Schedulability-Oriented Code Optimization of Hard Real-Time Multitasking Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von
Dipl.-Ing. Arno Luppold

aus
Esslingen am Neckar

2020

Gutachter:	Prof. Dr. Heiko Falk Prof. Dr. Sibylle Schupp Prof. Dr. Rolf Ernst
Tag der mündlichen Prüfung:	19. Juni 2020

Zusammenfassung

Mit dem Begriff *Eingebettete Systeme* werden Rechensysteme bezeichnet, deren Funktion in einen technischen Kontext eingebettet ist. Durch die physikalischen Eigenschaften der Umgebung ergeben sich häufig Zeitschranken, die das System bei seinen Berechnungen einhalten muss. Solche Systeme werden als *Real-* oder *Echtzeitsystem* bezeichnet. Ist die Einhaltung der Zeitschranken für die Funktionsweise des Systems unabdingbar, spricht man von einem *harten* Echtzeitsystem. Moderne harte Echtzeitsysteme sind oftmals Mehrprozess-Systeme, deren Ausführung von einem sogenannten *Scheduler* nach gegebenen Vorgaben geplant wird. Bestehende Verfahren ermöglichen es, die schlimmstmögliche Ausführungszeit jedes Prozesses innerhalb des Systems vorherzusagen. Hiermit kann dann abgeleitet werden, ob das Gesamtsystem mit allen Prozessen im Wechselspiel alle vorgegebenen Zeitschranken einhält.

Die vorliegende Arbeit beschäftigt sich mit der Frage, wie ein System auf Code-Ebene optimiert werden kann, wenn das System seine Zeitschranken initial *nicht* einhält. Dabei unterscheidet es sich von bestehenden Ansätzen insofern, dass sowohl wechselseitige Einflüsse zwischen Prozessen berücksichtigt werden, als auch detaillierte Informationen zum schlimmstmöglichen Laufzeitverhalten des entsprechenden Maschinencodes. Bestehende Compiler-Optimierungen beschränkten sich hingegen bislang entweder explizit auf die *durchschnittliche* Ausführungszeit oder konzentrierten sich auf einzelne Prozesse, ohne deren Wechselwirkungen zu berücksichtigen.

Im Rahmen der vorliegenden Arbeit wurde ein auf ganzzahlig-linearer Programmierung (ILP) basierendes Modell entworfen, um konkrete Vorhersagen zum schlimmstmöglichen Laufzeitverhalten einzelner Prozesse und des Gesamtsystems im Wechselspiel treffen zu können. Dies ermöglicht es, ein Mehrprozesssystem innerhalb eines Compiler-Frameworks gezielt und automatisiert hinsichtlich der Einhaltung *aller* Zeitschranken für *alle* Prozesse zu optimieren. Als Ergänzung und zur Vergleichsmöglichkeit wurde des weiteren ein Optimierungsmodell basierend auf einem genetischen Algorithmus entworfen. Die Arbeit zeigt, dass mithilfe des auf ILP basierenden Modells signifikante Verbesserungen bei der Optimierung harter Echtzeitsysteme mit mehreren Prozessen erzielt werden können. Des Weiteren zeigte sich, dass der ILP-Ansatz dem genetischen Algorithmus sowohl hinsichtlich der Laufzeit als auch der Güte der Optimierung durchgängig überlegen ist.

Kurzzusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der compilergestützten Optimierung eingebetteter harter Echtzeitsysteme mit mehreren Prozessen. Zwei Modelle basierend auf ganzzahlig-linearer Programmierung und einem genetischen Algorithmus werden genutzt, um Systeme gezielt hinsichtlich iherer Zeitschranken zu optimieren. Die Arbeit vereinigt dabei Erkenntnisse aus der systemnahen Planbarkeitsanalyse mit Konzepten der compilerbasierten Code-Optimierung.

Abstract

This thesis tackles the compiler-based optimization of embedded hard real-time systems featuring multiple processes. It presents two models, based on integer-linear programming and a genetic algorithm. They can be used in order to optimize a given multitasking system specifically with respect to its timing requirements. The thesis thereby unites system-level schedulability analysis with concepts from compiler-based code optimization.

Acknowledgments

I would like to thank my advisor Prof. Dr. Heiko Falk for his continuous efforts and support during the creation of this thesis. Despite your tight schedule you always found the time for scientific discussions and giving constructive feedback to any of my ideas. Without your expertise, this work could not have been done.

Deep thanks go to the Deutsche Forschungsgemeinschaft (DFG) for supporting this research under grants no. 200265263 and no. 380772147. I also appreciate the support and effort by my alma maters Ulm University and Hamburg University of Technology. Both universities laid the foundations and work environment for the creation of this thesis.

Special thanks go to the team of Hochschulsport Hamburg, as well as Alexander Pyrkotsch, for helping me free my mind after intensive work in a great atmosphere and with even greater people.

Neither my studies of electrical engineering at Ulm University, nor the subsequent strive for conferral of a doctorate would have been possible without the loving support of my parents Ernst and Rosemarie, as well as my sister Stefanie. Thank you for giving me your love and support over all these many years.

I would like to distinctively thank my friend and former colleague Dominic Oehlert for all the discussions, feedback and the great work environment during my years at the Hamburg University of Technology. Thank you for always finding the time to discuss my thoughts. Also thank you for the profound proofreading of this thesis' draft versions. Without your support I would have missed many errors, while I certainly still managed to slip some back in after your feedback.

Additionally, I would like to thank everyone else who helped proofreading this thesis. Thank you for spending your time to help me creating the best possible version of this thesis.

Last, but not least, I want to express my deep gratitude to my beloved wife Christina. Your love and support was crucial for helping me through these often stressful times. Thank you for all your understanding and indulgence even when my focus was mostly on finishing this thesis. Without you, this work would not have been possible.

Publications

Parts of this thesis have been published in peer-reviewed scientific journals, proceedings of conferences and workshops, or as technical report. They are listed below in chronological order. The concrete contributions of each paper to the respective part of this theses are discussed in the respective chapters.

