at the real-time operating system and its interaction with the application. There are two competing aspects and we have to weigh in the benefits of the provided coordination services and the frictions that a software-implemented RTOS introduces. On the one hand, the abstractions offered by the RTOS, like prioritized threads, periodic activations, and synchronization, significantly ease the development and composition of applications with multiple interacting control tasks. Especially, if multiple vendors are working on one product, a software-based solution is easier to adapt in the integration process. And while ambiguities exist between the RT domain and the OS domain (see Chapter 2), there is often a rather straight forward projection between both abstraction levels. In total, an RTOS interface fosters the productivity of the developers.

On the other hand, a system is less analyzable/predictable with the usage of a software-based RTOS if we compare it to bare-metal, or even hardwired, implementations of the real-time system. Since the RTOS execution and the application compete, with the RTOS always winning, for processor time and cache lines, significant latencies, jitter, and cache-induced preemption delays are put as a burden on the application. In Chapter 4, we have already discussed the RTOS impact on the worst-case response time. Furthermore, engineers shy away from software solutions if only parts of the functionality are actually required as they fear unnecessary overheads. In total, RTOSs often come as an all-or-nothing package that interferes with your application and that makes it harder to build a predictable and analyzable system.

If we push the RTOS into the hardware, we can reach a compromise, where we can still use RTOS abstractions and interfaces but avoid (some) of the negative effects of executing it in software. By offloading RTOS chores to a separate hardware component, like a scheduling co-processor, RTOS and application no longer compete for resources, which minimizes their interference. With a dedicated hardware component, we also have the chance to specialize it exactly for the requested chores instead of executing the RTOS' algorithms on the general purpose processor. Conceptually, such a push down to hardware harmonizes well with the system community's interpretation that the operating systems is an extension to the actual processor [Tan06] and part of a hierarchical machine. While a software-implemented RTOS performs a partial interpretation of the system calls, a hardware-implemented RTOS exposes system calls directly as part of the machine interface.

One prime example, where the trade-off between a hardware and a (partial) software implementation of the same OS functionality, namely virtual memory, was explored in an industrial context, is the MIPS [01] architecture. In its early days, MIPS implemented address translation with a software-managed translation-look–aside buffer (TLB) [BLM17, cha. 4.6]; for every TLB miss, the processor trapped and an OS-provided ISR had to fill the missing TLB entry. With this solution, the hardware was kept small but the user had to pay the execution and interference cost for walking the page tables in software. In later revisions, MIPS also provides a hardware-based page-table walker that performs this chore in a separate hardware component, which fostered performance but also reduced the flexibility for the OS developer how to organize the page tables.

Aside from MIPS, pushing (parts of) the operating system and its chores, like protection, scheduling, communication, and synchronization, into the hardware has a long-standing tradition [DD68; Org72; CKD94; AA82; Bur+99; Nak+95; Arn+14; Chi+15]. In this discussion of the related work, I will focus on works that touched the scheduling and thread synchronization as these are the most important aspects of an RTOS as they directly influence the timeliness of the whole RTCS.

HybridThreads [Agr+04; Agr+06] accomplish low run-time overhead and fast interrupt handling by placing the OS scheduler in a separate hardware component. The authors implemented different scheduling policies, like round robin or preemptive-priority scheduling, whose decisions are manifested by sending an IRQ to the processor, which triggers a thread switch. Mutex-based thread synchronization is also offloaded to hardware and allows the synchronization between software threads running on the CPU and activities that are performed by specialized hardware accelerators. The proposed hardware components are application agnostic and configured from software.

Sloth [Hof+09; Hof14; Mül+14; Dan+14; Hof+12; HLS11] achieved a far-reaching RTOS offloading for OSEK-like systems without requiring specialized hardware. For this, the scheduling was delegated to the interrupt controller, which already supports the selection of the highest pending interrupt. By mapping threads and ISRs onto distinct interrupt sources, they could not only reduce the interrupt and kernel latencies drastically, but also provide a uniform priority space, which avoids the rate-monotonic priority inversion [LMN06]. With *Sloth on Time* [Hof+12], they extended their work and offloaded the generation of periodic signals for a time-triggered system to the timer subsystem of the Infineon Tricore processor.

The FlexPRET processor [Zim+14] is an extension of a RISC-V [Wat+14] core that aims for an efficient and predictable execution of mixed-criticality systems. Their pipeline, which supports multiple hardware threads (in RISC-V lingo: *harts*), distinguishes between hard–real-time harts with a guaranteed time budget and soft–real-time harts. The execution of harts is interleaved on the instruction level, while the configured budgets are enforced. In contrast to OSEK-V, FlexPRET provides a processor abstraction instead of a thread abstraction as inter-thread dependencies or synchronization are not considered.

The ReconOS project [LP09] provides a unified OS interface, resembling POSIX, for software threads and hardware components. Here, the hardware components, which can be instantiated by dynamic reconfiguration of an associated FPGA, are integrated into the software-implemented OS by means of a *delegate thread*, which invokes requested system calls in their name. However, all OS chores are executed on the general-purpose processor and only its interface is exposed to the hardware components.

With a greater focus on configurable systems, the *δ-framework* [MB02] aims for an RTOS-HW co-design where the user decides manually whether components of the Atlanta RTOS are instantiated in hardware or in software. Thereby, it provides an application-specific platform on the level of features [▷Fie+18] that is agnostic to the number of system objects or their interactions. In a similar direction, *LaVA* [MBS15] combines the instantiation of peripheral devices (e.g., accelerators) and their automated integration into the RTOS' device-driver interface.

Compared to these OS–HW integrations, I aimed for a high degree of application-specific tailoring of the RTOS component. Instead of reproducing the software implementation structurally by means of a hardware implementation, OSEK-V incorporates the interaction model, and thereby only the RTOS *behavior*, into the hardware. Thereby, (synchronous) system calls become actual machine instructions, instead of partially interpreted ones. In some abstract sense, this resembles the path-specific system-call optimization known from Synthesis [PMI88; MP89] on the hardware level. In Synthesis, hot-path system calls, like `read()`, get, for example, (partially) specialized for the used file descriptor at run time; thus, avoiding most of the dynamic dispatching in the hot path. Later, with the Tempo framework [McN+01], an automated approach for such specializations was proposed.

At the static end of system-call specialization, Barthelmann [Bar02] proposed an optimized register allocation that minimizes the number of alive registers at the system-call sites; Thereby, he could minimize the size of the thread context for each synchronous preemption point. In my own work [▷DHL15b; ▷Die14], I also explored static system-call specialization. With the aid of the GCFG, individual system-call sites get specialized for the specific system state at that point. For example, if no inter-thread edges originate from a system-call ABB, no scheduler invocation is necessary. Nevertheless, the kernel's original code structure was still the line of guidance for these specialization methods.

In OSEK-V, I express the interaction model by means of a *finite-state machine (FSM)*. Using FSMs as the organizing principle of a whole system has also been proposed for deeply embedded sensor nodes to enhance simplicity and energy efficiency: SenOS [KH05] is a software event dispatcher and executor for multiple manually-encoded state machines. Kothari, Millstein, and Govindan [KMG08] derive compact state machines (< 16 states) from TinyOS programs by symbolic execution to foster understanding of existing applications.

7.2 System-State Machine as Executable Interaction Model

The core idea behind OSEK-V is to express the behavior that the RTOS *would* expose for a given application as a FSM. This *system-state machine (SSM)* is a Moore-automata [Moo56] that transitions on *system events*, like system calls or external interrupts, and has the currently running thread as an output signal. In Section 7.3.1, I will explain in detail how the integration of the SSM results into an application-specific processor pipeline works.

$$SSM : \underbrace{\langle\text{system event}\rangle}_{\text{input}} \times \langle\text{FSM state}\rangle \rightarrow \langle\text{FSM state}\rangle \times \underbrace{\langle\text{current thread}\rangle}_{\text{output}}$$

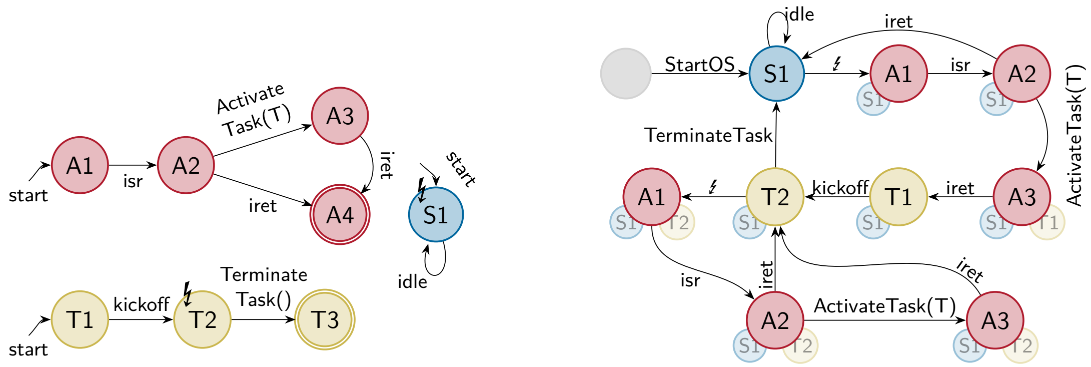
While we can describe, in general, the behavior of every OSEK kernel as such a FSM, the SSM includes only the *desired* kernel behavior, with regard to scheduling, in the presence of *one* specific application. This focus on one specific application allows us to tailor the SSM implementation close to those services that this application actually demands from the underlying RTOS. Therefore, our first goal is to derive a minimal SSM that exposes the same rescheduling sequence as a normal kernel implementation if fed with the same sequence of system events.

ASM-derived SSTG

The base for the SSM construction is an ASM-derived SSTG (see Section 3.4.3.2). I use this SSTG variant as an interaction model as it already exposes a smaller number of nodes but remains *trace equivalent* to the more detailed ABB-based SSTG. Since we only want to capture the rescheduling sequence without being interested in the internal SSM state, trace equivalence is sufficient and allows for SSTGs with a lower number of states and edges.

In Figure 7.1a, we see three ASMs that form a complete system with one ISR, one thread, and the idle thread. Derived from it, Figure 7.1b depicts the corresponding SSTG: Each node consists of one (foreground) ASM state, which indicates the position in the currently running thread; the preempted, or not-yet started, threads in the background. The edges are labeled with those system calls that the application has to invoke for a transition to occur.

The ASM-derived SSTG is already an instance of the SSM for the given application. Upon a system event (system-call labels or interrupt), it transitions between states, and we can derive output signal by inspecting the SSM state for the currently running thread. However, the SSTG is still larger than necessary as its state–state–transition trace distinguishes between different foreground ASM



(a) Application State Machines

(b) ASM-derived SSTG

Figure 7.1 – System State Machine. From the *application state machines (ASMs)*, the SSE analysis derives an SSTG, which is the base for the SSM. Adapted from [▷DL17].

states with the same foreground thread instead of only providing the sequence of foreground threads. Therefore, we can trade in this unnecessary precision for a more compact SSM.

For this, we minimize the SSTG and allow that states with different foreground ASM states are merged as long as the thread-switch trace remains equivalent. Luckily, state-machine minimization is a well-covered and long-standing topic of computer science [Moo56; Hop71]. Therefore, I will only discuss the rough outline of the used Moore's algorithm and discuss the SSM specifics bits.

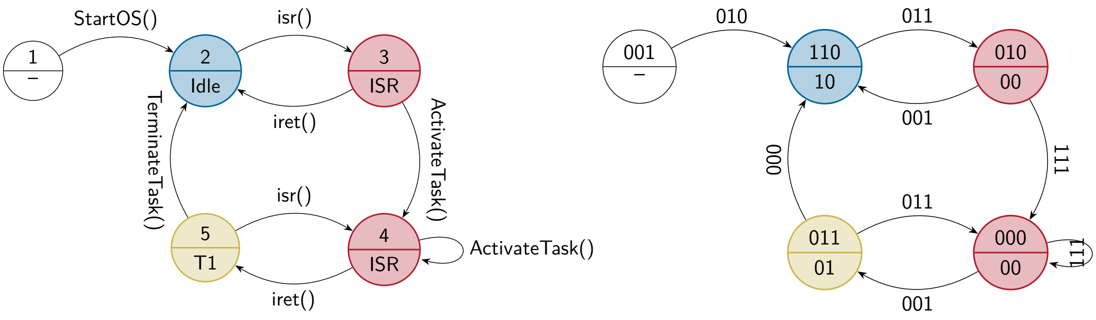
In general, Moore's algorithm for minimizing deterministic finite *automata* (DFA) works by an iterated state-partition refinement. While designed for automata that accept a formal language, we can also apply it to FSMs. The goal of Moore is to build a smaller DFA without changing the accepted formal language. For this, Moore starts with the 0-equivalence partition of the state set: each accepting state gets its own partition and all non-accepting states are placed in one large partition. In a fixpoint algorithm, we further split partitions according to an equivalence relation: Two states are in the same partition, if the set of ⟨edge label, followup partition⟩-pairs for their outgoing edges are equal. After the algorithm reaches a fixpoint, every partition becomes a new state in the minimized DFA and transitions are added accordingly to the partition's connections.

SSM minimization

Applied to SSM minimization, we have to make some adaptions to this basal algorithm. First, the SSM is no acceptor for a formal language but its value lies in the output signal. Therefore, we form the 0-equivalence partition according to the currently-running-thread property. Furthermore, the SSM is allowed to ignore system events as long as the thread-switch trace remains untouched. For example, if two system calls are always executed in sequence without intermediate rescheduling, the effect on the SSM can be tied to the latter system call. Therefore, we only use the set of ⟨followup partitions⟩ as the comparison key between states, instead of also considering the edge labels.

As result of this minimization, we get an SSM whose thread-switch trace is equivalent to the SSTG, but with a minimal number of states. However, the number of edges is not minimal yet: Due to the state-merging process, we introduced transitions that are self loops, which are also labeled with system events. If a label occurs only on self-loops, it can never bring the system into an observable different state, and we can eradicate the label, the corresponding SSM states, the system event, and all related system-call sites. At this point, it is important to re-emphasize that SSTG/SSM labels are not system-call *types* but system-call *sites*.

One point where I make conscious use of the self-loop eradication is the removal of interrupt system events (↯) from the SSM. These interrupts are the only system-event class that is triggered



(a) Minimized SSM **(b)** SSM with assigned labels

Figure 7.2 – Minimized State Machines. The minimized state machine, with and without assigned state, output, and transition labels, for the running example. Adapted from [▷DHL15a].

asynchronously from outside the processor. Therefore, we introduce an artificial `isr()` system call that is unconditionally executed at each ISR entry (see Figure 7.1a). As no thread switch happens between interrupt event (↯) and `isr()` invocation, the interrupt edge becomes a self-loop and the `isr()` performs the necessary state modification. Thereby, we are able to remove all interrupt edges and end up with an SSM that only transitions on (artificial) system-call invocations.

In Figure 7.2a, we see the minimized SSM for the example system from Figure 7.1. We see that the interrupt edges are removed from the SSM and that both interrupt–iret loops from the SSTG are folded into only two SSM states. Furthermore, from the example it becomes clear to what degree the SSM is dependent on the given application: While the application can return from the interrupt state 3 with or without invoking the `ActivateTask()` system call, the SSM contains no information why this decision is made; it only transitions upon the sequence of system events. Thereby, an implementation of the SSM is usable for many applications as long as their minimized SSM is a subgraph of the original SSM.

hardware implementation

The FSM minimization produces an SSM that is fully symbolic: Each state identifier, each thread identifier, and each label are symbolic (e.g., “`TerminateTask()`”, “T1”) and make only sense in the abstract domain. For an instantiation in hardware, we have to choose bit-vector widths for each symbol class and concrete numerical values for each symbol. This selection process, which is known as the *state-assignment problem* [VS89; Dev+88; VT88], largely influences the complexity of the hardware implementation.

Thereby, a good state assignment is heavily dependent on the targeted hardware design (i.e., PLA, CPLD, FPGA, or an actual ASIC). As this area is not the focus of this thesis, I chose the classical NOVA approach [VS89], which targets an optimal encoding for a two-level logic implementation as it is found in *programmable logic arrays (PLAs)*. NOVA chooses the bit vectors for the edge labels (system events) and the states identifiers, when supplied with an encoding for the output signal (thread identifier). While the thread-identifier encoding is irrelevant for the intended use case, NOVA does not support a minimizing output encoding. Therefore, I used the order of the thread declaration in the OIL file as the identifier. The result of the state assignment can be seen in Figure 7.2b, where, for example, “`TerminateTask()`” is encoded as 000_2.