- Arno Luppold, Benjamin Menhorn, Heiko Falk, and Frank Slomka. "A New Concept for System-Level Design of Runtime Reconfigurable Real-Time Systems". In: *ACM SIGBED Review* 10.4 (Dec. 2013), pp. 57–60. DOI: 10.1145/2583687.2583701
- Arno Luppold and Heiko Falk. "Schedulability-Oriented WCET-Optimization of Hard Real-Time Multitasking Systems". In: *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. Versailles / France, Oct. 2014, pp. 9–12
- Nicolas Roeser, Arno Luppold, and Heiko Falk. "Multi-Criteria Optimization of Hard Real-Time Systems". In: *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. Versailles / France, Oct. 2014, pp. 49–52
- Arno Luppold and Heiko Falk. "Code Optimization of Periodic Preemptive Hard Real-Time Multitasking Systems". In: *Proceedings of the 18th International Symposium on Real-Time Distributed Computing (ISORC)*. Auckland / New Zealand, Apr. 2015, pp. 35–42. DOI: 10.1109/isorc.2015.8
- Arno Luppold and Heiko Falk. "Schedulability aware WCET-Optimization of Periodic Preemptive Hard Real-Time Multitasking Systems". In: *Proceedings of the 18th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2015, pp. 101–104. DOI: 10.1145/2764967.2771930
- Arno Luppold, Christina Kittsteiner, and Heiko Falk. "Cache-Aware Instruction SPM Allocation for Hard Real-Time Systems". In: *Proceedings of the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2016, pp. 77–85. DOI: 10.1145/2906363.2906369
- Dominic Oehlert, Arno Luppold, and Heiko Falk. "Practical Challenges of ILP-based SPM Allocation Optimizations". In: *Proceedings of the 19th*

International Workshop on Software & Compilers for Embedded Systems (SCOPES). St. Goar / Germany, May 2016, pp. 86–89. DOI: 10.1145/2906363.2906371

- Arno Luppold and Heiko Falk. "Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns". In: *Proceedings of Design, Automation and Test in Europe (DATE)*. Lausanne / Switzerland, Mar. 2017, pp. 1074–1079. DOI: 10.23919/date.2017.7927149
- Dominic Oehlert, Arno Luppold, and Heiko Falk. "Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems". In: *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*. Dubrovnik / Croatia, June 2017, 1:1–1:22. DOI: 10.4230/LIPIcs.ECRTS.2017.1
- Kateryna Muts, Arno Luppold, and Heiko Falk. "Multi-Criteria Compiler-Based Optimization of Hard Real-Time Systems". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 54–57. DOI: 10.1145/3207719.3207730
- Dominic Oehlert, Arno Luppold, and Heiko Falk. "Mitigating Data Cache Aging through Compiler-Driven Memory Allocation". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 58–61. DOI: 10.1145/3207719.3207731
- Mikko Roth, Arno Luppold, and Heiko Falk. "Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 86–89. DOI: 10.1145/3207719.3207729
- Dominic Oehlert, Arno Luppold, and Heiko Falk. "Compilation for Real-Time Systems - An Overview of the WCET-Aware C Compiler WCC". in: *Proceedings of the 9th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Barcelona / Spain, July 2018, pp. 1–3. DOI: 10.15480/882.2271
- Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. "Automated generation of time-predictable multi-core hardware executables on multi-core". In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS)*. Poitiers / France, Oct. 2018, pp. 104–113. DOI: 10.1145/3273905.3273907
- Arno Luppold, Dominic Oehlert, and Heiko Falk. *Evaluating the Performance of Solvers for Integer-Linear Programming*. Technical Report. Hamburg / Germany: Institute of Embedded Systems, Hamburg University of Technology, Nov. 2018. DOI: 10.15480/882.1839

- Dominic Oehlert, Arno Luppold, and Heiko Falk. "Favorable Adjustment of Periods for Reduced Hyperperiods in Real-Time Systems". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2019, pp. 82–85. DOI: 10.1145/3323439.3323975
- Kateryna Muts, Arno Luppold, and Heiko Falk. "Compiler-Based Code Compression for Hard Real-Time Systems". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2019, pp. 72–81. DOI: 10.1145/3323439.3323976
- Arno Luppold, Dominic Oehlert, and Heiko Falk. "Compiling for the Worst Case: Memory Allocation for Multi-Task and Multi-Core Hard Real-Time Systems". In: *ACM Transactions for Embedded Computing Systems* 19.2 (Mar. 2020), 14:1–14:26. DOI: 10.1145/3381752

Contents

Acronyms and Notational Conventions

The following notational conventions are made throughout all subsequent sections and chapters:

- Constants are denoted by capital letters, e.g., D.
- Values which are variable and may be changed by an optimization are described by lower letters, e.g., c.
- Sets are denoted by caligraphic letters, e.g., $\mathcal{C}$.
- Descriptors like a task are described using Greek letters, e.g., τ or Γ.

The following sections give an overview of acronyms and commonly used symbols. Due to their number, symbols which are solely used for the Integer-Linear Programming (ILP) model are listed in a separate list.

Acronyms

ACET Average-Case Execution Time.
ALU Arithmetic Logical Unit.
ASIC Application-Specific Integrated Circuit.

BCET Best-Case Execution Time.

CFG Control Flow-Graph.
CISC Complex Instruction Set Computer.
CRPD Cache-Related Preemption Delay.

DMS Deadline-Monotonic Scheduling.
DSP Digital Signal Processor.

ECB Evicting Cache Blocks.
EDF Earliest Deadline First.

FIFO First In First Out.

HIR High-Level Intermediate Representation.

ILP Integer-Linear Programming.
IPET Implicit Path Enumeration Technique.
IR Intermediate Representation.

ISR Interrupt Service Routine.

LFU Least-Frequently Used.
LLIR Low-Level Intermediate Representation.
LRU Least-Recently Used.

MILP Mixed Integer-Linear Program.

RISC Reduced Instruction Set Computer.
RMS Rate-Monotonic Scheduling.
RTC Real-Time Calculus.

SAT Boolean Satisfiability Problem.
SPM Scratchpad Memory.

TDMA Time Division Multiple Access.

UCB Useful Cache Block.

VLIW Very Long Instruction Word.

`WCC` WCET-Aware C Compiler.
WCEC Worst-Case Execution Count.
WCEP Worst-Case Execution Path.
WCET Worst-Case Execution Time.
WCRT Worst-Case Response Time.

Commonly Used Symbols

b Blocking Time of a task. An additional index describes which task it is associated with.

c Worst-Case Execution Time of a task. An additional index describes which task it is associated with.

c^- Best-Case Execution Time of a task. An additional index describes which task it is associated with.

D Deadline of a task. An additional index describes which task it is associated with.

δ Interval function for the activations of a task.

e Eviction penalty of a task. Optional indices denote the evicting and the evicted task. $e_{i,j}$ holds the penalty for task τ_i preempted by task τ_j.

Γ The set of tasks in a system.

J Jitter of the period T of a periodical task. An additional index describes which task it is associated with.

η Density function for the activations of a task.

P Priority of a task τ_i. Numerically lower values denote higher priorities. An additional index describes which task it is associated with. In fixed-priority systems, it is assumed that $P_i \equiv i$..

r Worst-Case Response Time of a task. An additional index describes which task it is associated with.

T Period of a strictly periodical task. An additional index describes which task it is associated with.

τ A task in a task set Γ.

u Load of a system. A load of 1 corresponds to a workload of 100 %.

Commonly Used ILP Symbols

a Binary variable denoting an active cache line. $a_{i,k} \equiv 1$ denotes that the k'th cache line which is possibly occupied by basic block i is actually used by block i.

α Number of bits for the cache offset.

$\mathcal{B}$ Set of all basic blocks for a given system.

β Number of bits for the cache offset and index.

C Worst-Case Execution Time which is considered as a constant in the optimization model. An additional index describes which task or basic block it is associated with.

c ILP variable for the net execution time of one single execution of a basic block. c_i denotes the net execution time of basic block i.

q Binary ILP variable for the occupied cache line. $q_{i,j,k,k'}$ denotes whether the k'-'th occupied cache line of basic block j conflicts with the k'th occupied cache line of basic block i.

G Timing gain constant. G_i denotes the gain if basic block i is moved to the Scratchpad Memory (SPM).

γ Number of bits for the cache offset, index and tag.

h Binary ILP variable denoting a cache miss. $h_{i,k}$ denotes that the k'th cache line occupied by basic block i may suffer from a cache miss.

m Integer variable modeling a memory address. $m_{i,0}$ holds the address of the begin of basic block i. $m_{i,E}$ models the address of the end of basic block i.

$P_{\mathbf{con}}$ Worst-Case penalty inflicted to a task on preemption due to context saving and restoring.

$P_{\mathbf{int}}$ Worst-Case penalty inflicted to a task on preemption due to interrupt routine latencies.

$P_{\mathbf{miss}}$ Worst-Case penalty inflicted to a task on preemption due to one single cache miss.

S A size constant. S_i denotes the net size of basic block i. S_{SPM} denotes the total size of the SPM.

s An ILP integer variable modeling the overall size of a basic block. s_i describes the size of basic block i.

v ILP variable for the resource demand. $v_{i,K,\Delta t}$ denotes the resource demand for the K'th instance of task τ_i in the time interval Δt.

w ILP variable for the accumulated execution time. w_i denotes the accumulated execution time of basic block i and all its successors.

x Binary ILP decision variable. x_i is set to 1 if a basic block i should be allocated to the SPM memory, and 0 else.

z Binary ILP decision variable. $z_{i,j}$ is set to 1 if additional jump correction code must be inserted in order to reach block j from block i.

1. Introduction

1.1. Motivation

Over the last decades, computers made their way from large warehouse sized machines only used for scientific or military calculations into our everyday lives. Apart from general-purpose processing appliances like personal computers, laptops, smartphones or tablets, a huge variety of our everyday life devices and machines are controlled by small computing devices. Examples range from refrigerators over cars to airplanes. These so-called embedded systems are expected to work mostly invisible to the user and reliably fulfill their purpose.

Many of these systems must react in a given amount of time in order to correctly fulfill their given task. In this case, the functional correctness of the system is not only given by being sure that the system's output is computationally correct but also by guaranteeing that the result is made available within a predetermined amount of time. To keep up with the growing user-demand for features and comfort functions, these so-called real-time systems transformed from small, assembly-programmed, microprocessor devices into full-fledged computing systems running multiple tasks on powerful and complex hardware.

However, adhering to the given design constraints is as critical before. Depending on the concrete application, returning just one single result late may lead to catastrophic events including, but not limited to, endangering humans' lives. In order to prove whether such so-called hard real-time systems will provably meet all timing requirements – even under worst-case circumstances – so-called schedulability analysis techniques can be applied. Over the last years, these techniques grew from very simple and pessimistic models to sophisticated analyses which allow to tightly estimate the adherence to timing requirements even for the most complex systems.

As a precondition, however, it is mandatory that all timing properties like the worst possible execution time of each task are known. The so-called Worst-Case Execution Time (WCET) analysis therefore focuses on determining exactly these numbers and passing them on to the schedulability analysis.

Surprisingly, a lot less effort has been spent in order to tackle the issue what should be done if a system is proven to *not* hold its timing constraints. As a result,

system developers usually have to modify their design in an iterative trial-and-error process until the subsequent analysis works out. Alternatively, the system designer may resort to upgrading the system's hardware capabilities, resulting in more complex designs, higher production costs and usually also higher energy consumption of the final device.

This thesis tries to fill this void by providing a schedulability-oriented optimization framework on the compiler-level. It aims at providing the missing link between schedulability analysis on a system-level where a task is seen as an immutable black box and code optimizations on a compiler-level where schedulability of the final system used to be neglected. As a result, the proposed framework can specifically optimize an embedded multi-tasking system with regard to its schedulability. This work provides both a precise optimization framework based on ILP as well as a genetic approach which is used to compare the results of the ILP approach both with regard to the achieved quality and the needed execution time.

1.2. Contribution

System-level schedulability analysis models tasks as black-boxes, either by neglecting or by abstracting any implementation specific details. On the other hand, code-level analysis and optimization techniques tend to consider each task separately, thus ignoring any subsequent impact on the timing behavior of the overall system.

This work aims at closing this gap. To achieve this, a framework is proposed which offers the possibility of performing schedulability-oriented optimizations on a code-level without any manual interaction by the user. In detail, the following main issues are tackled in this work:

- A framework based on ILP was created to model schedulability-aware optimizations on a compiler-level.
- Due to the optimization potential and as good example for a limited shared resource in a multi-tasking environment, the model is evaluated using an SPM allocation optimization.
- Multi-Tasking specific effects like context switching costs and the effects of the used scheduling algorithm are modeled within the framework and can be automatically considered within the optimization.
- To cover modern embedded architectures which often feature cached memories, caches may be covered within the optimization framework.
- As a result, this work provides a way to link system-level schedulability analysis and code-level compiler optimizations and provides a holistic schedulability-oriented optimization platform. To the best knowledge of

the author, this is the first work which tackles the automatic compiler-based and schedulability-aware optimization of hard real-time systems.