With the state-assignment, we derived a transition function $\mathbb{B}^n \rightarrow \mathbb{B}^n$ from the SSM as a truth table, where each row represents one transition. For example, for the “`TerminateTask()`” edge, we get “011 000 → 110 10”. Internally, NOVA uses the ESPRESSO [Rud86] heuristic logic minimizer, which collapses rows and introduces don’t-care terms to derive a minimized truth table. With this

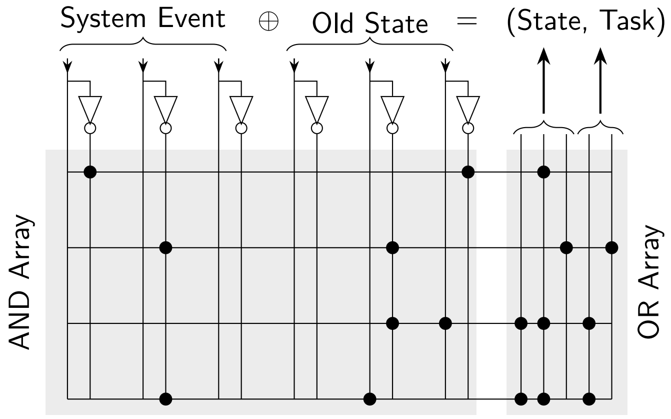


Figure 7.3 – PLA Implementation of the SSM. With the minimized truth table, we could implement the SSM binary function in a programmable logic array (PLA). From the system event and the old SSM state, it derives a new state and the next currently running thread. Adapted from [▷DHL15a].

minimized table, we could directly implement the SSM with a PLA chip, as it is shown for the running example in Figure 7.3.

Another implementation variant, which I used for prototyping OSEK-V is to emulate the PLA behavior in software: On every system event, the kernel loops over the minimized truth table, check if the input bit vector matches a row and OR it onto the result vector. While this implementation is able to replace large parts of the kernel code of an OSEK, it is only practical for very small systems as the kernel overhead scales linearly with the number of truth-table rows. However, for a scenario where code obfuscation is desired, such a software-implemented PLA variant could be of practical use. For the rest of this thesis, I will ignore this implementation variant, but refer to [▷DHL15a], where I investigated on the arising overheads.

7.3 The OSEK-V Processor

The goal of OSEK-V is to provide an application-specific processor design that behaves, in presence of a given application, like a software-implemented RTOS kernel. Therefore, we have to integrate the SSM transition function into a processor and let it control the thread execution.

7.3.1 Pipeline Integration of the SSM

OSEK-V is an extension of the Rocket core generator [Lee+14; Asa+16] that includes a given SSM model, in form of the minimized truth table and some system-configuration information (i.e., number of threads), into the design. The Rocket is a 5-stage in-order scalar core generator that implements the RISC-V [Wat+14] ISA, which is explicitly designed to support computer-architecture research. Its generative approach goes well with the OSEK-V concept and already exposes a multitude of configuration switches to build an application-specific processor. For OSEK-V, I used the 64-bit mode (RV64) as the 32-bit mode was not mature enough at the time of the OSEK-V development. However, the presented approach is independent of this. The generator produces a cycle accurate simulator of

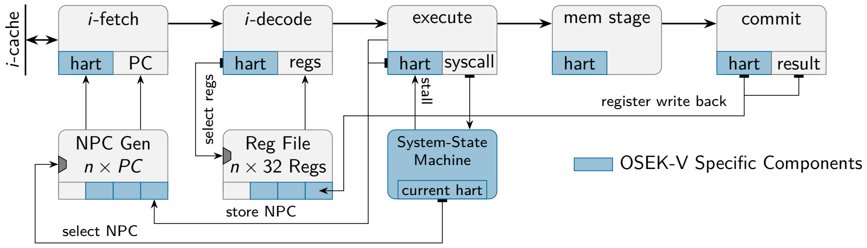


Figure 7.4 – The OSEK-V Pipeline. The 5-stage in-order Rocket [Asa+16] core is extended by the system-state machine (SSM) and hardware multithreading. Each RTOS thread gets executed in its own hardware thread (hart) and the SSM controls the hart selection. Communication from the pipeline to the SSM is done with special instructions in the execute stage. Adapted from [▷DL17].

the processor, as well as a Verilog description that can be synthesized for an FPGA or can be used in an ASIC process [Lee+14].

Figure 7.4 shows the extended Rocket pipeline; all blue components are OSEK-V specific additions. In a nutshell, OSEK-V uses hardware multithreading, with one hardware thread (*hart* in RISC-V lingo) for every RTOS thread. The scheduling of these harts is controlled by the SSM, which receives system events from the execution stage of the pipeline. Internally, the SSM consists of a register that holds the current state and an implementation of the transition function. Furthermore, it exposes the currently-running hart and is able to stall the pipeline if a transition takes more than one cycle.

Traditionally, hardware multithreading, which was first researched in the IBM ACS/360 project [Fen73; Cel+73], is used to increase the utilization of the processor by exploiting instruction-level parallelism. For example, in a superscalar processor, a hart can make progress on the arithmetic unit while another hart waits for a memory request. In these *simultaneous hardware multithreading* scenarios, the scheduling between harts is done independent of the operating system and each hart presents itself as an actual independent CPU, although it shares its execution engine with others.

For the implementation of multithreading in the Rocket core (see Figure 7.4), I had to add a hart tag to each stage that travels alongside a specific instruction through the pipeline. Furthermore, I extend the register file to hold a full set of registers for every hart. The *next-program-counter generator*, which holds one program counter (PC) for every hart, inserts the current hart tag and the current PC into the instruction-fetch stage. Down the pipeline, we use the tag to select the hart-private register context in the instruction-decode/operand-fetch and the commit/write-back stage.

additional instructions

As the threads have to communicate with the SSM, the OSEK-V pipeline provides two additional machine instructions: `ssm.tx` and `ssm.ld`. The `ssm.tx` instruction sends the system-event number, which was calculated in the state assignment, from the execution stage to the SSM, which triggers a transition. Before the transition, the SSM stalls the execution until the memory stage and the commit stage run empty, which ensures that traps that occur in the memory stage remain precise. If a hart switch becomes necessary after the transition, I use the branch-mispredict logic to flush the rest of the pipeline and to issue an instruction fetch for the new hart's program counter. If no switch is necessary, the execution directly continues.

The other new instruction, `ssm.ld`, is required during the system boot. There, the RTOS boot code loads the address of each thread's entry function into the corresponding program counter in the NPC-Gen component (see Figure 7.4).

7.3.2 External System-Activation Patterns

When an interrupt occurs, the OSEK-V chip starts the ISR execution, which at some point invokes the artificial `isr()` system call, some other system calls (e.g., `ActivateTask()`), and indicates the end of the ISR by invoking `ssm.tx` ⟨`iret`⟩. All of this is done in software, the current thread execution gets interrupted, and the pipeline is used to execute the code of the ISR. This comes at the cost of flushed pipelines, evicted cache lines, prolonged response times, and extended costs for interrupt synchronization; everything we tried to avoid with OSEK-V. For user-defined ISRs, we have no other choice but to use the CPU. However, the OSEK-V model enables us to offload periodic alarm activations into a separate hardware component. This *static-alarm* component is placed besides the SSM component and reduces the interruption frequency and the computation demand on the main pipeline.

static alarms

With the interaction model available, we can decide statically for which periodic events such an application-specific offloading is especially fruitful. Instead of utilizing a general-purpose timer hardware that is configured dynamically, like it was done by Hofer et al. [Hof+12], I aim for a more tailored variant that does not support dynamic reconfiguration. For this, I search the interaction model for OSEK alarms that are started at boot with a fixed period and offset but never get reconfigured at run time.

For each of these *static alarms*, the chip generator instantiates, according to the given static offset and period, a specialized timer that is placed in the static-alarm component (see Figure 7.5). For example, if the period is a power-of-two, we can use a simple flip-flop clock divider instead of an actual counter. Driven by the real-time clock, these timers push system events into a command queue, enclosed by the `isr()` and `iret()` event. The static-alarm component transfers the queue contents atomically to the SSM if interrupts are currently enabled on the pipeline. This atomicity ensures the same semantic as executing the timer ISR in software.

With the alarm offloading, we remove those code paths from the timer ISR that manage static alarms. However, non-static *dynamic* alarms are still handled in software, and we end up with the same number of interruptions if we use a periodic timer signal with a fixed base frequency. Nevertheless, the static analysis can inspect all alarm-reconfiguration system calls and deduce if we can lower the base frequency of the timer ISR activation. Thereby, we can also cut down on the number of interruptions by offloading of static alarms.

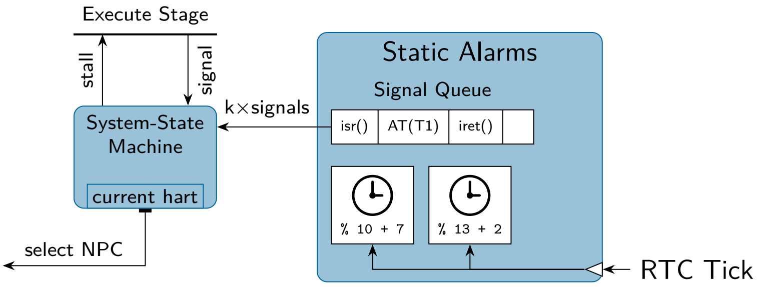


Figure 7.5 – Static-Alarm Component. Periodic system events, like OSEK alarms, are implemented into a hardware component if the event is not reconfigured at run time. If the periodic signal occurs, the alarm component sends a packet of transition signals to the SSM to ensure atomicity.

7.3.3 System Generation and Startup

The Rocket generator reads in the processor configuration, the SSM, and the system configuration (i.e., number of threads, static alarms). It generates an OSEK-V instance either as cycle-accurate emulator or as Verilog code that can be used in the FPGA synthesis. By using application-specific hardware, we also have to adapt the application code itself to use the custom OSEK-V features. For this, I use the dOSEK generator (see Section 3.7) to manipulate the application code, as well as to specialize the kernel implementation.

In the application code, I replace the system-call sites with `ssm.tx` instructions which send the corresponding system event to the SSM. These code locations have to be enclosed by interrupt disable and enable instructions to ensure that interrupts get enabled after a thread is resumed from a rescheduling point in an ISR. This is still necessary, since the current implementation only maps normal threads and the idle thread to a separate hart, while ISRs are executed in the context of the currently running hart. This is a trade-off between ISR latency and the used hardware resources, since the increase in the register file is, besides the SSM size, the driving resource consumer. On a technical level, we modify the transition function to output the hart identifier of the interrupted thread instead of a separate hart identifier for each ISR. However, in principle, it is possible to use a dedicated hart for each ISR.

At boot, the software initializes the program counters for each hart: the kernel uses the `ssm.ld` instruction, which sets the instruction pointer of a specific hart to a thread-specific initialization function. Generated by *d*OSEK, these functions set up the thread's stack pointer and invoke the actual thread-entry function. The generator also ensures that a thread jumps back to this initialization after each `TerminateTask()` system call.

7.4 Experimental Evaluation

In the evaluation, I applied the OSEK-V approach to the *i4*Copter (see Section 4.6.3) in order to demonstrate its costs and the benefits. Thereby, we will see how the push down of RTOS logic into the hardware results in highly-improved kernel latencies and fewer interferences with the application execution. Afterwards, I will discuss these results, as well as strengths and weaknesses of OSEK-V in general.

7.4.1 Evaluation Scenario

As application, I used the task setup of the *i4*Copter (see Section 4.6.3) with only the minimal application code to mimic the directed and undirected thread dependencies. In the interaction analysis, I used ASMs (see Section 3.4.3.2) and application knowledge about implicit deadlines (see Section 3.4.3.1) to start out the SSM construction from an already denser SSTG. In total, the *i4*Copter system has 11 threads, 3 alarms, 1 user-defined ISR, and 52 system-call sites. Two of the alarms are static, while the watchdog timer must be managed in software but can be operated with a lower timer base rate.

For the FPGA costs, I compare two different OSEK-V cores for the *i4*Copter with the *Baseline* Rocket core: the first OSEK–V core just includes the *SSM*, where the second also uses static alarms (*SSM+Alarms*). Additionally, I also built an OSEK-V core for the running *Example* (Section 7.2), which uses an SSM without static alarms.

The interaction analysis and the SSM generation were performed with dOSEK and NOVA (both single-threaded) on a single Intel Core i7-2600 machine. For the OSEK-V generation, I used the extended Rocket chip generator and produced, for each variant, a cycle accurate emulator in C++

and a Verilog source file. The Verilog code was synthesized with the Xilinx Vivado 2015.2 toolchain on the i7-2600 machine for the Zynq-7020 FPGA chip, which is integrated into the ZedBoard platform. For a single logic cell, the Zynq-7020 FPGA features a F_{max} of 100 Mhz. For the synthesis, Vivado was instructed to clock the Rocket's pipeline with at least 25 Mhz.

7.4.2 FPGA Synthesis Costs for the OSEK-V Core

First, I want to discuss the size of the SSM, the required run time to calculate it, and the hardware costs that arise if we integrate and synthesize it for the Zynq-7020 FPGA. For this, Table 7.1 gives an overview about the generation time and the SSM size, which is the driving factor behind the FPGA implementation costs.

Second, Table 7.2 shows the results of the FPGA synthesis. Here, we see the required FPGA resources, as well as the size of the remaining kernel implementation. For the FPGA side, combinatorial circuitry is implemented in lookup tables (LUTs), which are also used as memory elements (Memory-LUT); flip-flops are another storage option in Xilinx FPGAs. In Table 7.2, I only show those resource dimensions that differ between the variants (i.e., the usage of block RAM remained equal).

The running example results in a very small SSM, which is calculated and minimized in fractions of a second (see Table 7.1). Since the initial SSM is already small, the minimization cannot cut away much redundancy. Therefore, the minimized state-transition function consists only of four AND clauses (four AND gates with the outputs combined in one OR gate). Compared to the baseline Rocket (equal to Table 7.2/*i4*Copter/Baseline), we see only a small increase in FPGA-resource usage (+127 memory LUTs). These increases mainly stem from the second register file, which is synthesized by Vivado in Memory-LUT cells, for this two hart system (idle thread, thread T). Without counting the stack space, we end up with a minimal RTOS kernel that only uses 11 bytes of volatile memory for managing the single alarm dynamically (static alarms were not used).

For the *i4*Copter benchmark, the SSM generation takes more than one minute (see Table 7.1), where the run time is mainly driven by the state-assignment phase (96.95 %). The initially large SSM get dramatically shrunk by the Moore minimization (85.5 % less states). The resulting SSM transition function takes a 15-bit input vector (state: 10 bits, system event: 5 bits) and produces a 14-bit output signal (state: 10 bits, hart id: 4 bits).

On the FPGA side (see Table 7.2), we end up with a significant larger core by using OSEK-V. Without static-alarms, we require 8.8 percent more LUTs, which are mainly used for the SSM component (76.09 %). The increase in memory LUTs (+983) stems mostly (96.24 %) from the significantly larger register file, which now must hold 12 register contexts instead of only one. A

		Example	*i4*Copter
SSM Generation	[seconds]	0.06	73.68
SSM (Initial FSM)	#States	9	4 834
	#Transitions	13	7 479
SSM (Minimized FSM)	#States	6	701
	#Transitions	9	1 246
Transition Function	#Clauses	4	781

Table 7.1 – SSM Generation for the Benchmarks. The SSTG interaction model was built with the SSE, shrunk with a variant of Moore FSM minimization, and implemented as an PLA transition function with NOVA. Adapted from [▷DL17].

			Example	*i4*Copter		
			(Figure 7.1)	Baseline	SSM	SSM+Alarms
FPGA	LUT		29 460	29 216	32 041	32 341
	Memory-LUT		1 033	1 160	2 016	2 016
	Flip-Flops		14 208	14 117	14 129	14 196
	F_{max}	[Mhz]	26.37	26.56	26.7	25.67
RTOS (SW)	Code Size	[bytes]	2 436	7 848	3 621	3 349
	Data Size	[bytes]	11	1 904	406	350

Table 7.2 – Resource Consumption of Different OSEK-V Cores. The Verilog code for three OSEK-V cores, one for the running example and two for the *i4*Copter, were synthesized for a Xilinx Zynq-7020 Zedboard. Adapted from [▷DL17].

closer investigation of the synthesized system reveals that we require 86 distributed memory cells per mapped RTOS thread.