- To be able to rate the performance of the proposed framework both with regard to optimization quality and optimization runtime, a competing framework was created which uses a genetic algorithm for the optimization.
- To prove the practical usage of the proposed methods and framework, all algorithms were implemented into the WCET-Aware C Compiler (`WCC`) framework and evaluated for the ARM7TDMI and Infineon TriCore architectures.

1.3. Structure

This work is organized as follows: Chapter 2 gives an overview of previous approaches and related work. Chapter 3 elaborates on the properties and structure of typical embedded system hardware architectures. Chapter 4 continues by introducing basic definitions for hard real-time systems and the task model used throughout this thesis. Additionally, existing approaches on Worst-Case Execution Time (WCET) optimizations and schedulability analysis are explained. Chapter 5 introduces Integer-Linear Programming and provides an introduction on how this can be used in order to model logical relationships and Boolean expressions. Chapter 6 introduces the WCET-Aware C Compiler (`WCC`) which will be used as a basis for the evaluation of the proposed optimization framework.

Chapter 7 describes an existing WCET-aware optimization framework based on ILP for single-tasking systems. Chapter 8 introduces a new schedulability centered optimization framework using ILP which can be combined with the model introduced in Chapter 7. Further additions to the proposed framework in order to model instruction caches are presented in Chapter 9.

As an alternative approach, Chapter 10 introduces a genetic framework to perform schedulability-aware optimizations.

Chapter 11 describes the evaluation setup. Based on this, Chapter 12 shows and compares evaluation results for the different approaches. This work closes with a conclusion and an outlook on possible future work in Chapter 13.

2. Related Work

Analysis and optimization of embedded hard real-time systems has been researched intensively over the last decades. This chapter aims at giving an overview of previous research related to this thesis. Precise definitions on terms and detailed descriptions of algorithms are not given within this chapter. As far as they are relevant for the upcoming thesis, these will be discussed in Chapters 3 and 4.

Section 2.1 gives an overview of different approaches for the *analysis* of hard real-time systems. Section 2.2 proceeds with an overview of the current state of the art in the *optimization* aspects. Section 2.3 elaborates on mathematical optimization approaches relevant to the scope of this thesis. Finally, Section 2.4 briefly summarizes the open issues which will subsequently be tackled in the course of this thesis.

2.1. Real-Time Analysis

Real-Time analysis techniques can be divided into two fundamentally different approaches: *System-level* and *code-level*.

At the system level, tasks and computing resources are considered as black boxes with given timing properties. Then, based on these given properties, different analyses are applied in order to show whether all timing requirements are provably being met, or not. This allows for a high-level analysis of large-scale complex systems. System-level analyses can therefore not derive safe upper bounds on the worst-case execution behavior of actual tasks in the system. Instead, they expect this information as an input.

Code-level analyses, on the other hand, feature sophisticated techniques in order to calculate the worst-case time a task needs for its execution. They consider machine instructions, hardware architectures and all other implementation-specific details.

2.1.1. System-Level Analysis

Basic requirements for the schedulability of hard real-time multi-tasking systems were proposed more than 40 years ago by Liu and Layland [LL73]. They derived

upper bounds for the load of strictly periodical multi-tasking systems up to which they can safely be scheduled without any deadline misses. Since their original findings, many different approaches have been presented in order to tighten analysis results, improve analysis performance or analyze systems which are no longer triggered periodically.

Joseph et al. [JP86] and Lehoczky et al. [LSD89] provided a technique to calculate the Worst-Case Response Time (WCRT) of periodic multi-tasking systems using a fixed-priority scheduler. The WCRT denotes the worst possible time span from a task being ready for execution until it has finished, including all timing penalties inflicted by other tasks. Later, Lehoczky extended the approach in order to cover arbitrary deadlines [Leh90]. This is later extended by, e.g., Staschulat et al. [SSE05] to account for context switching costs.

Joseph, Lehoczky and Liu and Layland base their analyses on simple systems where each task is described by its WCET, deadline and a fixed period. For systems where tasks are *not* strictly periodically executed, the maximum execution frequency must be set as fixed period in order to achieve safe schedulability estimates. This may obviously introduce significant pessimism. As a result, a system may easily be considered to violate some of its timing constraints in the analysis, despite the fact that it actually does not.

To tackle this issue, several groups started to refine the strictly periodical task model in order to be able to improve schedulability analysis for tasks with complex activation patterns. Tindell et al. [TBW94] extend the approach by Joseph and Lehoczky for systems with jitter and periodically recurring bursts. Further approaches exist for special cases like, e.g., a task model based on differential equations of pendulums [Gue11] for mixed-criticality task sets whose valid time frame for execution is not strictly hard.

An ILP-based approach which aims at analyzing the end-to-end delays of a set of dependent tasks was presented earlier [Kim+12]. Sets of ILP inequations are used in order to model the execution flow over multiple computing units. Then, a maximization objective is used in order to find the worst-case end-to-end latency for each task set. The approach is based on an abstract task model consisting of Best-Case Execution Time (BCET), WCET, a strict period and is limited to fixed-priority scheduling algorithms. Despite its good results with respect to the tightness of the calculated end-to-end latencies, the approach suffers from extreme scalability issues. The approach basically writes out the execution time of each instance of each task within the hyper-period, and subsequently models preemptions by pair-wise comparing each task instance. Subsequently, multiple max constraints are used over the results of these modelings in order to find the worst possible execution traces. According to the authors, their approach is only computationally feasible, if the valid solution space for each variable is limited by preceeding non-

ILP-based analyses. Alternatively, tasks should only feature one single execution path, such that their best-case and worst-case execution times are identical.

The first approaches on real-time analysis only covered so-called *time-triggered* systems. In such systems, the exact points in time at which a task is scheduled for execution is known at system design time. While this complicates the initial system design, it eases predictability of the resulting system and thus schedulability analysis [Kop91]. In contrast to time-triggered systems, tasks are triggered in *event-based* real-time systems by some activation stimuli, e.g., an interrupt which is released on a given incident. These activation patterns are usually not known exactly at system design time. Instead, only bounds on the worst-case activation patterns are known. Gresser [Gre93a] introduced an event model in order to formally describe such systems. Subsequently, Gresser also proposed an approach on the analysis of these event-based systems [Gre93b]. Further schedulability analyses have been proposed by, e.g., Baruah et al. [BMR90; Bar03].