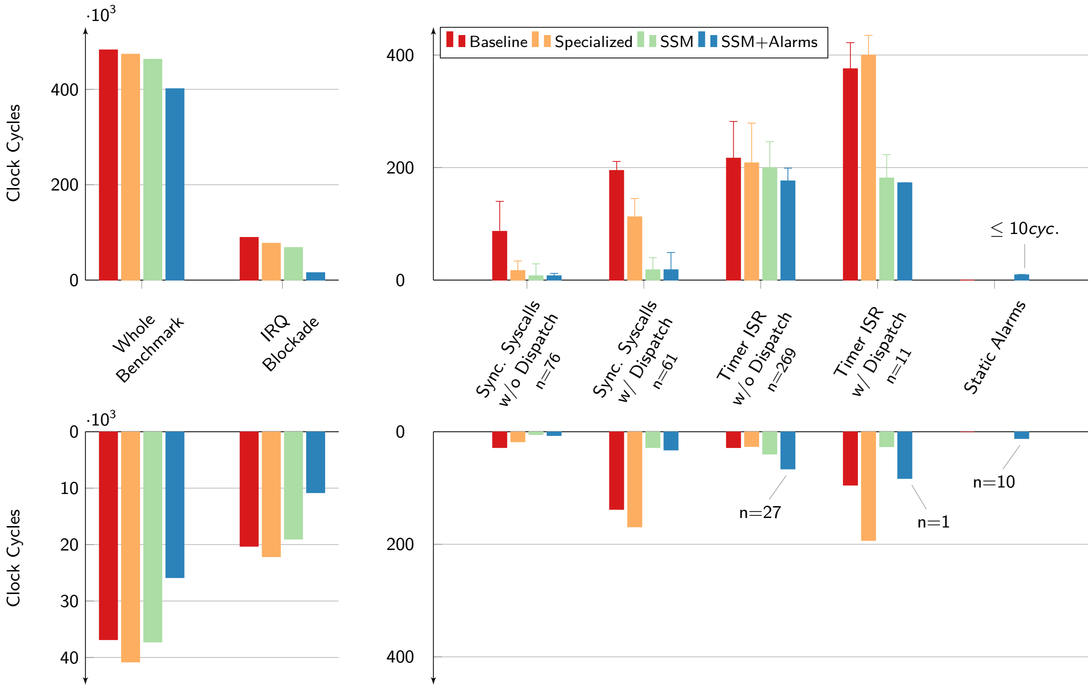
If we add static alarms to the SSM system, the FPGA costs remain on a similar level, and we only require a few additional LUTs and flip-flops. For all variants, the Vivado synthesis tool took about the same time (at least 10 minutes) and was always able to fulfill the 25 MHz timing constraint for the pipeline.

On the software side of the *i4*Copter OSEK-V core, we see that the increased LUT count translates into a much smaller RTOS code segment since no scheduler or SRP resource management is required. From the 1 498 bytes that we saved in the data segment, 1440 bytes were formerly located in the software-managed thread contexts (120 bytes per thread). If we add static alarms (SSM+Alarms), the RTOS shrinks further as two of three alarms are now managed in hardware.

7.4.3 Run-Time Latencies

In order to assess the benefits of using an OSEK-V core, I ran the *i4*Copter for three hyper periods with different system configurations in the cycle-accurate circuit simulator. During the execution, a trace of the pipeline states was recorded, analyzed, and the results are shown in Figure 7.6. The first two variants run on an unmodified Rocket core and give the necessary baseline for the OSEK-V results: The *Baseline* variant is the standard *d*OSEK implementation; all system calls are generic and alarms are managed in software with a constant-frequency timer IRQ. The *Specialized* variant uses the results of the interaction analysis to generate specialized system-call implementations for each system-call call site [▷DHL15b]. In these specializations, the generator removes unnecessary operations, like finding the highest-priority runnable thread, if the result can be deduced from the GCFG. The *SSM* and the *SSM+Alarm* variants execute the *i4*Copter on application-specific OSEK-V cores (without and with static alarms).

In the cycle-accurate trace, I search for instruction sequences that manipulate the kernel state and categorize them into synchronous system calls (with and without thread dispatch), timer ISR activations (with and without thread dispatch), and activations of the static-alarm component. Within these instruction sequences, I distinguish between cycles where the pipeline makes actual progress (Figure 7.6 (a+b)) and cycles where the pipeline is fully stalled because it waits for the cache and the memory subsystem (Figure 7.6 (c+d)). This separation allows us to discriminate the actual computational cost of OSEK-V from the influence of the processor-specific cache hierarchy.



(a) Execution Time (Cumulative)

(b) Execution Time (Per Rescheduling Point)

(c) Cache Stalls (Cumulative)

(d) Cache Stalls (Per Rescheduling Point)

Figure 7.6 – OSEK-V Latencies and Cache-Stall Cycles. Summary of the *i4*Copter running on different Rocket and OSEK-V Cores. The execution trace is divided into execution time, where instructions progress in the pipeline, and cache-stall cycles, where the pipeline is totally idle.

For the execution times, I give not only the average run times but also the largest observed run times (upper bar), since this is an indicator for the predictability of the system.

Over the whole benchmark (kernel+application), the effective clock cycles, where the processor is not in idle, see a successive decrease by specializing in software, and even more by using tailored hardware components (Figure 7.6 (a)). For the moment, we will ignore all cache-induced delays but discuss them later explicitly. The benchmark-wide reduction becomes more evident, if we look at the number of cycles where interrupts are blocked, which is the mean to synchronize the kernel with its interrupts. Especially the SSM+Alarm variant shines with a cumulative interrupt-blockade time that is 83 percent smaller than in the baseline system. This originates from a vastly reduced IRQ load as the base frequency of the timer interrupt can be reduced by a factor of 10 by the usage of static alarms.

If we look at individual interrupt-blockade intervals, which are the defining factor for interrupt latency in real-time systems, we see a significant improvement: When we only use the SSM, we reduce the average interrupt-blockade interval from 195 cycles to 138 cycles. With static alarms, which shortens the execution time of the timer ISR, the average time even drops to 41 cycles.

For synchronous system calls (see Figure 7.6 (b)) that do not lead to a thread dispatch, we see that the time spent in the kernel is about the same as for the specialized system calls (−76 %) and the OSEK-V machine (−79 %). This similarity is caused by the fact that the system-call specialization is able to statically eradicate many of the scheduler invocations that actually do not lead to a dispatch. However, if we look at system calls that lead to a dispatch, we see that OSEK-V has at least 75 percent benefit over the baseline if we compare the maxima. On average, this benefit even increases to over 90 percent and the average synchronous system call takes 12 cycles on an OSEK-V machine with SSM (Baseline: 135 cycles).

For executions of the timer ISR, which manages those alarms that are not handled by the static-alarm HW component, I measured the length of individual activations between interrupt entry and the `iret` instruction. Here, the system-call specialization has only a minor influence on the cycle count. When no thread dispatch happens, the SSM variant shows only a minor average improvement (−8 %) over the baseline as the ISR invokes the scheduler only once if the ready list is modified. However, if a rescheduling is necessary, the SSM variant executes about as twice as fast in the worst case (−47 %). Furthermore, it significantly reduces the number of cache stalls (−72 %) which directly translates to fewer evicted application cache lines.

In the SSM+Alarms variant, several things change for the timer ISRs: Since we offload two of three alarms to the hardware, we see *Static Alarm* activations only in this variant. Furthermore, the number of timer IRQs drops from 280 to 28, since the generator could reduce the base timer rate for the remaining dynamic alarm. This is also the driving factor behind the observed reduction in the cumulative interrupt-blockade times (see Figure 7.6). While each timer ISR execution takes about as long as in the pure SSM variant, we see that static-alarm activations execute much faster and take at most 10 cycles.

For the cache stalls (Figure 7.6 (c)), we see that the Specialized variant leads to more cache stalls (+9 %), since different system-call site share less code due to their specialized implementations. For SSM, we see a decrease by 6 percent, which grows to 47 percent if we use static alarms. In this SSM+Alarm variant, cache evictions only arise from the dynamic handling of alarms and from the three instructions in each system-call site.[15]

7.4.4 Discussion

In the evaluation, OSEK-V showed some unique non-functional properties: Enabled by the application-specific tailoring, it exhibits short system-call execution times, minimal interference with the application's machine state, and much shorter interrupt-blockade times. However, this narrowed focus and the need for specialized hardware comes at a cost and also puts a limit to its applicability.

predictable RTOS implementation

As we have seen in Chapter 4, we must consider the influence of the RTOS for the WCRT analysis. Thereby, compositional, as well as an integrated, WCRT analyses profit from an execution platform that provides tight timing bounds, as they simplify the provisioning of a timing-predictable system. Thereby, OSEK-V has similar goals as the T-CREST/Patmos project [Sch+15], but on a higher abstraction level. Where T-CREST aims for a predictable processor architecture, OSEK-V provides a more predictable RTOS interface.

As we have seen in the evaluation, system calls on OSEK-V have minimal influence on the application. An SSM activation finishes in a few cycles, which are dominated by the instruction fetch (cache-stall cycle) if we reschedule to another thread. Thereby, the SSM execution itself does not evict any cache lines but only flushes the pipeline once. With static alarms, we can offload even more RTOS chores, which reduces the impact of each ISR execution and the interrupt load itself.

While timing predictability is a first-class design goal for RTCS, embedded devices are also challenged by security concerns, especially when they are connected to the public internet. Since the RTOS abstraction, provided by an OSEK-V core, exposes only the interaction that is actually required by the application, we cut down on the trusted code base. Furthermore, I believe that OSEK-V is inherently more trustworthy than an RTOS implementation on a general-purpose processor: First,

---

[15] disable interrupts, `ssm.tx`, enable interrupts

the computational substrate that exhibits the application-requested behavior is of a less powerful computation model (finite-state machine vs. Turing machine with sufficient memory). Furthermore, the solidification as hardware component makes OSEK-V less vulnerable to tampering at run-time.

In future research, a combination of OSEK-V and tailored memory protection could lead to perfect isolation between threads. Without executing a single instruction, the OSEK-V core would also switch protection domains on every hart switch.

specialization vs. standardization

OSEK-V targets an application scenario where an FPGA is already employed or where the use of an ASIC is intended. This high dependence on application-specific hardware is in stark contrast to the general industry trend that reduces HW/SW development costs by using high-volume common-off-the-shelve computing platforms.

However, there are two reasons why application-specific hardware designs, like OSEK-V, can become more and more practical in the future. First, we already see how the increasing automation degree in the hardware-design process leads to drastically reduced equipment and per-unit costs for manufacturing custom chips. As Patterson and Nikolić outlined in an EETimes blog post [PN15], an "ASIC on demand" industry is currently emerging and already provides small batches of custom chips at reasonable costs. The second aspect is the high automation degree of the OSEK-V generation itself. After we have implemented the interaction analysis and modified the chip generator, we can produce OSEK-V cores, without much manual work, for every application that is developed against a standard RTOS interface. Synthesizing an OSEK-V core is, from a developer's perspective, in the same league as compiling a software-implemented RTOS.

updatability

The OSEK-V approach is only applicable if the inflexibility of application-specific chips is tolerable: Produced as an ASIC, an OSEK-V core cannot be updated but it must be replaced if the interaction between application and RTOS changes. However, as the application itself remains a software component, it can be changed and updated as long as the location of system calls remains fixed. More precisely, an OSEK-V core that was manufactured for one application can be used for all applications whose ASMs are a subgraph of the original ones and whose real-time properties are compatible. For an FPGA system, the situation is much easier since we can treat the updated OSEK-V core as a part of the new system image.

If a full OSEK-V system is too inflexible, we can still fall back to hybrid scheduling approach and push down only parts of the interaction into the hardware. For such a hybrid scheme, we map only the high-priority threads to their own harts and use software-based scheduling in a single low-priority hart for all other threads. Thereby, we could still provide low latencies for the high-priority threads while threads can interact fully flexible in the low-priority hart.

Besides the real-time domain, application-specific chips are also a good fit for the emerging Internet-of-Things field. With this trend, small control systems are sold in large numbers and become ubiquitous, which renegotiates the trade-off between specialization and flexibility, as they strive a high price pressure and are tightly coupled to the chore and the life span of the employed device.

Taking a step back, we can see that the degree of updatability is another non-functional property of a system. The more behavior we fixate in hardware, the more we exchange updatability (and flexibility) for improved performance and a more resilient execution environment. Like for every non-functional property, there is a trade-off between both ends and we have to choose the optimal point in the design space according to the given application requirements. With OSEK-V, I provide more design-space points between a pure software solution and a pure hardware solutions, where the whole application is fixated in hardware. For RTOSs, the presented approach surely marks one of the far ends of the investigated trade-off.

scalability

We have already discussed the state-explosion problem of the SSTG in Section 3.4.3. Since OSEK-V depends on an SSTG, instead of only a GCFG, this problem also arises for the SSM. While ASMs (Section 3.4.3.2), real-time constraints (Section 3.4.3.1), and the SSM minimization ease the problem, the construction time and the hardware costs can be infeasible for large systems. Nevertheless, the evaluation showed that we can produce OSEK-V cores in reasonable time for a real-world scenario.

On the hardware-cost side, the scalability is determined by the size of the SSM component and the number of additional hart contexts. For every mapped RTOS thread, we must allocate the storage capacity for a full register file. This scales linearly with the number of harts, and we could use the FPGA's block memory to avoid the usage of distributed memory (Memory-LUTs for Xilinx). The SSM component grows with the application complexity and the degree of environmental indeterminism. However, this also means that the SSM for a small system, like we have seen with our running example, results in a small OSEK-V core.

7.5 Summary

Traditionally, the coordinating service of the RTOS is implemented in software and executes on the same processor as the application. This co-location of kernel and application introduces latencies and interference between both as the hardware is shared and must be used in a time-multiplexed manner. However, the application only depends on the RTOS interface and the correct behavior of the system software, as we have seen in Chapter 5; how the RTOS implements this behavior is insignificant.

In a HW/SW co-design setting, we have the chance to push (parts of) the kernel into the hardware. Instead of implementing a general-purpose RTOS with hardware means, my proposed OSEK-V approach pushes the interaction model into a hardware component: the *system-state machine (SSM)*. This component is a state-machine–based petrifaction of the SSTG and shows the same *behavior* as the regular kernel if executed together with the specific application. By using the interaction model, the SSM is not derived from the code structure of a software RTOS, but only from the requested interaction between RTOS and application. Thereby, I can avoid situations, that often arise with software-implemented RTOSs, where the system software provides more flexibility, services, and specified behavior than the application will request at run-time. This oversupply in services leads to unnecessary overheads and increases the security-relevant attack surface of the privileged RTOS code.

On a technical level, OSEK-V instantiates the SSM, which is a minimized version of the SSTG, besides the processor pipeline. Each RTOS thread gets mapped to a hardware thread and the SSM controls the hardware-thread scheduling. Upon the invocation of a system call, the SSM transitions and outputs the currently running thread, which induces hardware-thread switches if necessary. Furthermore, the SSM can be triggered by a static-alarm component, which offloads periodic thread activations to another hardware component.

In the evaluation with the *i4*Copter, I could show significant improvements in the different non-functional system properties, like low event latencies (—79 % average IRQ lock times), interference-reduced RTOS execution (-47 % cache stalls in the kernel), and fast thread re-scheduling (—81 % cycles for dispatching system calls). These improvements come at moderate FPGA cost of 9.8 percent more LUTs and 86 distributed memory cells per mapped RTOS thread.

research question

With OSEK-V, I could show that the integrated view on the whole RTCS, which is provided by the detailed interaction analysis, allows for an in-depth RTOS tailoring that utilizes the strength of the hardware domain: true parallelism. With these tailored systems, which only exhibit the RTOS behavior for a single application, I can provide an answer to my third research question (RQ3) about the improvement of non-functional properties: The push-down of the RTOS–application interaction into hardware exhibits significantly improvements in kernel-execution times, latencies, and interferences, and thereby the OSEK-V platform provides a more predictable execution environment for event-based real-time systems.

With OSEK-V, I come back to my initial problem analysis of mismatch between real-time and the implementation domain from Chapter 3. The interactions from the real-time domain are mapped to the RTOS interface in the implementation phase in order to encode the desired relationship between different tasks. With the interaction analysis, I extract these intended interactions, as well as all other interactions that arise from the implementation. The derived SSM, and therefore also the application-specific OSEK-V core, exhibits exactly that behavior that is desired by the application.

8

Summary, Conclusions, and Further Ideas

> There is no real ending. It's just the place where you stop the story.
>
> FRANK HERBERT, 1969

In this thesis, I investigated on the usage and the benefits of control-flow–sensitive interaction analyses for the characterization and optimization of real-time computer systems. The interaction between real-time operating system and application does not only implement the desired relationship between the system's entities, like threads and interrupts, but it also determines the system-wide execution sequences on the actual processor unit. The overall goal was to show that a whole-system view on the application's execution paths is feasible and that it provides significant insights that foster the understanding of the application requirements, which enables us to improve on important system properties.

8.1 Summary of the Thesis

generality costs

This thesis started with the argument that the emergence of cheap general-purpose computers pushed engineering from the construction of problem-specific solutions, like it is still done in mechanical engineering, towards a one-size-fits-all mentality. Due to the high flexibility and the reusability of software components, companies were enabled to reduce development costs and unveil whole new market segments. However, with the million-fold deployment of one solution, like it is the case in the automotive industry, the overheads of general-purpose solutions pile up but provide no benefit for the specific instance of a product.

One area where this over provisioning of flexibility came to my attention is the boundary between *real-time operating system (RTOS)* and real-time application. At this interface, the solution, which the real-time engineer developed in the real-time domain, comes into contact with the reality of the implementation level. There, the developer is able to choose from different mappings onto the RTOS interface, since the flexibility of the interface allows for different projections of the application requirements (see Section 2.2.1). Motivated by the prospect of improved efficiency and performance, or forced by engineering necessities, developer use this opportunity and adapt and bend the application code to their needs. While this situation not only calls for an automated validation that the real-time guarantees still hold on the implementation level, we can also ask the question if the deployed RTOS provides more flexibility than what the specific application actually requires.

interaction model

Both problems, over provisioning and post-mapping validation, materialize at the RTOS interface, since application and kernel interact with each other there: The application uses RTOS services, like job control, and the RTOS coordinates the timely execution of all activities in the system. Therefore, I use control-flow–sensitive interaction analysis as the means to better understand the structure and the requirements of a given real-time application. While the interaction analysis requires higher efforts in the offline phase, it provides, due to its system-wide path sensitivity, a precise picture of the potential behavior of the *real-time computing system (RTCS)*.

With the resulting in-depth *interaction model*, we can grasp the requirements of the whole application instead of considering only individual threads or an idealistic model of the indented behavior. With its control-flow sensitivity, I can further increase the specificity of this view as I consider only the possible system-call sequences, which are imposed by the application logic. Thereby, the RTOS configuration and its interface becomes a mere markup language that the developer uses to express the relationship between threads, interrupts, locks, and signal variables.

integrated WCRT analysis

As first application of the interaction knowledge, I presented SysWCET in Chapter 4, which is an integrated method to solve the *worst-case response time (WCRT)* problem. In order to find an upper bound for the time the RTCS takes to complete a job, I lift the *implicit path-enumeration technique (IPET)*, a WCET analysis technique, to the system level: The interaction-analysis result, in form of the *static state-transition graph (SSTG)*, contains all possible execution paths through the system and covers the application code, as well as all kernel and interrupt-handler code. At the nodes of the SSTG, which capture the system state at one point in time, I virtually attach all machine-code blocks of the implementation level such that no executed instruction on the WCRT path can escape my analysis. From this, I use the *implicit path-enumeration technique (IPET)* to construct an *integer linear programming (ILP)* that provides an integrated view on the system-wide execution paths, including thread–thread and thread–ISR transitions. Thereby, schedule and machine-level information are available in the same problem formulation, which brings inter-thread control-flow sensitivity and anchor points for additional fine-grained system-wide flow facts into the WCRT analysis. Together, this provides the means for implementation-level validation of the response-time analysis and fosters tighter response-time bound estimates. Summarized, the SysWCET approach to WCRT analysis provides an integrated view on the system-wide execution paths and is able to consider thread-crossing control-flow constraints that originate from the interaction model.

RTOS verification

Besides the timing validation, the logical correctness of the implementation also depends on the correct behavior of all system components, including the RTOS. Although a general verification of the RTOS implementation can tackle this problem and provide the necessary guarantees, it makes the employment of optimizing system generators problematic: Since every RTOS instance becomes specific for the given application, which is desirable to cut down on the overheads, we would have to verify the, often complex, system generator. Instead, I propose a method for an automated per-instance verification of kernel binaries in Chapter 5: First, a model checker ensures that the interaction model was constructed according to the specification and contains only valid system paths for a given application and system configuration. Second, a mock-up application probes the actual state space of the kernel binary on the final hardware platform, providing a high degree of implementation-level specificity. If both, the statically-calculated interaction model and the dynamically-explored state space, are equivalent, we know that the kernel binary behaves according to the specification. Summarized, the automatic per-instance verification of kernel binaries connects the model checking of specified RTOS properties, over the interaction model, to the actually exhibited behavior of the implementation.

fine-grained stack sharing

Another aspect of implementation-level validation is memory consumption: all memory-consumption sources have to be bounded and their combined maximal memory demand must remain below the calculated space reservations; otherwise memory corruptions can occur and undermine the correctness of the whole system. One class of memory requests are targeted at the execution stacks, where each function call allocates a new stack frame. While the calculation of an upper bound for a private thread-local stack can be done in relative isolation, the co-location of multiple threads onto the same shared stack, which is a technique to preserve memory, calls for a whole-system view in order to reach tight upper bounds. Therefore, I developed a *worst-case stack consumption (WCSC)* analysis in Chapter 6, on base of the IPET, that is able to cover call-graphs from multiple threads and fine-grained preemption constraints. One source of such constraints is the interaction model: Since it covers all possible preemptions, we can determine all preemptions that are possible on the shared stack. With the tighter bounds, we not only get the guarantee that no stack overflow can occur at run time but we can also reduce the static stack-space reservation, yielding a smaller memory consumption of the whole system.

Together with this improved WCSC analysis, I developed the *semi-extended task (SET)* concept to optimize the memory consumption even further: One obstacle for the usage of shared stacks are threads that wait passively for an event at some point instead of executing in a run-to-completion manner. If such a blocking thread would sleep on the shared stack, it can corrupt overlying stack frames of lower-priority threads if it wakes up. In order to avoid this situation but still utilize a shared stack, SETs start their execution on a private stack, but transfer their control flow onto the shared stack in phases where they surely never wait passively. Integrated with my WCSC analysis and with a genetic optimization procedure, SETs decrease a system's memory consumption without changing the application structure or semantic. After employing SETs, these systems exhibit lower stack usages and are now closer to the application's actual requirements with respect to stack-space usage.

RTOS hardware integration

Another source of inefficiencies that is related to the usage of an RTOS is the kernel's execution time and its negative influence on the machine state (i.e., additional cache misses). In the interaction analysis, I interpret the RTOS implementation as a state-machine transaction function that gets activated by a system call (or by an interrupt), modifies its internal state, and produces an output in form of a thread dispatch. Traditionally, the RTOS transitions are implemented in software and execute on the same processor as the application, leading to interferences with the actual workload and purpose of the system. In order to improve the situation, I propose the OSEK-V method in Chapter 6, which implements the extracted state-machine interpretation of the RTOS as a hardware component besides the processor pipeline. For this, I provide the tooling to include a condensed variant of the SSTG, the *system-state machine (SSM)*, into a RISC-V processor, which thereby becomes highly application specific as it only exposes the specified RTOS behavior in presence of one application. The SSM component receives requests from the execution stage of the pipeline, modifies its state register, and selects the now running thread for execution on the pipeline. In order to minimize the thread-switch overheads, I map every RTOS thread to its own hardware thread with a separate register file, such that a dispatch boils down to a pipeline flush-and-reload operation. With these measures, SSM offloading and the usage of hardware threads, an application-specific OSEK-V platform captures the required RTOS behavior more precisely than a software-implemented kernel and exhibits unique non-functional properties.

8.2 Further Ideas

Looking at the four main chapters of this thesis, I see several directions of future research that have the potential to further improve the state-of-the-art. In the following, I want to outline three concrete ideas that are directly related to and/or based on the projects from this thesis.

Implicit State Transition Graph. In the Chapter 4, we discussed how the (task-local) execution-time analysis progressed from structural analysis, over path-based methods, to the implicit-path-enumeration technique. With the transition from the compositional WCRT analysis to SysWCET, I took the first step for the response-time analysis in this direction. However, the SSE analysis explicitly enumerates all possible system states and, therefore, is, in the worst case, of exponential complexity. Therefore, it could be beneficial to take the second step for the whole-system view and make the SSTG also implicit. In such an implicit interaction model, we would combine the application logic, the system configuration, and the RTOS semantic without enumerating all possible states. Instead we would have to use symbolic combination rules to express the interactions between the system's entities. Such an implicit interaction model that is still specific to a single application would not only help with the WCRT analysis, but it would also foster the other methods from this thesis.

Worst-Case Heap Consumption. In the WCSC analysis, we tracked the allocation and deallocation of call frames, which are tightly coupled to the hierarchical nature of a thread's call sequence. However, not only the stack-space reservation, but also the heap contributes to the static memory consumption. By allocating objects from the heap, an application can communicate information cross-tree and across thread boundaries. From an object's allocation side, its references propagate along the global data flow until the object gets deallocated, potentially in another thread. In order to find the worst-case heap consumption, we would have to find that location in the global control-flow where the heap size is maximal. However, unlike the tracking of stack frames, the heap exposes the additional challenge of fragmentation. While this is challenging for worst-case analysis, it also includes the potential to calculate an application-specific optimal placement strategy that minimizes the required heap size.

Reconfigurable Variant of OSEK-V. OSEK-V surely states a far end of the static-tailoring method for RTOSs. Here the kernel behavior is fixated, unchangeable, in the organization of gates and flip-flops. However, a more flexible variant of the OSEK-V processor is thinkable. In this fictitious processor, some aspects, like the number of hardware threads, would still be fixed but we could reconfigure the SSM component such that it implements the RTOS behavior for a SSTG. In principle, we would build a processor that exposes complex and explicit control over the hardware multi-threading to the operating system.

8.3 Research Questions

With my contributions, I am now able to provide answers to my stated research questions that arise from the identified problems and the application of the interaction-aware take on real-time–system analysis and optimization. In the following, I want to give my answers to these questions.

Research Question 1 *Is a control-flow sensitive view on the RTOS–application interaction feasible for whole-system analysis?*

The first question is related to the feasibility of my chosen approach for a with range of systems, including realistically-sized application. With a look at the evaluation section of the four previous chapters, we see how the prototypical implementation of the *system-state enumeration (SSE)* and the *system-state flow (SSF)* analyses were able to calculate interaction models for systems of various sizes over the whole range of the specified OSEK semantic. The extracted interaction models were detailed enough to enable different whole-system validations and optimizations.

Furthermore, I consider the systematic evaluation of the SET concept in Section 6.4 as the strongest argument to answer this research question positively. There, I generated more than 14 000 synthetic benchmark systems with varying system properties (e.g., number of threads and interrupts) and ran the SSF interaction analysis in order to extract fine-grained preemption constraints. With this broad investigation, I could not only demonstrate the feasibility but also the scalability of my interaction-aware approach.

Research Question 2 *What analysis and run-time inefficiencies arise in the real-time application from its segregation into distinct execution threads?*

In Chapter 4, I demonstrated that the compositional WCRT analysis accumulates pessimism by first considering threads in isolation before combining the results of the WCET analysis according to the system configuration. With the interaction-aware take on these problems, I could tear down the thread boundaries and provide an integrated problem formulation for the WCRT analysis. All savings in the response-time estimates that I could achieve with SysWCET are inefficiencies in the analysis of real-time systems that our methods had in the past because we structured our analyses according to the implementation structure. However, the presence of an implementation structure does not necessarily mean that our analyses also have follow the same structure; they only have to consider it.

In Chapter 6, I applied the principle of an integrated view onto execution-stack allocations: Instead of seeing a thread's stack usage as an indivisible source of memory consumption, the simple stack-switch mechanism on the granularity of functions hinted us to look at individual stack-frame allocations. With the use of the interaction-aware preemption knowledge, my WCSC analysis calculates tighter bounds that we could not achieve by looking at every thread in isolation. So, every saving that the WCSC analysis achieves for a BTS-only system proofs that an isolated view on the stack consumption results in overly pessimistic stack-space reservations.

Even more, the introduction of SETs enabled threads that wait at some point in their execution to utilize the shared stack. Thereby, I constructively broke up the monolithic stack-allocation area and transferred parts of application that do not require a private stack onto the shared stack. I achieved this shrinkage of the memory usage *without* modifying the application's usage pattern, but I only adapted the run-time system to match the actual requirements of the applications. Thereby, I could show that not only the segregated analysis of the resource consumption results in overly pessimistic results, but also that the isolated consumption of resources itself yields run-time inefficiencies.

Research Question 3 *What beneficial non-functional RTOS properties can we achieve, and what statements about functional properties can we make, if we have an integrated view on the real-time computing system?*

With the application-specific verification of kernel binaries, I could show that an focus on the actually used and expected RTOS behaviors can help in the verification of a system's functional correctness. Instead of verifying an optimizing system generator, I proposed the per-system-image verification of the RTOS–application interactions that is based on model checking and dynamic probing of the implementation. With the SSTG interaction model as a connecting centerpiece between the semantic of the RTOS specification and the actually exposed state space of the kernel binary, I was able to verify the correct scheduling behavior of over hundred OSEK systems. Even more, this connecting centerpiece of the whole-system verification allows for the reuse of verification results: As long as the binary exposes the same reschedule sequence, the model-checking results can be reused, even if different implementation strategies are used.

Similar to this verification, but more on the constructive side, OSEK-V demonstrates that an execution platform that is only able to expose the specified RTOS behavior for a single application is sufficient to operate a system. Moreover, it even has beneficial non-functional system properties and I could demonstrate unique RTOS run-time properties, like low and predictable kernel-execution latencies and a minimal interference with the processor's hardware state. While these properties were achieved by the elimination of the partial interpretation that is typical for a software-implemented RTOS, the exact knowledge about the potential interactions is the enabling centerpiece of this approach.

8.4 Conclusion

While each of the four presented projects (Chapter 4-Chapter 7) is a contribution in itself, they are only particular aspects and indicators for my main contribution: With this thesis, I advance the state-of-the-art in the area of embedded real-time systems by the *integrated view on the RTOS–application interaction* and the detailed investigation on its costs, benefits, and potentials. Instead of separating kernel and application for the system analysis and optimization, I argue for a tool-based, automated, and interaction-aware whole-system methodology that is located on the implementation level. With the high degree of application knowledge that arises from the interaction analysis, I demonstrate how we can develop methods to build better systems that are closer to their actual purpose: An implementation's capabilities should be determined by its usage scenario, not by the features of the components that we use to build it. Instead of constructing a one-size-fits-all solution, this thesis provides a methodology to analyze and to build embedded systems that are specialized for those interactions that can actually happen after their deployment. By focusing on the implementation level, the benefits and guarantees of the interaction-aware methods are robust in the presence of engineering necessities and after stripping away those abstractions that are only necessary during the development phase.

Glossary

ABB atomic basic block **See:** Section 3.3.2
SESE region with a CFG that either contains only computation code or a single system call.

AbSS abstract system state **See:** Section 3.4.1
SSTG node that captures the state of an RTOS at one point in time. It, for example, contains the state of all threads in the system.

ASM application state machine **See:** Section 3.4.3.2
FSM that captures all possible system-call sequences that can arise from a thread or ISR.

AST abstract syntax tree
The syntax-derived representation of a program. This compiler-internal data structure is constructed by the parser and reflects the hierarchical nesting of language elements and the operator precedence.

BT basic task **See:** Section 2.1.3
OSEK thread type that is not allowed to enter a waiting state. Therefore, all BTs can be co-located on the same stack.

BTS basic-task stack **See:** Figure 6.2
The stack-space reservation in an OSEK system that is shared between all basic tasks.

CFG control-flow graph **See:** Section 3.3.1
A directed graph of basic blocks with one entry node. It covers all possible paths through a function or program.

COTS common-of-the-shelve
Describes those platforms and chips that are in wide use and easily accessible for a commercial enterprise.

CTL computation-tree logic **See:** Figure 5.2
Temporal logic that describes the progress of time as a tree where the each decision leads to an inner node and all possible outcomes are different children.

DAG directed acyclic graph
Graph where edges have a direction and that contains no cyclic path if we only follow edges in their defined direction.

DSTG dynamic state-transition graph **See:** Section 5.2.2
DAG-shaped interaction model that is extracted by execution and probing of a concrete kernel binary.

ECU electronic control unit
Term from the automotive industry that describes systems, like the engine control, that replaced the traditional analog control systems.

EDF earliest deadline first **See:** [LL73]
Common real-time scheduling policy that executes the jobs with the closest absolute deadline first.

ET extended task **See:** Section 2.1.3
OSEK thread type that has no restriction on its system-call usage and is therefore allowed to wait (on its private stack)

FSM finite-state machine
Automaton that has a finite number of states. Transitions between states are either labeled with consumed inputs and/or with produced output symbols.

GCFG global control-flow graph **See:** Section 3.5
Lifting of the CFG concept onto the system level. It contains all ABBs in the system and the control flow follows the rescheduling sequence of the RTOS

ICFG interprocedural control-flow graph **See:** Section 3.3.1
CFG that includes basic blocks from several function and that proceeds in the callee on a function invocation.

ILP integer linear programming **See:** Section 4.4
Class of mathematical optimization problems with one linear optimization objective and multiple constraints that restrict the possible solution space. All variables are integer typed.

IPET implicit path-enumeration technique **See:** Section 4.4
ILP-based method to calculate the WCET for a given program.

IR immediate representation
Compiler-internal representation of the program that is close to the machine code. Often, the IR uses a machine model with an infinite number of registers.