In recent time, event-based system modeling became the de facto standard to model the execution behavior of arbitrarily triggered hard real-time multi-tasking systems. Albers et al. refined the event model and formulated an alternative approach to precisely express the activation patterns of arbitrary systems by using so-called event streams [AS04].

Also building upon the event model, Wandeler [Wan06] proposed a framework which allows for the compositional analysis of complex distributed hard real-time systems. Over the years, several sophisticated frameworks and tools emerged which allow for the holistic analysis and verification of the worst-case timing behavior of distributed embedded systems. The original compositional analysis by Wandeler [Wan06] lead to the creation of the `MATLAB`-based analysis library Real-Time Calculus (RTC) Toolbox [WT19]. Furthermore, approaches by Ernst et al. [Hen+05; Kün+07] resulted in the creation of the analysis tool `SymTA/S` [Hen+05] and its open source derivative `pyCPA` [DAE12].

2.1.2. Code-Level Analysis

All analysis techniques described above have one thing in common: They consider safe numbers for the WCET of each task as given. Code-level analysis subsequently tackles this issue and tries to derive tight yet sound bounds on the WCETs of the tasks in a system. A general overview of typical approaches is presented by, e.g., Wilhelm et al. [Wil+08].

WCET analysis can be divided into different steps. First, the execution time of each instruction and memory access must be analyzed. Then, the worst possible execution path through a task's control flow must be found. Finally, interference

between different tasks, which may lead to additional timing penalties in a multi-tasking environment, have to be considered.

Cousot and Cousot introduced the notion of abstract interpretation [CC77]. This means that machine code is not executed in order to determine its runtime but its execution time is safely approximated. This is done by bundling the ranges of all valid inputs of each instruction into sets and "interpreting" the instruction. This results in a set containing all valid outputs of this instruction which can then be used as input for subsequent instructions. Interval arithmetic is usually used to reduce the size of these sets. With a detailed model of the underlying hardware, this allows for a precise and tight estimation of the worst-case execution time of each instruction. Abstract interpretation has become the de-facto standard in WCET analysis.

Li and Malik presented a model based on integer-linear programming which is able to find the longest path through a program by implicitly modeling the control flow using flow constraints [LM95]. The model operates on the Control Flow-Graph (CFG) of a program and models constraints similar to Kirchhoff's laws which are well known from physics. Like the current flow into electrical components, the valid execution paths within a task are modeled as incoming and outgoing flows through the nodes of this graph.

Several works have subsequently focused on bounding cyclic flows in the graph which may be created by, e.g., recursive function calls or loops [Hea+98; Lok+09a]. Additional research focuses on excluding so-called infeasible paths which denote paths through a task which are logically infeasible (e.g., due to contradicting conditional expressions) [Gus+06; Suh+06].

A different approach is to express tasks as single-path programs [Pus03]. In such tasks, each instruction is always executed. Later instructions conditionally select, whether a result will be used or discarded. Such approaches render traditional WCET analysis de facto unnecessary. Instead, the code can simply be executed or simulated, as its execution time will always be constant, despite its input. However, such an approach does not only require specialized timing predictable hardware, but also usually provides bad performance. The reason for this simply is that in a traditional program, in case of control flow split inflicted by a conditional expression (e.g., an `if() {...} else {...}` construct), only one of the possible paths is executed. For WCET analysis, the path which results in the larger WCET must be considered. However, in single-path tasks, both paths are being executed. As a result, single-path applications are only used in a very small niche of embedded systems.

There exist a number of analyzers which build upon the previously described techniques to provide a full WCET analysis. The `Chronos` analyzer is a tool which aims at analyzing the WCET of tasks running on the so-called Simple

Scalar architecture [Li+07]. Simple Scalar aims at being an easy to analyze timing predictable architecture for research. It is not used in practice and cannot be bought as real hardware.

`aiT` is a commercial WCET analyzer and part of the tool suite a^3 by the German company AbsInt Angewandte Informatik GmbH [FH04]. It allows for the analysis of safe upper bounds on the runtime of single tasks on single CPUs. `aiT`'s main feature is its support for multiple real-world microcontrollers like the Infineon TriCore microprocessor family, and several ARM based microcontrollers. `aiT` currently focuses on the WCET analysis of single tasks. Multiple tasks running on multiple processing units on a multi-core CPU cannot be analyzed automatically. Instead, the user has to manually select and analyze each task and each core separately. Afterwards, the analysis of inter-task eviction penalties and schedulability analysis have to be performed externally by the user.

The WCET-Aware C Compiler (`WCC`) [FL10] is a WCET-aware optimizing C compiler targeting the Infineon TriCore and ARM7TDMI architectures. `WCC` is primarily meant as an optimization tool and tightly coupled to `aiT` which is used for WCET analyses. However, for the ARM7TDMI architecture, `WCC` also features its own internal WCET analysis framework. This was designed and implemented by Timon Kelter as part of his PhD thesis [Kel15] and supports the precise micro-architectural WCET analysis of single tasks running on multiple processing units. It accounts for multi-core specific timing penalties like conflicting accesses to a shared memory bus. Due to the fact that the evaluation of this work was done using `WCC`, a more detailed description of its features is provided in Chapter 6.

With the increasing complexity of microprocessors, safe yet tight static timing analysis becomes harder as well. The reason for this is, that static timing analysis needs a profound knowledge of the underlying hardware in order to be able of giving any safe estimation on the timing behavior of a piece of code. To avoid this, Burns et al. proposed a probabilistic measurement based timing analysis [BE00]. The approach is based on probabilistic extreme value theory and tries to deduce a so-called probabilistic WCET from a set of measured execution times. In order to provide a reliable probabilistic estimate on the WCET, the set of measured execution times must be representative. Santinelli et al. aim at tackling this issue by providing a formal guideline on the application of extreme value theory for measurement based timing analysis [SGM17]. However, the problem remains that by definition the probabilistic WCET is not a *safe* guarantee of the worst-case timing behavior. Additionally, the measurement based timing analysis does not provide any insights on the worst-case timing behavior of small parts of a system's task but only provides estimates on the WCET of the task as a whole.

In multi-tasking environments, tasks can conflict with each other, e.g., due to conflicting accesses to a cache. This results in changes of the tasks' execution timing behavior. The problem is described by, e.g., Staschulat et al. [SSE05] who

also extended existing response time analyses to account for these timing penalties in system-level analysis. Over the last years, especially Cache-Related Preemption Delay (CRPD) has been researched intensively. Most notably, Kleinsorge [KFM11; Kle15] provides a sophisticated framework to precisely bound such conflicts. An alternative approach which is commonly used in literature is given by Altmeyer et al. [ADM12].

[Dav+18] aim at providing a holistic approach which integrates multiple aspects of code-level analysis into a system-level schedulability analysis. However, their approach is currently limited to constrained deadlines (i.e., deadlines smaller than or equal to the minimal inter-arrival time) and fixed-priority scheduling. Furthermore, the approach only works for simple microprocessor architectures which do not suffer from timing anomalies. Finally, task execution time is modeled by explicitly listing possible execution traces which may easily become computationally infeasible.

2.2. Optimization of Real-Time Systems

Numerous optimization techniques of hard real-time systems have been proposed over the last decades. The main goal of such optimizations is to transform a system which violates at least some of its timing constraints in a way which results in a system which will provably always meet all requirements.

Obviously, one way in order to achieve this is to modify the hardware the system is running on. Replacing a microcontroller by a more powerful one or replacing given memory by faster memory will – at some point – result in a system which meets all its requirement. However, this is not within the scope of this thesis. Instead, the idea of this thesis is to only consider the available hardware and subsequently try and use the given resources as efficient as possible with regard to the system's schedulability constraints. Therefore, optimization approaches which require changes to the underlying hardware will not be discussed in the following.

Similar to the analysis of hard real-time systems, optimization techniques can also be divided into system-level and code-level optimizations. The remainder of this section will first give an overview of possible approaches on system level and then proceed with code-level optimizations.

2.2.1. System-Level Optimizations

One central method to optimize a multi-tasking hard real-time system with regard to its schedulability is the selection of a suitable scheduler and, in case of selecting a fixed-priority scheduling algorithm, choosing an optimal priority for

each task within the system. For dynamic-priority scheduling where priorities are calculated at system runtime, common scheduling algorithms are Earliest Deadline First (EDF) and least laxity scheduling [HTT89]. In EDF, that task whose deadline is occuring the soonest is chosen to be executed. In a least laxity scheduler, that task with the smallest laxity is assigned the highest priority. The laxity is defined as the time interval between a task's deadline and the time it was triggered to be executed, minus the time it has already been running. Both EDF and least laxity have been proven to be optimal scheduling algorithms when neglecting scheduling and task preemption costs. Despite being proven theoretically optimal, least laxity scheduling is rarely found in real-world hard real-time systems. The reason for this is that the scheduler has to re-calculate all priorities of all tasks continuously, leading to a high timing overhead due to the scheduler and possibly many context switches between tasks.

Alternatively to dynamic-priority scheduling schemes, fixed-priority scheduling algorithms can be used. For these algorithms, each task is assigned a fixed priority at system design time. Common fixed-priority algorithms are rate-monotonic scheduling and deadline-monotonic scheduling. They have proven to be optimal for the class of fixed-priority scheduling algorithms [LL73].

In practice, choosing one scheduling algorithm over another may not only be done based on scheduling analysis results. Somtimes, a given scheduling scheme or priority assignment may result from regulatory constraints or historic reasons. For EDF, Deadline-Monotonic Scheduling (DMS) and Rate-Monotonic Scheduling (RMS) scheduling, the execution behavior solely depends on the timing requirements defined by the system designer. The actual task's execution behavior (i.e., its WCET) does not change the priority assignments.

At system level, so-called *sensitivity analyses* can be used to optimize a system which violates at least some of its timing constraints. Despite its name, sensitivity analysis can be used to tune the parameters of the system in order to repair a previously not schedulable system. The key idea behind sensitivity analysis is to identify those timing parameters of a system which bear the most positive effect towards overall schedulability with the least needed modification. These timing parameters include the execution frequency of a task, its respective deadline but also its WCET [BDB06; ZBB11; Neu+13].

Unfortunately, the practical usage of sensitivity analysis may be somewhat limited. If a deadline or execution frequency of a software task is predetermined by the surrounding physical systems, changing these timing parameters is often practically infeasible. Reducing the WCET of a task will surely improve schedulability. However, as a high-level approach, sensitivity analysis considers tasks to be black boxes. As a result, any recommendations on WCET improvements are purely speculative. Sensitivity analysis is not able to predict whether a given task's WCET can be reduced by the proposed amount. It can also not give any

hints on *how* a task's WCET can be reduced. Furthermore, negative side effects on the timing behavior of one task, inflicted by code-level modification of another task, is also not modeled, because on the system-level side, any knowledge about the inner structure of tasks is missing.

2.2.2. Code-Level Optimizations

An optimizing compiler may perform a number of code-level optimizations in order to improve a task's runtime behavior. Common compilers like `LLVM` or `GCC` use heuristics in order to improve the Average-Case Execution Time (ACET) of the system. ACET optimizations like loop unrolling, function inlining or instruction scheduling are well-known and discussed at length in literature [Muc14]. Despite their aim at optimizing the average runtime, those optimizations do not have any information about the actual impact of an optimization on the resulting execution time. Profiling tools like `gprof` which come with the `GCC`, allow the user to support the compiler with some information about which parts of the program are executed frequently and which ones are executed only occasionally. The compiler can then try to optimize frequently used parts more aggressively. However, on the one hand, the compiler still neither features a timing model nor does it have an exact model of the target architecture. As result, especially in the domain of embedded systems with limited memory, possible negative effects like, e.g., additional cache misses due to aggressive function inlining or loop unrolling, cannot be predicted. On the other hand, the sheer idea behind these standard optimizations is to optimize the *average* execution time. The worst-case timing behavior, which is the key property of a hard real-time system, is explicitly not considered in the optimization techniques. Furthermore, multi-tasking effects, execution frequencies or task periods are also not modeled at all. As a result, those standard optimizations are neither suitable for specific schedulability-aware optimizations nor even usable for optimizing a single task's WCET. Becker at al. recently proposed an approach to use mainstream compilers like `GCC` in order to optimize the WCET [BC18]. Their idea is to provide `gprof`-like profiling information to `GCC` which in fact contains information on the worst-case execution paths rather than average-case profiling information. However, their work shows that even then, `GCC`'s optimizations may easily result in negligible improvements on the WCET. Some evaluation cases even result in a WCET which is more than 70 % higher than the standard `-O2` optimization level without profiling information.