IRQ interrupt request **See:** Section 2.1.2.1
Hardware signal that is triggered asynchronously to the sequential program execution on the CPU. Often, these interrupts indicate the completion of an I/O operation or some exceptional situation.

ISA instruction-set architecture
The processor-provided interface that consists of instructions, registers, and execution semantic.

ISR interrupt-service routine **See:** Section 2.1.2.1
Function that is invoked upon an IRQ. Normally, the processor forces the ISR activation upon the currently program execution.

LTL linear temporal logic
Temporal logic that is similar, but not equal to, CTL.

MCU microcontroller unit
A small system that includes an, often small, processor, memory, and simple peripheral device, like an GPIO port or an SPI-bus interface.

MMU memory-management unit **See:** Section 6.4.5
Hardware unit that provides page-grained memory virtualization and protection and that is configured with a page directory.

MPU memory-protection unit **See:** Section 6.4.5
Hardware unit that provides fine-grained memory protection and that is configured by writing range registers.

OIL OSEK implementation language **See:** Section 2.1.3
Configuration language for OSEK systems that is used to statically declare all system objects, like threads or events, ahead of time. The OIL configuration is, besides the application code, the second input to every OSEK system generator.

PABB power atomic basic block **See:** Section 4.7.2
Refinement of the ABB concept such that also the power-state of processor and peripheral devices remains constant in the SESE region.

PCP priority ceiling protocol **See:** Section 2.1.3, [SRL90a]
Deadlock-free locking protocol where all potential acquirers have to be known at acquisition time.

PLA programmable logic array **See:** Figure 7.3
Hardware component with configurable rows of configurable AND gates. Each row consists of a pattern with don't-care terms that is matched against the input, and an output mask that is OR'ed to the result if the pattern matches.

PTS preemption-threshold scheduling **See:** Section 6.2
Extension to a fixed-priority scheduling policy, where each activity uses one priority to preempt other threads and another priority, the threshold, to defend itself against preemption.

RT real time **See:** Section 2.1
The physical time. Often this term describes that some action and its completion is linked to the physical time.

RTA real-time application **See:** Section 2.1
An program that must be executed under real-time constraints.

RTCS real-time computing system **See:** Section 2.1
A RTS that is implemented with a computer.

RTOS real-time operating system **See:** Section 2.1
Specialized operating-systems that provide strict guarantees on their semantic and timeliness.

RTS real-time system **See:** Section 2.1
A system that must perform an operation upon some external event in an bounded time span.

SESE single-entry–single-exit **See:** Section 3.3.2
CFG region of basic blocks that can only be entered via one block (or edge) and that can also only left via a single block (or edge).

SET semi-extended task **See:** Chapter 6
Hybrid concept between BT and ET that executes on the shared stack and is allowed to wait passively.

SRP stack-based resource protocol **See:** Section 2.1.3, [Bak91]
PCP variant where the priority is raised to the ceiling priority immediately at acquisition time.

SSE system-state enumeration **See:** Section 3.4.2
Method to calculate the SSTG for a given combination of RTOS and application.

SSF system-state flow **See:** Section 3.5.2
Fast method, based on a fixpoint data-flow analysis, to calculate the GCFG

SSM system-state machine **See:** Section 7.2
Condensed form of the SSTG that is prepared for hardware integration. It only exposes the correct thread–thread rescheduling sequence instead of distinguishing between the fine-grained application states

SSTG static state-transition graph **See:** Section 3.4
DAG-shaped interaction model that captures all possible RTOS states (in AbSSs) that can arise for a given application

TMR triple modular redundancy
Method to increase the resilience against (transient) hardware faults. An operation is executed three times with the same input and a voter performs a majority vote on the results.

WCEC worst-case energy consumption **See:** Section 4.7
The highest amount of energy that can spent for a task if it is executed in isolation.

WCET worst-case execution time **See:** Section 2.1.1
The longest duration for which a task can execute if run in isolation.

WCRE worst-case response energy **See:** Section 4.7
The highest amount of energy that can spent for a task if it is executed in the context of a system.

WCRT worst-case response time **See:** Section 4.2
The longest duration for which a task can execute if run in the context of a whole system.

WCSC worst-case stack consumption **See:** Section 6.1
The maximal memory demand on an shared or private execution stack.

Bibliography

Own Articles

[▷Dei+17a] Hans-Peter Deifel, **Christian Dietrich**, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. “Automatic Verification of Application-Tailored OSEK Kernels.” In: *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. FMCAD ’17 (Vienna, Austria). New York, NY, USA: ACM Press, Nov. 2017, pp. 1–8. DOI: `10.23919/FMCAD.2017.8102260`.

[▷Dei+17b] Hans-Peter Deifel, **Christian Dietrich**, Merlin Göttlinger, Daniel Lohmann, Stefan Milius, and Lutz Schröder. *Automatic Verification of Application-Tailored OSEK Kernels*. Tech. rep. Extended version of [▷Dei+17a]. 2017. DOI: `10.15488/1761`.

[▷DHL15a] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Back to the Roots: Implementing the RTOS as a Specialized State Machine.” In: *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. OSPERT ’15 (Lund, Sweden). July 2015, pp. 7–12. URL: `http://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p7.pdf`.

[▷DHL15b] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems.” In: *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES ’15 (Portland, Oregon, USA). New York, NY, USA: ACM Press, June 2015. ISBN: 978-1-4503-3257-6. DOI: `10.1145/2670529.2754963`.

[▷DHL17] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. “Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis.” In: *ACM Transactions on Embedded Computing Systems* 16.2 (2017), 35:1–35:25. DOI: `10.1145/2950053`.

[▷Die+17] **Christian Dietrich**, Peter Wägemann, Peter Ulbrich, and Daniel Lohmann. “SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems.” In: *Proceedings of the 23rd IEEE International Symposium on Real-Time and Embedded Technology and Applications*. RTAS ’17 (Pittsburgh, Pennsylvania, USA). Washington, DC, USA: IEEE Computer Society Press, 2017, pp. 37–48. ISBN: 978-1-5090-5269-1. DOI: `10.1109/RTAS.2017.37`.

[▷Die14] **Christian Dietrich**. “Global Optimization of Non-Functional Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses.” Master’s Thesis. Department of Computer Science 4, Distributed Systems and Operating Systems; University of Erlangen-Nuremberg, 2014.

[▷DL17] **Christian Dietrich** and Daniel Lohmann. “OSEK-V: Application-Specific RTOS Instantiation in Hardware.” In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES ’17 (Barcelona, Spain). New York, NY, USA: ACM Press, June 2017. DOI: `10.1145/3078633.3078637`.

[▷Fie+18] Björn Fiedler, Gerion Entrup, **Christian Dietrich**, and Daniel Lohmann. “Levels of Specialization in Real-Time Operating Systems.” In: *Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT ’18)* (Barcelona, Spain). July 2018.

[▷Hof+14] Martin Hoffmann, Peter Ulbrich, **Christian Dietrich**, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “A Practitioner’s Guide to Software-based Soft-Error Mitigation Using AN-Codes.” In: *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering*. HASE ’14 (Miami, Florida, USA). IEEE Computer Society Press, Jan. 2014, pp. 33–40. ISBN: 978-1-4799-3465-2. DOI: `10.1109/HASE.2014.14`.

[▷Hof+15] Martin Hoffmann, Florian Lukas, **Christian Dietrich**, and Daniel Lohmann. “dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel.” In: *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications*. RTAS ’15 (Seattle, Washington, USA). Washington, DC, USA: IEEE Computer Society Press, 2015, pp. 259–270. DOI: `10.1109/RTAS.2015.7108449`.

[▷Wäg+18] Peter Wägemann, **Christian Dietrich**, Tobias Distler, Peter Ulbrich, and Wolfgang Schröder-Preikschat. “Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems.” In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems*. ECRTS ’18 (Barcelona, Spain). Ed. by Sebastian Altmeyer. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. DOI: `10.4230/LIPIcs.ECRTS.2018.24`.

Related Work

[01] *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture*. Mar. 2001.

[04] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. Oct. 2004. ISBN: 0-9524156-2-3.

[08] *TriCore 1 User’s Manual (V1.3.8), Volume 1: Core Architecture*. Infineon Technologies AG. 81726 Munich, Germany, Jan. 2008.

[98] *Portable Operating System Interfaces (POSIX®) – Part 1: System Application Program Interface (API) – Amendment 1: Realtime Extension*. 1998.

[AA82] Sudhir R. Ahuja and Abhaya Asthana. “A Multi-microprocessor Architecture with Hardware Support for Communication and Scheduling.” In: *Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I)*. Palo Alto, California, USA: ACM Press, 1982, pp. 205–209. ISBN: 0-89791-066-4. DOI: `10.1145/800050.801844`.

[Abs19] AbsInt. *StackAnalyzer: Stack Usage Analysis*. 2019. URL: `https://www.absint.com/stackanalyzer/index.htm` (visited on 02/22/2019).

[AEE03] AEEC. *Avionics Application Software Standard Interface (ARINC Specification 653-1)*. ARINC Inc, 2003.

[AEe17] Aspencore, EETimes, and embedded.com. “2017 Embedded Markets Study.” In: (2017).

[Agr+04] Jason Agron, David Andrews, Mike Finley, E Komp, and W Peck. “FPGA implementation of a priority scheduler module.” In: *Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works in Progress Session (RTSS WIP), Lisbon, Portugal*. Citeseer. 2004.

[Agr+06] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. “Run-Time Services for Hybrid CPU/FPGA Systems on Chip.” In: *Proceedings of the 27th IEEE International Symposium on Real-Time Systems (RTSS ’06)*. 2006, pp. 3–12. DOI: `10.1109/RTSS.2006.45`.

[Aho+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. second. Boston, MA, USA: Addison-Wesley, 2007. ISBN: 0-321-49169-6.

[Akt17] Volkswagen Aktiengesellschaft. “Geschäftsbericht 2017.” In: (2017).

[All70] Frances E. Allen. “Control Flow Analysis.” In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: `10.1145/390013.808479`.

[Arn+14] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. “Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs.” In: *ACM Transactions on Embedded Computing Systems* 13.3s (Mar. 2014), 107:1–107:24. ISSN: 1539-9087. DOI: `10.1145/2517087`.

[AS99] Tarek F. Abdelzaher and Kang G. Shin. “Combined Task and Message Scheduling in Distributed Real-Time Systems.” In: *IEEE Transactions on Parallel and Distributed Systems* 10.11 (1999), pp. 1179–1191. ISSN: 1045-9219. DOI: `10.1109/71.809575`.

[Asa+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin andChristopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz andSagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love andMartin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. University of California Berkley, Electrical Engineering and Computer Sciences Department, 2016.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[ATB93] Neil C Audsley, Ken Tindell, and Alan Burns. “The end of the line for static cyclic scheduling?” In: *RTS*. 1993, pp. 36–41.

[Aud+91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. “Hard Real-Time Scheduling: The Deadline Monotonic Approach.” In: *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*. Atalanta: IEEE Computer Society Press, 1991.

[Aud+92] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. “Deadline Monotonic Scheduling Theory.” In: *Real-Time Programming 1992*. Ed. by L. Boullart and J.A. De La Puente. IFAC Postprint Volume. Oxford: Pergamon, 1992, pp. 55–60. ISBN: 978-0-08-041894-0. DOI: `10.1016/B978-0-08-041894-0.50012-5`.

[Aud+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. “Applying new scheduling theory to static priority pre-emptive scheduling.” In: *Software Engineering Journal* 8.5 (Sept. 1993), pp. 284–292. ISSN: 0268-6961.

[Aud93] Neil C Audsley. “Flexible scheduling of hard real-time systems.” PhD thesis. University of York, 1993.

[AUT13] AUTOSAR. *Specification of Operating System (Version 5.1.0)*. Tech. rep. Automotive Open System Architecture GbR, Feb. 2013.

[Awa88] IEEE Computer Pioneer Award. *Friedrich L. Bauer - For computer stacks*. `https://www.computer.org/web/awards/pioneer-friedrich-bauer`. 1988.

[BA10] Björn B. Brandenburg and James H. Anderson. "Optimality Results for Multiprocessor Real-Time Locking." In: *Proceedings of the 31st IEEE International Symposium on Real-Time Systems (RTSS '10)* (San Diego, CA, USA, Nov. 30, 2010–Dec. 3, 2010). Washington, DC, USA: IEEE Computer Society Press, Dec. 2010, pp. 49–60. ISBN: 978-0-7695-4298-0. DOI: `10.1109/RTSS.2010.17`.

[Bab64] Charles Babbage. *Passages from the Life of a Philosopher*. Ed. by Martin Campbell-Kelly. New Brunswick, NJ, USA: Rutgers University Press, 1864. ISBN: 0813520665.

[Bak91] Theodore P. Baker. "Stack-based Scheduling for Realtime Processes." In: *Real-Time Systems Journal* 3.1 (Apr. 1991), pp. 67–99. ISSN: 0922-6443. DOI: `10.1007/BF00365393`.

[Bar02] Volker Barthelmann. "Inter-Task Register-Allocation for Static Operating Systems." In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES '02)* (Berlin, Germany). New York, NY, USA: ACM Press, 2002, pp. 149–154. ISBN: 1-58113-527-0. DOI: `10.1145/513829.513855`.

[Bar10] Sanjoy Baruah. "The non-cyclic recurring real-time task model." In: *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, pp. 173–182.

[Bat45] Harry Bateman. "The control of an elastic fluid." In: *Bulletin of the American Mathematical Society* 51.9 (1945), pp. 601–646.

[BB05] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests." In: *Real-Time Systems* 30.1 (May 2005), pp. 129–154. ISSN: 1573-1383. DOI: `10.1007/s11241-005-0507-9`. URL: `https://doi.org/10.1007/s11241-005-0507-9`.

[BB97] Iain Bate and Alan Burns. "Schedulability analysis of fixed priority real-time systems with offsets." In: *Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on*. IEEE. 1997, pp. 153–160.

[BDP01] Dennis Brylow, Niels Damgaard, and Jens Palsberg. "Static checking of interrupt-driven software." In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE. 2001, pp. 47–56.

[Beh+03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services." In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 268–281. ISBN: 1-58113-757-5. DOI: `10.1145/945445.945471`.

[Ber+06] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. "Building a Global System View for Optimization Purposes." In: *Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)* (Boston, USA). Washington, DC, USA: IEEE Computer Society Press, June 2006.

[Bev89] William R. Bevier. “Kit: A study in operating system verification.” In: *IEEE Transactions on Software Engineering* 15.11 (1989), pp. 1382–1396.

[BH13] Bernard Blackham and Gernot Heiser. “Sequoll: A framework for model checking binaries.” In: *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS ’13)*. 2013, pp. 97–106.

[Bla+11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. “Timing analysis of a protected operating system kernel.” In: *Proceedings of the 32th Real-Time Systems Symposium (RTSS ’11)*. 2011, pp. 339–348.

[BLH14] Bernard Blackham, Mark Liffiton, and Gernot Heiser. “Trickle: Automated Infeasible Path Detection Using All Minimal Unsatisfiable Subsets.” In: *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS ’14)*. 2014, pp. 169–178.

[BLM17] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers, 2017. ISBN: 9781627056021.

[Boh+08] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. “Bounding Shared-Stack Usage in Systems with Offsets and Precedences.” In: *2008 Euromicro Conference on Real-Time Systems*. July 2008, pp. 276–285. DOI: `10.1109/ECRTS.2008.29`.

[BRA09] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. “Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions.” 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 für Informatik, 2009, pp. 1–11. ISBN: 978-3-939897-14-9. DOI: `10.4230/OASIcs.WCET.2009.2285`. URL: `http://drops.dagstuhl.de/opus/volltexte/2009/2285`.

[Bra11] Björn B. Brandenburg. “Scheduling and Locking in Multiprocessor Real-Time Operating Systems.” PhD thesis. The University of North Carolina at Chapel Hill, 2011. URL: `http://www.cs.unc.edu/~bbb/diss/`.

[Bro06] Manfred Broy. “Challenges in Automotive Software Engineering.” In: *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)* (Shanghai, China). New York, NY, USA: ACM Press, 2006, pp. 33–42. ISBN: 1-59593-375-1. DOI: `10.1145/1134285.1134292`.

[Bry03] Dennis Brylow. “Static Checking of Interrupt Driven Software.” PhD thesis. Purdue University, Aug. 2003. URL: `http://www.mscs.mu.edu/~brylow/papers/Brylow-Dissertation2003.pdf`.

[BS57] Friedrich Ludwig Bauer and Klaus Samelson. “Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens.” Auslegeschrift 1094019, B 44122 IX/42m. 1957.

[BS88] Theodore P. Baker and Alan C. Shaw. “The Cyclic Executive Model and Ada.” In: *Proceedings of the 9th IEEE International Symposium on Real-Time Systems (RTSS ’88)*. IEEE Computer Society Press, 1988, pp. 120–129.