Apart from holistic optimization frameworks like `WCC` [FL10], code-based WCET-aware optimization techniques have been proposed by numerous researchers. Lokuciejewski et al. [LM11] give an overview, with a focus on the optimizations in `WCC`. Exemplary, cache-aware basic block positioning was presented by [FK11].

A WCET-aware register allocater was proposed in [FSS11]. Some works also focus on WCET-aware source code optimizations, e.g., Lokuciejewski et al. [LKM10].

Suhendra et al. present a WCET-oriented static data SPM allocation based on ILP [Suh+05]. They model the control flow of a task using integer-linear (in)equations and are subsequently able to set the minimization of the task's WCET as objective function for the ILP solver. Subsequent constraints model timing gains for each data object if it is moved to the fast but very small SPM. Their model was later used by Falk et al. for a static instruction SPM allocation [FK09]. These techniques are also used within this thesis to illustrate the potential of the presented schedulability-aware optimization framework.

All these optimizations aim at minimizing *the* WCET of a single task, neglecting negative side effects on other tasks in case of a multi-tasking system, as well as limited resources which must be shared among all tasks.

Only relatively few previous works exist on the optimization of multi-tasking hard real-time systems. They mostly aim at separating tasks from each other in the system [PLM09]. Then, subsequently, the optimization can proceed by simply minimizing the sum over all WCETs of all tasks in a system, yet again neglecting any timing constraints. This however, clearly does not lead to optimal results, as the separation of the tasks from each other restricts their optimization potential without including any knowledge on which task even needs to be optimized in order to fulfill all timing constraints.

2.3. Optimization Methods

WCET-oriented optimizations can usually be represented as a large combinatorial problem. E.g., the static data SPM allocation by Suhendra et al. [Suh+05] can easily be considered as a form of the knapsack problem: A number of basic blocks is to be allocated to the SPM which has a limited capacity, such that a given cost function (here: the WCET) is minimized. For other optimizations like the static instruction SPM allocation, the problem gets even more complicated, as assigning one basic block to the SPM might change the cost of the block itself and of some of the remaining blocks. As a result, as shown by Richard Karp [Kar72], WCET optimizations can be considered as NP-complete optimization problems.

Mathematical approaches in order to solve such problems can be divided into two categories: Heuristics and optimal approaches.

Heuristics aim at finding a *good* solution within a reasonable amount of time. Both definitions of "good" and "reasonable amount of time" are somewhat fuzzy terms. In the end, the user of an approach must decide, whether an optimization is sufficiently fast or precise. While this thesis does not aim at defining "sufficiently

fast", any optimization result is considered as "sufficiently precise", if the result of the optimization leads to a system which adheres to all of its design constraints.

Optimal optimization algorithms, on the other hand, are able to deterministically find an optimal solution with regard to a given objective function.

The probably most famous commonly used heuristic approaches are genetic algorithms as proposed by Goldberg [Gol89] and machine learning based approaches [Sam88]. Some approaches exist trying to apply machine learning techniques to the domain of WCET analysis and optimization [Lok+09b; Bon+17]. However, the approaches' results are quite mixed. The probably main reason which objects the usage of machine learning is the lack of sufficiently large learning sets. On the other hand, genetic algorithms have previously been used successfully [FS06; Pla+11; OLF17].

For optimal approaches, formulating combinatorial problems as a set of linear (in)equations has been proven a powerful tool. So-called *SAT solvers* can convert such (in)equation systems into a Boolean Satisfiability Problem (SAT) and subsequently provide one valid solution. A popular example for such a solver is `z3` by Microsoft. Alternatively, the (in)equation system can be accompanied by an objective function. Numerous tools exist to solve the subsequently resulting so-called *linear program*. Some of the most popular solvers are `lpSolve` [Lps19], `CPLEX` by IBM [Cpl19] and `Gurobi` [Gur19]. Their solving algorithms mainly base on the simplex method proposed by Dantzig [DOW55]. Gomory extended the model by Dantzig to cover so-called *integer*-linear problems, where variables are integer-valued [Gom58]. A more efficient approach for large-scale problems was proposed later on by Karmarkar [Kar84].

2.4. Summary

Schedulability analysis techniques operate on a high abstraction level. They can safely predict whether a given system may violate its timing constraints or not. However, due to their high level of abstraction, they cannot be used efficiently for code-level optimizations.

Code-level analyses provide detailed insights on the runtime behavior of the code being executed on the embedded system. The analysis results can be used by code-level optimizations for the reduction of the worst-case timing behavior. This has been been researched intensively in the past. However, these optimizations do not account for the scheduling behavior of the overall system, but rather aim at minimizing *the* WCET of a task, without regard to limited optimization resources or timing constraints.

As a result, schedulability-aware code optimization is usually a tedious task where the system designer has to play with compiler optimization flags and approaches, hoping that finally one combination of optimizations will lead to a schedulable system.

This work aims at closing this gap between system-level analysis and code-level optimization. Code-level optimization techniques are extended by a schedulability-aware optimization framework. This way, code-level optimizations can precisely predict their impact on the schedulability of the overall system, paving the way for automated schedulability-aware compiler optimizations.

In order to be able to provide a code-level optimization which is aware of its impact on the overall system's schedulability, a number of open challenges have to be tackled:

Multiple Tasks In modern hard real-time systems, an arbitrary number of tasks may be defined by the system designer. An optimization framework has to account for possible interference between the tasks and scale well enough to cope with optimizing multiple tasks.

Arbitrary Activation Patterns Activation patterns of tasks may vary from simple time-triggered and strictly periodical systems to complex event-triggered systems with complex activation patterns and deadlines. In order to minimize the level of pessimism in the optimization, the framework should be able to precisely model these activation patterns within the underlying optimization model in order to safely estimate the system's schedulability.

Scheduling Algorithm The system designer should be able to freely choose between fixed-priority and dynamic-priority scheduling. The optimization framework should either be decoupled from any concrete scheduling algorithm or support multiple common scheduling algorithms.

Architectural Features Each task's WCET and therefore also its WCRT is heavily dependent on architectural features like, e.g., caches. The optimization framework should therefore be able to optionally model such features.