[Bur+99] Wayne P. Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles C. Weems. "The Spring Scheduling Coprocessor: A Scheduling Accelerator." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 38–47. DOI: `10.1109/92.748199`.

[BW73] Daniel G. Bobrow and Ben Wegbreit. "A Model and Stack Implementation of Multiple Environments." In: *Commun. ACM* 16.10 (Oct. 1973), pp. 591–603. ISSN: 0001-0782. DOI: `10.1145/362375.362379`.

[Car+02] Martin Carlsson, Jakob Engblom, Andreas Ermedahl, Jan Lindblad, and Björn Lisper. "Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system." In: *Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*. 2002.

[Car93] BE Carpenter. "Turing and ACE: lessons from a 1946 computer design." In: (1993).

[CB02] A. Colin and G. Bernat. "Scope-tree: a program representation for symbolic worst-case execution time analysis." In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. June 2002, pp. 50–59. DOI: `10.1109/EMRTS.2002.1019185`.

[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`.

[Cel+73] J. Celtruda, W. Crosthwait, J. Earle, J. Fennel, and R. Henderson. *Apparatus and method for serializing instructions from two independent instruction streams*. US Patent 3,771,138. Nov. 1973.

[CH94] Jyh-Herng Chow and W. L. Harrison. "State space reduction in abstract interpretation of parallel programs." In: *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. May 1994, pp. 277–288. DOI: `10.1109/ICCL.1994.288373`.

[Cha+03] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A Henzinger, and Jens Palsberg. "Stack size analysis for interrupt-driven programs." In: *International Static Analysis Symposium*. Springer. 2003, pp. 109–126.

[Cha09] Robert N. Charette. *This Car Runs on Code*. `https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`, accessed 29.10.2018. Feb. 2009.

[Chi+15] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine." In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey). ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 117–130. ISBN: 978-1-4503-2835-7. DOI: `10.1145/2694344.2694367`. URL: `http://doi.acm.org/10.1145/2694344.2694367` (visited on 02/21/2019).

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "A simple, fast dominance algorithm." In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8.

[Cim+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. “NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking.” In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, July 2002.

[CK88] T. L. Casavant and J. G. Kuhl. “A taxonomy of scheduling in general-purpose distributed computing systems.” In: *IEEE Transactions on Software Engineering* 14.2 (1988), pp. 141–154. ISSN: 0098-5589. DOI: `10.1109/32.4634`.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. “Hardware Support for Fast Capability-based Addressing.” In: *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. San Jose, California, USA: ACM Press, 1994, pp. 319–327. ISBN: 0-89791-660-3. DOI: `10.1145/195473.195579`.

[CKL00] N. Chang, K. Kim, and H. G. Lee. “Cycle-accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI.” In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED ’00)*. 2000, pp. 185–190.

[CL90] Min-Ih Chen and Kwei-Jay Lin. “Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems.” In: *Real-Time Systems Journal* 2.4 (1990), pp. 325–346. DOI: `10.1007/BF01995676`.

[CMD62] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. “An experimental time-sharing system.” In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM. 1962, pp. 335–344.

[CMS07] Benoît Colson, Patrice Marcotte, and Gilles Savard. “An Overview of Bilevel Optimization.” In: *Annals of Operations Research* 153.1 (Sept. 1, 2007), pp. 235–256. ISSN: 1572-9338. DOI: `10.1007/s10479-007-0176-2`. URL: `https://doi.org/10.1007/s10479-007-0176-2` (visited on 03/07/2019).

[CP00] Antoine Colin and Isabelle Puaut. “Worst Case Execution Time Analysis for a Processor with Branch Prediction.” In: *Real-Time Systems* 18.2 (May 2000), pp. 249–274. ISSN: 1573-1383. DOI: `10.1023/A:1008149332687`. URL: `https://doi.org/10.1023/A:1008149332687`.

[CP01] Antoine Colin and Isabelle Puaut. “Worst-case execution time analysis of the RTEMS real-time operating system.” In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS ’01)*. 2001, pp. 191–198.

[CP09] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. “On non-functional requirements in software engineering.” In: *Conceptual modeling: Foundations and applications*. Springer, 2009, pp. 363–379.

[CPL16] IBM ILOG CPLEX. “V12.7: User’s Manual for CPLEX.” In: *International Business Machines Corporation* (2016), p. 586.

[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`.

[Dan+14] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. "Safer Sloth: Efficient, Hardware-Tailored Memory Protection." In: *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '14)*. Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 37–47.

[Dav13] Robert I. Davis. *Burns Standard Notation for Real Time Scheduling*. 2013.

[DB82] Douglas D. Dunlop and Victor R. Basili. "A Comparative Analysis of Functional Correctness." In: *ACM Computing Surveys* 14.2 (June 1982), pp. 229–244. ISSN: 0360-0300. DOI: `10.1145/356876.356881`. URL: `http://dx.doi.org/10.1145/356876.356881`.

[DD68] Robert C. Daley and Jack Bonnell Dennis. "Virtual Memory, Processes, and Sharing in MULTICS." In: *Communications of the ACM* 11.5 (May 1968), pp. 306–312. DOI: `10.1145/363095.363139`.

[Dev+88] S. Devadas, Hi-Keung Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. "MUSTANG: state assignment of finite state machines targeting multilevel logic implementations." In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 7.12 (Dec. 1988), pp. 1290–1300. ISSN: 0278-0070. DOI: `10.1109/43.16807`.

[DF04] Radu Dobrin and Gerhard Fohler. "Reducing the number of preemptions in fixed priority scheduling." In: *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*. IEEE. 2004, pp. 144–152.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki — a Lightweight and Flexible Operating System for Tiny Networked Sensors." In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, FL, USA, Nov. 2004.

[DHL15c] **Christian Dietrich**, Martin Hoffmann, and Daniel Lohmann. "Globale Kontrollflussanalyse von eingebetteten Echtzeitsystemen." In: *Betriebssysteme und Echtzeit, Echtzeit 2015, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V. (GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG) sowie der Fachgruppe Betriebssysteme von GI und ITG, Boppard, 12. und 13. November 2015*. 2015, pp. 128–136. DOI: `10.1007/978-3-662-48611-5_14`.

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[Dij65] Edsger Wybe Dijkstra. *Cooperating Sequential Processes*. Tech. rep. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). Eindhoven, The Netherlands: Technische Universiteit Eindhoven, 1965. URL: `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`.

[DL18] **Christian Dietrich** and Daniel Lohmann. "Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads." In: *Proceedings of the 39th IEEE Real-Time Systems Symposium 2018*. Ed. by Sebastian Altmeyer. Nashville, Tennessee, USA: IEEE Computer Society Press, 2018. DOI: `10.1109/RTSS.2018.00049`.

[DMT00] Robert Davis, Nick Merriam, and Nigel Tracey. “How Embedded Applications Using an RTOS Can Stay Within On-Chip Memory Limits.” In: *Proceedings of the Work in Progress and Industrial Experience Session of the 12th Euromicro Conference on Real-Time Systems (ECRTS-WiP ’00)*. 2000, pp. 43–50.

[DR05] N. Diniz and J. Rufino. “ARINC 652 in Space.” In: (2005).

[Dra+91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. “Using Continuations to Implement Thread Management and Communication in Operating Systems.” In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP ’91)* (Pacific Grove, CA, USA). New York, NY, USA: ACM Press, Sept. 1991, pp. 122–136. ISBN: 0-89791-447-3. DOI: `10.1145/121132.121155`.

[Dun+06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. “Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems.” In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. Boulder, Colorado, USA, Nov. 2006. URL: `http://dunkels.com/adam/dunkels06protothreads.pdf`.

[Edv99] Jon Edvardsson. “A survey on automatic test data generation.” In: *Proceedings of the 2nd Conference on Computer Science and Engineering*. 1999, pp. 21–28.

[Eic+18] Christian Eichler, Tobias Distler, Peter Ulbrich, Peter Wägemann, and Wolfgang Schröder-Preikschat. “TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses.” In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Ed. by Florian Brandner. Vol. 63. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, 6:1–6:12. ISBN: 978-3-95977-073-6. DOI: `10.4230/OASIcs.WCET.2018.6`. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/9752`.

[Eng02] Jakob Engblom. “Processor pipelines and static worst-case execution time analysis.” PhD thesis. Acta Universitatis Upsaliensis, 2002.

[Erm03] Andreas Ermedahl. “A modular tool architecture for worst-case execution time analysis.” PhD thesis. Acta Universitatis Upsaliensis, 2003.

[Evi12] Evidence. *ERIKA Enterprise Manual, version 1.4.4*. `http://download.tuxfamily.org/erika/webdownload/manuals_pdf/ee_refman_1_4_4.pdf`. 2012.

[Fen73] J Fennel. *Instruction selection in a two-program counter instruction unit*. US Patent 3,728,692. Apr. 1973.

[FN79] Richard J Feiertag and Peter G Neumann. “The foundations of a provably secure operating system (PSOS).” In: *Proceedings of the National Computer Conference*. Vol. 48. 1979, pp. 329–334.

[Fre13] Freescale Semiconductor, Inc. *KL46 Sub-Family Reference Manual*. 2013, pp. 1–932.

[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 (Mar. 1986), pp. 218–221. ISSN: 0001-0782. DOI: `10.1145/5666.5673`.

[FW99] Christian Ferdinand and Reinhard Wilhelm. “Efficient and precise cache behavior prediction for real-time systems.” In: *Real-Time Systems* 17.2-3 (1999), pp. 131–181.

[GD07] Rony Ghattas and Alexander G Dean. “Preemption threshold scheduling: Stack optimality, enhancements and analysis.” In: *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS’07. 13th IEEE*. IEEE. 2007, pp. 147–157.

[GLD01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. “Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip.” In: *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*. RTSS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 73–. ISBN: 0-7695-1420-0.

[GN96] Dirk Grunwald and Richard Neves. “Whole-Program Optimization for Time and Space Efficient Threads.” In: *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (Cambridge, MA, USA). New York, NY, USA: ACM Press, 1996, pp. 50–59. ISBN: 0-89791-767-7. DOI: `10.1145/237090.237149`.

[Gre16] Brendan Gregg. “The Flame Graph.” In: *Communications of the ACM* 59.6 (May 2016), pp. 48–57. ISSN: 0001-0782. DOI: `10.1145/2909476`.

[Gur19] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual 8.1*. `http://www.gurobi.com`. 2019.

[Hän+06] K. Hänninen, J. Maki-Turja, M. Bohlin, J. Carlson, and M. Nolin. “Determining Maximum Stack Usage in Preemptive Shared Stack Systems.” In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS’06)*. Dec. 2006, pp. 445–453. DOI: `10.1109/RTSS.2006.18`.

[Har77] William H. Harrison. “Compiler analysis of the value ranges for variables.” In: *IEEE Transactions on software engineering* 3 (1977), pp. 243–250.

[Har87] Paul K Harter Jr. “Response times in level-structured systems.” In: *ACM Transactions on Computer Systems (TOCS)* 5.3 (1987), pp. 232–248.

[Hec+03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. “The influence of processor architecture on the design and the results of WCET tools.” In: *Proceedings of the IEEE Real-Time Systems Symposium* 91.7 (2003), pp. 1038–1054.

[Hep+15] S. Hepp, B. Huber, D. Prokesch, and P. Puschner. “The platin Tool Kit - The T-CREST Approach for Compiler and WCET Integration.” In: *Proceedings of the 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS ’15)*. 2015.

[HKL91] M.G. Harbour, M.H. Klein, and J.P. Lehoczky. “Fixed priority scheduling periodic tasks with varying execution priority.” In: *Proceedings of the 12th IEEE International Symposium on Real-Time Systems (RTSS ’91)*. IEEE Computer Society Press, 1991, pp. 116–128. DOI: `10.1109/REAL.1991.160365`.

[HL01] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. Seventh. New York, NY, USA: McGraw-Hill, 2001.

[HLS11] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Sleepy Sloth: Threads as Interrupts as Threads.” In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS ’11)* (Vienna, Austria, Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, Dec. 2011, pp. 67–77. ISBN: 978-0-7695-4591-2. DOI: `10.1109/RTSS.2011.14`.

[HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D. Ullmann. *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Addision Wesley, 2001. ISBN: 0-201-44124-1.

[Hoa78] C.A.R. Hoare. "Communicating Sequential Processes." In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677.

[Hof+09] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. "Sloth: Threads as Interrupts." In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)* (Washington, D.C., USA, Dec. 1–4, 2009). IEEE Computer Society Press, Dec. 2009, pp. 204–213. ISBN: 978-0-7695-3875-4. DOI: `10.1109/RTSS.2009.18`.

[Hof+12] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS." In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12)* (San Juan, Puerto Rico, Dec. 4–7, 2012). IEEE Computer Society Press, Dec. 2012, pp. 237–247. ISBN: 978-0-7695-4869-2. DOI: `10.1109/RTSS.2012.75`.

[Hof14] Wanja Hofer. "Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems." PhD thesis. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014. URL: `http://opus4.kobv.de/opus4-fau/files/4875/WanjaHoferDissertation.pdf`.

[Hop71] John Hopcroft. *An $n \log n$ algorithm for minimizing states in a finite automaton*. Tech. rep. Computer Science Department, University of California, 1971.

[HPP13] B. Huber, D. Prokesch, and P. Puschner. "Combined WCET Analysis of Bitcode and Machine Code Using Control-flow Relation Graphs." In: *Proceedings of the 14th Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*. 2013, pp. 163–172.

[HR95] M. Hamdaoui and P. Ramanathan. "A dynamic priority assignment technique for streams with (m, k)-firm deadlines." In: *IEEE Transactions on Computers* 44.12 (Dec. 1995), pp. 1443–1451. ISSN: 0018-9340. DOI: `10.1109/12.477249`.

[HT05] Michael Hohmuth and Hendrik Tews. "The VFiasco approach for a verified operating system." In: *2nd PLOS* (2005).

[Hua+11] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. "Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP." In: *Proceedings of the 5th International Symposium on Theoretical Aspects of Software Engineering (TASE'11)* (Xi'an, China). Washington, DC, USA: IEEE Computer Society Press, Aug. 2011, pp. 142–149. DOI: `10.1109/TASE.2011.11`.

[Hug17] Phil Hughes. *Inside the numbers: 100 billion ARM-based chips*. `https://community.arm.com/processors/b/blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105`, accessed on 4.10.2018. Feb. 2017.

[HWH95] Christopher A Healy, David B Whalley, and Marion G Harmon. "Integrating the timing analysis of pipelining and instruction caching." In: *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*. IEEE. 1995, pp. 288–297.

[ISO11] ISO 26262-4. *ISO 26262-4:2011: Road vehicles – Functional safety – Part 4: Product development at the system level*. Geneva, Switzerland: International Organization for Standardization, 2011.

[JM09] Ranjit Jhala and Rupak Majumdar. “Software model checking.” In: *ACM Comput. Surv.* 41 (2009), 21:1–21:54.

[Joh98] Dirk John. “OSEK/VDX history and structure.” In: *IET Conference Proceedings* (Jan. 1998), 2–2(1). URL: `http://digital-library.theiet.org/content/conferences/10.1049/ic_19981073`.

[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: `/oup/backfile/content_public/journal/comjnl/29/5/10.1093/comjnl/29.5.390/2/290390.pdf`. URL: `http://dx.doi.org/10.1093/comjnl/29.5.390`.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. “The program structure tree: computing control regions in linear time.” In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’94)* (Orlando, FL, USA). New York, NY, USA: ACM Press, 1994, pp. 171–185. ISBN: 0-89791-662-X. DOI: `10.1145/178243.178258`.

[KBL10] Uğur Keskin, Reinder J Bril, and Johan J Lukkien. “Exact response-time analysis for fixed-priority preemption-threshold scheduling.” In: *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE. 2010, pp. 1–4.

[KE15] S. Kerrison and K. Eder. “Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor.” In: *ACM Transactions on Embedded Computing Systems (ACM TECS)* 14.3 (2015), p. 56.

[KE95] Steve Kleiman and Joe Eykholt. “Interrupts as Threads.” In: *ACM SIGOPS Operating Systems Review* 29.2 (Apr. 1995), pp. 21–26. ISSN: 0163-5980.

[KH05] Tae-Hyung Kim and Seongsoo Hong. “State Machine Based Operating System Architecture for Wireless Sensor Networks.” In: *Parallel and Distributed Computing: Applications and Technologies*. Ed. by Kim-Meow Liew, Hong Shen, Simon See, Wentong Cai, Pingzhi Fan, and Susumu Horiguchi. Vol. 3320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 803–806. ISBN: 978-3-540-24013-6. DOI: `10.1007/978-3-540-30501-9_158`.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming.” In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)* (Finland). Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer-Verlag, June 1997, pp. 220–242.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: formal verification of an OS kernel.” In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)* (Big Sky, Montana, USA). New York, NY, USA: ACM Press, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629596`.

[Kle09] Gerwin Klein. "Operating system verification—An overview." In: *Sadhana* 34.1 (Feb. 2009), pp. 27–69. ISSN: 0973-7677. DOI: `10.1007/s12046-009-0002-4`. URL: `http://dx.doi.org/10.1007/s12046-009-0002-4`.

[Klu+09] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. "TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS." In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*. Berkeley, California: ACM, 2009, pp. 127–140. ISBN: 978-1-60558-519-2. DOI: `10.1145/1644038.1644052`.

[KMG08] Nupur Kothari, Todd Millstein, and Ramesh Govindan. "Deriving State Machines from TinyOS Programs Using Symbolic Execution." In: *IPSN '08: Proceedings of the 7th International Conference on Information Processing in Sensor Networks*. Washington, DC, USA: IEEE Computer Society Press, 2008, pp. 271–282. ISBN: 978-0-7695-3157-1. DOI: `10.1109/IPSN.2008.62`.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Second Edition. Springer-Verlag, 2011. ISBN: 978-1-4419-8236-0.

[Kor90] Bogdan Korel. "Automated software test data generation." In: *IEEE Transactions on software engineering* 16.8 (1990), pp. 870–879.

[KW59] James E Kelley Jr and Morgan R Walker. "Critical-path planning and scheduling." In: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. ACM. 1959, pp. 160–173.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Mar. 2004.

[Lam67] Butler W. Lampson. "A Scheduling Philosophy for Multi-processing Systems." In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP '67. New York, NY, USA: ACM, 1967, pp. 8.1–8.24. DOI: `10.1145/800001.811677`. URL: `http://doi.acm.org/10.1145/800001.811677`.

[Lam82] Butler W. Lampson. "Fast Procedure Calls." In: *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*. ASPLOS I. Palo Alto, California, USA: ACM, 1982, pp. 66–76. ISBN: 0-89791-066-4. DOI: `10.1145/800050.801827`.

[Lan73] David S Landes. *Der entfesselte Prometheus: Technologischer Wandel und industrielle Entwicklung in Westeuropa von 1750 bis zur Gegenwart*. Kiepenheuer & Witsch, 1973. ISBN: 3462009281.

[Lee+01] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. "Bounding cache-related preemption delay for real-time systems." In: *IEEE Transactions on Software Engineering* 27.9 (Sept. 2001), pp. 805–826. ISSN: 0098-5589. DOI: `10.1109/32.950317`.

[Lee+14] Yunsup Lee, A. Waterman, R. Avizienis, H. Cook, Chen Sun, V. Stojanovic, and K. Asanovic. "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators." In: *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th*. Sept. 2014, pp. 199–202. DOI: `10.1109/ESSCIRC.2014.6942056`.

[Lev+05] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. "TinyOS: An Operating System for Wireless Sensor Networks." In: Ambient Intelligence. Heidelberg, Germany: Springer-Verlag, 2005.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.

[LL73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *Journal of the ACM* 20.1 (1973), pp. 46–61. ISSN: 0004-5411.

[LM95] Yau-Tsun Steven Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration." In: *ACM SIGPLAN Notices*. Vol. 30. ACM. 1995, pp. 88–98.

[LMN06] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware." In: *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2006, pp. 14–23. DOI: `10.1109/RTAS.2006.34`.

[LMW96] Y. S. Li, S. Malik, and A. Wolfe. "Cache modeling for real-time software: beyond direct mapped instruction caches." In: *17th IEEE Real-Time Systems Symposium*. Dec. 1996, pp. 254–263. DOI: `10.1109/REAL.1996.563722`.

[LP09] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers." In: *ACM Transactions on Embedded Computing Systems* 9.1 (Oct. 2009), 8:1–8:33. ISSN: 1539-9087. DOI: `10.1145/1596532.1596540`.

[LRG95] Phillip A. Laplante, Eileen P. Rose, and Maria Gracia-Watson. "An historical survey of early real-time computing developments in the U.S." In: *Real-Time Systems* 8.2 (Mar. 1995), pp. 199–213. ISSN: 1573-1383. DOI: `10.1007/BF01094343`. URL: `https://doi.org/10.1007/BF01094343`.

[LS] T. Lundqvist and P. Stenstrom. "Timing anomalies in dynamically scheduled microprocessors." In: *Proceedings 20th IEEE Real-Time Systems Symposium* (). DOI: `10.1109/real.1999.818824`. URL: `http://dx.doi.org/10.1109/real.1999.818824`.

[LT79] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141. ISSN: 0164-0925. DOI: `10.1145/357062.357071`.

[Ltd07] Live Devices Ltd. *RTA-OSEK - User Guide, v5.0.2*. 2007, pp. 1–378.

[Lv+09a] Mingsong Lv, Nan Guan, Yi Zhang, Rui Chen, Qingxu Deng, Ge Yu, and Wang Yi. "WCET Analysis of the mC/OS-II Real-Time Kernel." In: *Proceedings of the International Conference onComputational Science and Engineering (CSE '09)*. 2009, pp. 270–276.

[Lv+09b] Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. "A survey of WCET analysis of real-time operating systems." In: *Proceedings of the International Conference on Embedded Software and Systems (ICESS '09)*. 2009, pp. 65–72.

[LW82] Joseph Y-T Leung and Jennifer Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks." In: *Performance evaluation* 2.4 (1982), pp. 237–250.

[LZA09] Yu Liu, Wei Zhang, and Kemal Akkaya. "Static worst-case energy and lifetime estimation of wireless sensor networks." In: *Proceedings of the 28th International Performance Computing and Communications Conference (IPCCC '09)*. IEEE. 2009, pp. 17–24.

[LZM04] Wang Lei, Wu Zhaohui, and Zhao Mingde. "Worst-case response time analysis for OSEK/VDX compliant real-time distributed control systems." In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* Sept. 2004, 148–153 vol.1. DOI: `10.1109/CMPSAC.2004.1342819`.

[Mah+92] Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. "Effective compiler support for predicated execution using the hyperblock." In: *ACM SIGMICRO Newsletter*. Vol. 23. 1-2. IEEE Computer Society Press. 1992, pp. 45–54.

[Mar65] James Thomas Martin. *Programming real-time computer systems*. English. Englewood Cliffs, N.J. : Prentice-Hall, 1965.

[MB02] Vincent J. Mooney and Douglas M. Blough. "A Hardware-Software Real-Time Operating System Framework for SoCs." In: *IEEE Journal on Design and Test of Computers* 19.6 (2002), pp. 44–51. ISSN: 0740-7475. DOI: `10.1109/MDT.2002.1047743`.

[MB17] F. Mauroner and M. Baunach. "StackMMU: Dynamic stack sharing for embedded systems." In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2017, pp. 1–9. DOI: `10.1109/ETFA.2017.8247614`.

[MBS15] Matthias Meier, Mark Breddemann, and Olaf Spinczyk. "Interfacing the Hardware API with a Feature-based Operating System Family." In: *Journal of Systems Architecture* 61.10 (Nov. 2015), pp. 531–538. ISSN: 1383-7621. DOI: `10.1016/j.sysarc.2015.07.010`.

[McN+01] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. "Specialization Tools and Techniques for Systematic Optimization of System Software." In: *ACM Transactions on Computer Systems* 19.2 (May 2001), pp. 217–251. ISSN: 0734-2071. DOI: `10.1145/377769.377778`.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Min] N. Minorsky. "Directional Stability of Automatically Steered Bodies." In: *Journal of the American Society for Naval Engineers* 34.2 (), pp. 280–309. DOI: `10.1111/j.1559-3584.1922.tb04958.x`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1559-3584.1922.tb04958.x`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1559-3584.1922.tb04958.x`.

[Mok83] Aloysius K. L. Mok. "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment." PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, MIT, May 1983.

[Moo56] Edward F. Moore. “Gedanken-experiments on sequential machines.” In: *Automata studies*. Annals of mathematics studies, no. 34. Princeton University Press, Princeton, N. J., 1956, pp. 129–153.

[MP89] Henry Massalin and Calton Pu. “Threads and Input/Output in the Synthesis Kernel.” In: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP ’89)*. New York, NY, USA: ACM Press, 1989, pp. 191–201. ISBN: 0-89791-338-8. DOI: `10.1145/74850.74869`.

[MRR04] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Precise Call Graphs for C Programs with Function Pointers.” In: *Automated Software Engineering* 11.1 (Jan. 2004), pp. 7–26. ISSN: 1573-7535. DOI: `10.1023/B:AUSE.0000008666.56394.a1`. URL: `https://doi.org/10.1023/B:AUSE.0000008666.56394.a`.

[MSB08] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. “MTSS: Multitask Stack Sharing for Embedded Systems.” In: *ACM Trans. Embed. Comput. Syst.* 7.4 (Aug. 2008), 46:1–46:37. ISSN: 1539-9087. DOI: `10.1145/1376804.1376814`.

[Mue00] Frank Mueller. “Timing Analysis for Instruction Caches.” In: *Real-Time Systems* 18.2 (May 2000), pp. 217–247. ISSN: 1573-1383. DOI: `10.1023/A:1008145215849`. URL: `https://doi.org/10.1023/A:1008145215849`.

[Mül+14] Rainer Müller, Daniel Danner, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling.” In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS ’14)* (Madrid, Spain). Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 289–198. ISBN: 978-1-4799-5798-9. DOI: `10.1109/ECRTS.2014.30`.

[Mur+98] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. “An empirical study of static call graph extractors.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), pp. 158–191.

[Nak+95] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. “Hardware Implementation of a Real-Time Operating System.” In: *Proceedings of the 12th TRON Project International Symposium (TRON ’95)*. Nov. 1995, pp. 34–42. DOI: `10.1109/TRON.1995.494740`.

[Nel18] Jian-Jia ChenEmail authorGeoffrey NelissenWen-Hung HuangMaolin YangBjörn BrandenburgKonstantinos BletsasCong LiuPascal RichardFrédéric RidouardNeil AudsleyRaj RajkumarDionisio de NizGeorg von der Brüggen. “Many suspensions, many problems: a review of self-suspending tasks in real-time systems.” In: *Real-Time Systems* (2018), pp. 1–64. DOI: `10.1007/s11241-018-9316-9`.

[Org72] Elliot I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972. ISBN: 0-262-15012-3.

[OSE04a] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, visited 2014-09-29. OSEK/VDX Group, 2004.

[OSE04b] OSEK/VDX Group. *OSEK/VDX Communication 3.0.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf`. OSEK/VDX Group, July 2004.

[OSE04c] OSEK/VDX Group. *OSEK/VDX Network Management 2.5.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf`. OSEK/VDX Group, July 2004.

[OSE05] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[Pet63] Carl Adam Petri. “Fundamentals of a Theory of Asynchronous Information Flow.” In: *Proceedings of the IFIP Congress 62*. Amsterdam: North Holland Publ. Comp., 1963, pp. 386–390.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. “The Synthesis Kernel.” In: *Computing Systems* 1.1 (1988), pp. 11–32.

[PN15] David Patterson and Borivoje Nikolić. *Agile Design for Hardware*. EE|Times blog post. 2015. URL: `http://www.eetimes.com/author.asp?section_id=36&doc_id=1327291`.

[Pri08] P. J. Prisaznuk. “ARINC 653 role in Integrated Modular Avionics (IMA).” In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. Oct. 2008, 1.E.5-1-1.E.5–10. DOI: `10.1109/DASC.2008.4702770`.

[PS97] Peter Puschner and Anton Schedl. “Computing Maximum Task Execution Times: A Graph-Based Approach.” In: *Real-Time Systems* 13 (1997), pp. 67–91.

[Pus+13] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. “The T-CREST Approach of Compiler and WCET-Analysis Integration.” In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS ’13)*. 2013, pp. 33–40.

[Rec73] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata, 15. Stuttgart-Bad Cannstatt: Frommann-Holzboog, 1973. 170 pp. ISBN: 978-3-7728-0373-4.

[Roc14] Christine Rochange. *WCET Tool Challenge 2014*. Talk held at the 14th International Workshop on Worst-Case Execution Time Analysis (WCET ’14). 2014. URL: `http://www.uni-ulm.de/fileadmin/website_uni_ulm/wcet2014/slides/8.pdf`.

[Rou81] Robert Routledge. *Discoveries and inventions of the nineteenth century*. fifth edition. https://archive.org/details/discoveriesinven00routrich/page/6. George Routledge and Sons, London, 1881.

[Roy70] Winston W Royce. “Managing the development of large software systems.” In: *IEEE WESCON*. IEEE Computer Society Press. 1970, pp. 1–9.

[RRW05a] John Regehr, Alastair Reid, and Kirk Webb. “Eliminating Stack Overflow by Abstract Interpretation.” In: *ACM Transactions on Embedded Computing Systems* 4.4 (2005), pp. 751–778. ISSN: 1539-9087. DOI: `10.1145/1113830.1113833`.

[RRW05b] John Regehr, Alastair Reid, and Kirk Webb. “Eliminating stack overflow by abstract interpretation.” In: *ACM Transactions on Embedded Computing Systems (ACM TECS)* 4.4 (2005), pp. 751–778.

[RT74] Dennis MacAlistair Ritchie and Ken Thompson. “The Unix Time-Sharing System.” In: *Communications of the ACM* 17.7 (July 1974), pp. 365–370. DOI: `10.1145/361011.361061`.

[Rud86] Richard L Rudell. *Multiple-valued logic minimization for PLA synthesis*. Tech. rep. CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB, 1986.

[SA00] Friedhelm Stappert and Peter Altenbernd. "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs." In: *Journal of System Architecture* 46.4 (2000), pp. 339–355.

[Sak98] Ken Sakamura. *μItron 3.0: An Open and Portable Real-Time Operating System for Embedded Systems : Concept and Specification*. IEEE Computer Society Press, 1998. ISBN: 978-0818677953.

[San+04] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. "Static timing analysis of real-time operating system code." In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2004, pp. 146–160.

[Sch+15] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparso, and Alessandro Tocchi. "T-CREST: Time-predictable multi-core architecture for embedded systems." In: *Journal of Systems Architecture* 61.9 (2015), pp. 449–471. DOI: `10.1016/j.sysarc.2015.04.002`.

[Sch11] Fabian Scheler. "Atomic Basic Blocks: Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur." PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät, 2011. URL: `http://opus4.kobv.de/opus4-fau/files/1740/FabianSchelerDissertation.pdf`.

[SEE01] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. "Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects." In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '01. Atlanta, Georgia, USA: ACM, 2001, pp. 132–140. ISBN: 1-58113-399-5. DOI: `10.1145/502217.502240`. URL: `http://doi.acm.org/10.1145/502217.502240`.

[SH97] Marc Shapiro and Susan Horwitz. "Fast and accurate flow-insensitive points-to analysis." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 1–14.

[Sha+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. "Real-Time Scheduling Theory: A Historical Perspective." In: *Real-Time Systems Journal* 28.2–3 (2004), pp. 101–155.

[Sha80] M. Sharir. "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers." In: *Comput. Lang.* 5.3-4 (Jan. 1980), pp. 141–153. ISSN: 0096-0551. DOI: `10.1016/0096-0551(80)90007-7`.

[Sha89] A. C. Shaw. "Reasoning About Time in Higher-Level Language Software." In: *IEEE Trans. Softw. Eng.* 15.7 (July 1989), pp. 875–889. ISSN: 0098-5589. DOI: `10.1109/32.29487`. URL: `https://doi.org/10.1109/32.29487`.

[Sho10] Michael Short. “Improved Task Management Techniques for Enforcing EDF Scheduling on Recurring Tasks.” In: *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2010, pp. 56–65. DOI: `10.1109/RTAS.2010.22`.

[SL07] Olaf Spinczyk and Daniel Lohmann. “The Design and Implementation of AspectC++.” In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20.7 (2007), pp. 636–651. DOI: `10.1016/j.knosys.2007.05.004`.

[SMK13] Thomas Sewell, Magnus Myreen, and Gerwin Klein. “Translation validation for a verified OS kernel.” In: *Proc. PLDI’13*. ACM, 2013, pp. 471–482.

[SP81] Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data flow analysis.” In: *Program Flow Analysis: Theory and Applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Englewood Cliffs, NJ: Prentice-Hall, 1981. Chap. 7, pp. 189–234.

[SRL90a] L. Sha, R. Rajkumar, and J. P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Transactions on Computers* 39.9 (Sept. 1990), pp. 1175–1185. ISSN: 0018-9340. DOI: `10.1109/12.57058`.

[SRL90b] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: `10.1109/12.57058`.

[SS11] Fabian Scheler and Wolfgang Schröder-Preikschat. “The Real-Time Systems Compiler: migrating event-triggered systems to time-triggered systems.” In: *Software: Practice and Experience* 41.12 (2011), pp. 1491–1515. ISSN: 1097-024X. DOI: `10.1002/spe.1099`.

[SSL89] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. “Aperiodic Task Scheduling for Hard Real-Time Systems.” In: *Real-Time Systems Journal* 1.1 (1989), pp. 27–60.

[Ste77] Guy Lewis Steele Jr. “Macaroni is Better Than Spaghetti.” In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 60–66. DOI: `10.1145/800228.806933`. URL: `http://doi.acm.org/10.1145/800228.806933`.

[Ste96] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time.” In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41. ISBN: 0-89791-769-3. DOI: `10.1145/237721.237727`.

[Sti+11] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. “The digraph real-time task model.” In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE. 2011, pp. 71–80.

[SW12] James Stanier and Des Watson. “A Study of Irreducibility in C Programs.” In: *Softw. Pract. Exper.* 42.1 (Jan. 2012), pp. 117–130. ISSN: 0038-0644. DOI: `10.1002/spe.1059`. URL: `http://dx.doi.org/10.1002/spe.1059`.

[Sys17] Espressif Systems. *ESP8266 Technical Reference*. 2017.

[Tan06] Andrew S. Tanenbaum. *Structured Computer Organization*. Fifth. Prentice Hall PTR, 2006. ISBN: 978-0131485211.

[TBW94] Ken W Tindell, Alan Burns, and Andy J. Wellings. “An extendible approach for analyzing fixed priority hard real-time tasks.” In: *Real-Time Systems* 6.2 (1994), pp. 133–151.

[Ten00] David Tennenhouse. “Proactive Computing.” In: *Communications of the ACM* (May 2000), pp. 43–45.

[The00] Henrik Theiling. “Extracting safe and precise control flow from binaries.” In: *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*. IEEE. 2000, pp. 23–30. DOI: `10.1109/RTCSA.2000.896367`.

[The02a] Henrik Theiling. “Control flow graphs for real-time systems analysis.” PhD thesis. 2002.

[The02b] Henrik Theiling. “ILP-Based Interprocedural Path Analysis.” In: *Embedded Software*. Ed. by Alberto Sangiovanni-Vincentelli and Joseph Sifakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 349–363. ISBN: 3-540-44307-X.

[The97] The Santa Cruz Operation (SCO). *System V Application Binary Interface - Intel386 Architecture Processor Supplement*. bibtex:sco:97:i386-abi. 1997. URL: `http://www.sco.com/developers/devspecs/abi386-4.pdf`.

[Tig+17] K. Tigori, J.-L. Béchennec, S. Faucou, and O. Roux. “Formal Model-Based Synthesis of Application-Specific Static RTOS.” In: *ACM Transactions on Embedded Computing Systems* 16 (2017), 97:1–97:25.

[Tur36] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem.” In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265. URL: `http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf`.

[Ulb+11] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. “I4Copter: An Adaptable and Modular Quadrotor Platform.” In: *Proceedings of the 26th ACM Symposium on Applied Computing (SAC ’11)* (TaiChung, Taiwan). New York, NY, USA: ACM Press, 2011, pp. 380–396. ISBN: 978-1-4503-0113-8.

[UU04] Ryuhei Uehara and Yushi Uno. “Efficient algorithms for the longest path problem.” In: *International Symposium on Algorithms and Computation*. Springer. 2004, pp. 871–883.

[VA12] Dieu-Huong Vu and Toshiaki Aoki. “Faithfully formalizing OSEK/VDX operating system specification.” In: *Proceedings of Symposium on Symposium on Information and Communication Technology 2012*. ACM, 2012, pp. 13–20.

[Val98] Antti Valmari. “The state explosion problem.” In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528. ISBN: 978-3-540-49442-3. DOI: `10.1007/3-540-65306-6_21`. URL: `https://doi.org/10.1007/3-540-65306-6_21`.

[VHL14] Marcus Völp, Marcus Hähnel, and Adam Lackorzynski. “Has Energy Surpassed Timeliness? – Scheduling Energy-Constrained Mixed-Criticality Systems.” In: *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS ’14)*. 2014, pp. 275–284.

[Von45] John Von Neumann. *First Draft of a Report on the EDVAC*. Research Report. University of Pennsylvania, 1945. URL: `https://archive.org/details/vnedvac`.

[VS89] T. Villa and A. Sangiovanni-Vincentelli. “NOVA: State Assignment of Finite State Machines for Optimal Two-level Logic Implementations.” In: *Proceedings of the 26th ACM/IEEE Design Automation Conference* (Las Vegas, Nevada, USA). DAC ’89. New York, NY, USA: ACM Press, 1989, pp. 327–332. ISBN: 0-89791-310-8. DOI: `10.1145/74382.74437`.

[VT88] D. Varma and E.A. Trachtenberg. “A fast algorithm for the optimal state assignment of large finite state machines.” In: *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*. Nov. 1988, pp. 152–155. DOI: `10.1109/ICCAD.1988.122483`.

[Vu+16] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, and Toshiaki Aoki. “Verifying OSEK/VDX OS Design Using Its Formal Specification.” In: *Proc. TASE’16*. IEEE Computer Society, 2016, pp. 81–88.

[Wäg+15] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. “Worst-case energy consumption analysis for energy-constrained embedded systems.” In: *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS ’15)* (Lund, Sweden). Washington, DC, USA: IEEE Computer Society Press, 2015, pp. 105–114. ISBN: 978-1-4673-7570-2. DOI: `10.1109/ECRTS.2015.17`.

[Wäg+17] Peter Wägemann, Tobias Distler, Christian Eichler, and Wolfgang Schröder-Preikschat. “Benchmark Generation for Timing Analysis.” In: *Proceedings of the 23nd Real-Time and Embedded Technology and Applications Symposium (RTAS ’17)*. 2017.

[Wan+16] Chao Wang, Chuansheng Dong, Haibo Zeng, and Zonghua Gu. “Minimizing Stack Memory for Hard Real-Time Applications on Multicore Platforms with Partitioned Fixed-Priority or EDF Scheduling.” In: *ACM Transactions on Design Automation of Electronic Systems* 21.3 (May 2016), 46:1–46:25. ISSN: 1084-4309. DOI: `10.1145/2846096`.

[Wat+14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014.

[WB13] Alexander Wieder and Björn B. Brandenburg. “On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks.” In: *Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS ’13)* (Vancouver, Canada, Dec. 3, 2013–Dec. 6, 2013). IEEE Computer Society Press, Dec. 2013, pp. 45–56. ISBN: 978-1-4799-2007-5. DOI: `10.1109/RTSS.2013.13`.

[WGZ16] Chao Wang, Zonghua Gu, and Haibo Zeng. “Global Fixed Priority Scheduling with Preemption Threshold: Schedulability Analysis and Stack Size Minimization.” In: *IEEE Transactions on Parallel and Distributed Systems* 27.11 (Nov. 2016), pp. 3242–3255. ISSN: 1045-9219. DOI: `10.1109/TPDS.2016.2528978`.

[WH08] Libor Waszniowski and Zdeněk Hanzálek. “Formal Verification of Multitasking Applications Based on Timed Automata Model.” In: *Real-Time Systems* 38.1 (Jan. 2008), pp. 39–65. ISSN: 0922-6443. DOI: `10.1007/s11241-007-9036-z`.

[Whi94] Darrell Whitley. “A Genetic Algorithm Tutorial.” In: *Statistics and Computing* 4.2 (June 1, 1994), pp. 65–85. ISSN: 1573-1375. DOI: `10.1007/BF00175354`. URL: `https://doi.org/10.1007/BF00175354` (visited on 03/07/2019).

[Wil+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 (ACM TECS)* 7.3 (2008), pp. 1–53.

[WKP13] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. "Worst case analysis of DRAM latency in multi-requestor systems." In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE. 2013, pp. 372–383.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. "Specification and verification of the UCLA Unix security kernel." In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 118–131. ISSN: 0001-0782. DOI: `10.1145/358818.358825`. URL: `http://dx.doi.org/10.1145/358818.358825`.

[WS99] Yun Wang and Manas Saksena. "Scheduling Fixed-Priority Tasks with Preemption Threshold." In: *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*. RTCSA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 328–. ISBN: 0-7695-0306-3.

[Xia+18] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson. "CheriRTOS: A Capability Model for Embedded Devices." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018 IEEE 36th International Conference on Computer Design (ICCD). Orlando, FL, USA: IEEE, Oct. 2018, pp. 92–99. ISBN: 978-1-5386-8477-1. DOI: `10.1109/ICCD.2018.00023`. URL: `https://ieeexplore.ieee.org/document/8615673/` (visited on 03/05/2019).

[XP90] J. Xu and David Parnas. "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations." In: *IEEE Transactions on Software Engineering* 16.3 (1990), pp. 360–369. ISSN: 0098-5589. DOI: `10.1109/32.48943`.

[YB10] Gang Yao and Giorgio Buttazzo. "Reducing Stack with Intra-task Threshold Priorities in Real-time Systems." In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '10. Scottsdale, Arizona, USA: ACM, 2010, pp. 109–118. ISBN: 978-1-60558-904-6. DOI: `10.1145/1879021.1879036`.

[Yi+07] Sangho Yi, Seungwoo Lee, Yookun Cho, and Jiman Hong. "OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems." In: *Computational Science – (ICCS'07)*. Ed. by Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 905–912. ISBN: 978-3-540-72590-9.

[ZAC15] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba. "Verifying OSEK/VDX Applications: A Sequentialization-Based Model Checking Approach." In: *IEICE Transactions* 98-D.10 (2015), pp. 1765–1776.

[ZBN93] Ning Zhang, Alan Burns, and Mark Nicholson. "Pipelined processors and worst case execution times." In: *Real-Time Systems* 5.4 (1993), pp. 319–343.

[ZCO14] Min Zhang, Yunja Choi, and Kazuhiro Ogata. “A Formal Semantics of the OSEK/VDX Standard in K Framework and Its Applications.” In: *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*. Springer, 2014, pp. 280–296.

[ZDZ14] Haibo Zeng, Marco Di Natale, and Qi Zhu. “Minimizing Stack and Communication Memory Usage in Real-Time Embedded Applications.” In: *ACM Trans. Embed. Comput. Syst.* 13.5s (July 2014), 149:1–149:25. ISSN: 1539-9087. DOI: `10.1145/2632160`.

[Zha+13] Haitao Zhang, Toshiaki Aoki, Hsin-Hung Lin, Min Zhang, Yuki Chiba, and Kenro Yatake. “SMT-Based Bounded Model Checking for OSEK/VDX Applications.” In: *Proc. APSEC’13*. IEEE Computer Society, 2013, pp. 307–314.

[Zim+14] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. “FlexPRET: A processor platform for mixed-criticality systems.” In: *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS ’14)*. Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 101–110. DOI: `10.1109/RTAS.2014.6925994`.

List of Figures

1.1 Centrifugal Governor . . . 3
1.2 Interaction Graph for General and Specialized RTOS . . . 5
1.3 Example for Response Time Analysis . . . 6
1.4 Structural Overview of this Thesis . . . 7

2.1 Real-Time System . . . 11
2.2 RTOS with and without Native Task Support . . . 16
2.3 Mapping of Task Dependencies with RTOS Services . . . 18
2.4 Diffrerent Mapping of Real-Time Concepts onto RTOS Abstractions . . . 23
2.5 Non-Canonical Usages of the RTOS API . . . 25

3.1 Interaction between Threads and RTOS . . . 31
3.2 Fine-Grained Interaction Analysis . . . 32
3.3 Basic Blocks and Control-Flow Graph . . . 35
3.4 Instruction Stream . . . 36
3.5 Atomic Basic Blocks . . . 37
3.6 Node Expansion for SESE Regions and ABB Merging . . . 38
3.7 Function-local ABB Control-Flow Graphs . . . 40
3.8 Abstract System State . . . 40
3.9 Static State Transition Graph . . . 44
3.10 State Explosion for Computation ABBs . . . 45
3.11 Compression for Application State Machines . . . 46
3.12 Example Global Control-Flow Graph . . . 48
3.13 GCFG without ISR Regions . . . 48
3.14 Data-Flow of Imprecise AbSSs in the SSF Analysis . . . 50
3.15 dOSEK Generator Framework . . . 56

4.1 Control-Flow Sensitive Response Times . . . 63
4.2 IPET Example . . . 67
4.3 SysWCET IPET Layers . . . 69
4.4 Example System with Response-Time Annotations . . . 69
4.5 Extended Example System and its SSTG . . . 71
4.6 Problems with naïve ABB-Count Derivation . . . 72
4.7 IRQ Handling in Operation Systems . . . 78
4.8 The *i4*Copter . . . 83
4.9 Problems with Self-Suspending Tasks and Idle Times . . . 86
4.10 Worst-Case Energy Consumption with Switchable Device . . . 88
4.11 Energy Consumption on Different System Paths . . . 89
4.12 Power Atomic Basic Blocks . . . 90

5.1 dOSEK System Generation and Behavioral Verification . . . 99
5.2 Slice of the OSEK CTL Specification . . . 101

6.1 Private Thread Stacks . . . 114
6.2 Basic Task Stack . . . 115

6.3 Basic Principle of Semi-Extended Tasks 119
6.4 Regular Intel IA-32 Call Frame 120
6.5 Intra-Thread Stack Switch Mechanism 121
6.6 Stack Consumption and Preemption Constraints 122
6.7 Switch-Function Selection Problem 126
6.8 Genetic Operators for Switch-Function Selection 127
6.9 Run-Time of Stack-Optimization Process 131
6.10 Stack-Saving Factors 133
6.11 Composition of the Switch-Function Set 135

7.1 System State Machine 146
7.2 Minimized State Machines 147
7.3 PLA Implementation of the SSM 148
7.4 The OSEK-V Pipeline 149
7.5 Static-Alarm Component 150
7.6 OSEK-V Latencies and Cache-Stall Cycles 154

List of Tables

2.1 Important Real-Time System Parameters 12
2.2 Example for a Time-Triggered Scheduling Table 14
2.3 OSEK System Calls 21

4.1 Results for the Micro Benchmarks 80
4.2 WCRT Results for the *i4*Copter 84

5.1 Verification Effort for Positive Tests 104

6.1 Market Prices of AVR ATXmega C3 Series 113
6.2 Dimensions of the Generated Benchmarks 129

7.1 SSM Generation for the Benchmarks 152
7.2 Resource Consumption of Different OSEK-V Cores 153

List of Listings

2.1 Example for an OIL file . . . 20
3.1 WaitEvent in the Abstract Universe . . . 42
3.2 The System-State Flow Analysis . . . 51
3.3 Scheduler for SSF analysis . . . 53
4.1 Triple Modular Redundant Application . . . 79
4.2 Computation Interrupted by Periodic Alarm . . . 81
4.3 Abortable Computation . . . 81
5.1 NuSMV Model - Global Variables . . . 100
5.2 NuSMV Model - Transitions . . . 100
5.3 Mockup Application for Dynamic State Exploration . . . 103

Christian Dietrich

Persönliche Daten

Name	Christian Dietrich
Anschrift	Mecklenheidestrasse 7, 30419 Hannover
E-Mail	dietrich@sra.uni-hannover.de
Geburtsdaten	05. Dezember 1989, Rothenburg o.d.T

Wissenschaftlicher Werdegang

1996 – 2000	**Grundschule**, *Grund- und Teilhauptschule Geslau.*
2000 – 2009	**Abitur**, *Reichsstadtgymnasium Rothenburg o.d.T..*
Oktober 2009 - September 2012	**Studium: Informatik, B.Sc.**, *Friedrich-Alexander-Universität Erlangen-Nürnberg.*
Oktober 2012 - Oktober 2014	**Studium: Informatik, M.Sc.**, *Friedrich-Alexander-Universität Erlangen-Nürnberg.*
Januar 2015 - Dezember 2016	**Wissenschaftlicher Mitarbeiter**, *Lehrstuhl für Verteilte Systeme und Betriebssysteme*, Uni Erlangen-Nürnberg.
Januar 2017 -	**Wissenschaftlicher Mitarbeiter**, *System- und Rechnerarchitektur*, Leibniz Universität Hannover.

Auszeichnungen

2012	**Beste Bachelorarbeit**, *Department Informatik*, Uni Erlangen-Nürnberg.
2015	**RTAS, Best Paper**, *“dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel”*, Hoffmann, Lukas, Dietrich und Lohmann.
	Beste Abschlussarbeit, *Fachgruppe Betriebssysteme.*
2017	**RTAS, Outstanding Paper**, *“SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems”*, Dietrich, Wägemann, Ulbrich und Lohmann.
	USENIX ATC, Best Paper, *“cHash: Detection of Redundant Compilations via AST Hashing”*, Dietrich, Rothberg, Füracker, Ziegler und Lohmann.
2018	**ECRTS, Outstanding Paper**, *“Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems”*, Wägemann, Dietrich, Distler, Ulbrich und Schröder-Preikschat.
	OSPERT Workshop, Best Paper, *“Levels of Specialization in Real-Time Operating Systems”*, Fiedler, Entrup, Dietrich und Lohmann.