Platform Dependency Compiler optimizations are heavily dependent on the target architecture's features and assembly instruction set. The optimization framework should, however, be as generic as possible in order to be used on different architectures.

3. Embedded System Architectures

Nowadays, computing systems are not only used in personal computers or as large-scale server infrastructures but are often small and dedicated systems embedded into a specific technical context. These systems can be found in a broad variety of applications, ranging from from multimedia decoders over household appliances to safety-critical control systems in cars or airplanes.

Although these systems' characteristics differ significantly, they have some common properties: They are dedicated systems tailored towards their specific use-case and often act completely invisible to the end-user. They interact with their surroundings and are subject to certain runtime constraints. Loosely based on the definition by Marwedel [Mar18, p. 1-2], an Embedded System may therefore be defined as follows:

Definition 3.1 (Embedded System)
An Embedded System *is a computing system which is embedded into a larger product. It is designed in order to fulfill a dedicated task with limited resources and is subject to one or several design and runtime constraints.*

Fig. 3.1 provides an abstract view on the interaction of a typical embedded system with its environment. The embedded system retrieves data about its environment through sensors and analog-to-digital conversion units (ADC). This data is then processed within the embedded system's software components. Depending on these calculations' results, a digital output is generated, converted into an analog signal by a digital-to-analog conversion unit (DAC). This signal is finally sent to an actor (e.g., an engine) which will, to some extent, influence the physical environment.

Usual design constraints of an embedded system include, but are not limited to, cost efficiency, dependability, security, and the adherence to given timing constraints. These timing constraints directly arise from the technical context the system is embedded into. For example, a video decoding unit must decode each frame within a certain amount of time in order to provide a smooth replay of the decoded video. It is important to emphasize that these timing constraints are coupled to some biological or physical constant, like the human eye or the speed of a moving car. The system designer is bound to these timing constraints due to

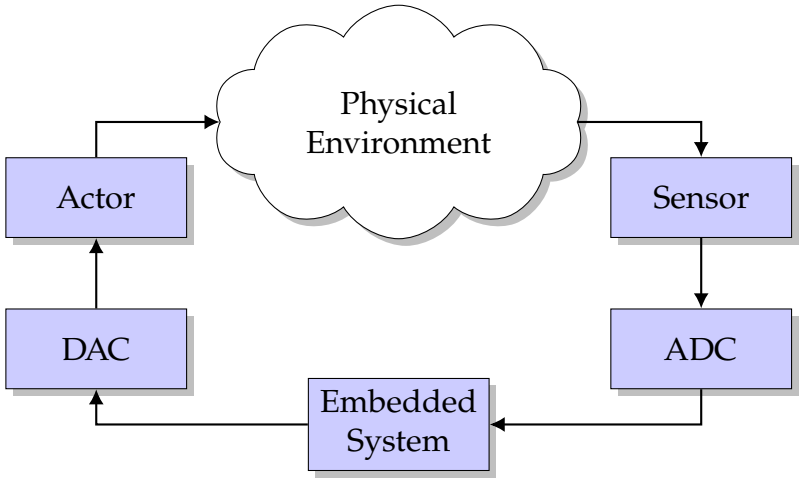


Figure 3.1: Typical composition of an embedded system. The embedded system receives data from the physical environment and influences the physical environment in some way depending on the received input.

the nature of the technical context and may not modify them arbitrarily. Systems which have to comply to given timing constraints are called *Real-Time systems* as previously defined by, e.g., Manacher [Man67].

Definition 3.2 (Real-Time System)
A computing system which has to finish its calculation and return the corresponding result within a given amount of time is called a real-time system. [Man67]

A real-time system can further be categorized as *soft* or *hard* real-time system. Manacher [Man67] defines *soft* real-time systems as systems where tasks must finish their execution within "reasonable" time, yet tolerate occasional lateness. *Hard* real-time systems, on the other hand, are systems where the tasks are given strict deadlines by their surrounding physical environment. When the task does not finish in time, the result is no longer useful in any way. By this definition, a *late* result is equivalent to *no* result at all and the strict adherence to all timing constraints becomes a functional requirement of the system.

The following sections give a brief introduction into the structure of embedded systems as far as this is needed for a better understanding of this thesis. Extensive introductions are given by, e.g., [HP12; Sta16; Mar18].

3.1. Hardware Architectures

Modern computing systems can be categorized by the level of generalization of the underlying architecture. Common examples for processing units range from Application-Specific Integrated Circuits (ASICs) whose hardware is unchangeably tailored towards one specific task [Mar18, p. 139] over more generalized but still special-purpose processors like Digital Signal Processors (DSPs) to general-purpose control units.

Within these categories, architectures may be further classified by their instruction type. E.g.,Very Long Instruction Word (VLIW) processors follow the paradigm to code multiple operations into one instruction packet, allowing for easy parallelization on the cost of complex hardware and increasingly complex compilation [Mar18, p. 152].

Reduced Instruction Set Computer (RISC) architectures offer a small number of different instructions, typically resulting in less specialization but also simplifying the underlying hardware of the computing unit, as well as the effort necessary for creating a working system. Although each instruction is way less powerful, one instruction can be executed within a few cycles of the processing unit's operating clock. As result, as long as the timing penalties of getting the instructions from the system's memory into the execution unit of the microprocessor can be handled, RISC machines can provide significant performance.

As a compromise between VLIW and RISC, Complex Instruction Set Computer (CISC) machines offer a more complex instruction set than RISC, thus needing less memory accesses to fetch the code needed to solve a given task [Mar18, p. 144]. Despite this advantage, most embedded systems, especially in the domain of real-time systems, rely on the RISC architecture. Apart from reduced production costs due to a relatively simple hardware design, the better timing predictability of RISC architectures is a major factor. By definition, hard real-time systems must provably comply to given timing constraints. The more complex the underlying hardware architecture, the harder it gets to provide safe but tight estimates on the maximum execution time of a given sequence of instructions.

However, due to the fact that the whole domain of embedded systems is relatively application-specific, often multiple additional design constraints like a small program size or a limited amount of energy consumption must be kept. This leads to architectures offering some features which originally stem from different architecture classes. Therefore, the frontiers between different architectural concepts are somewhat blurry.

Embedded microprocessors like the Infineon TriCore [INF08] are based on the RISC architecture but also feature multiple instructions which are typical for a DSP. To show applicability of the upcoming contributions of this thesis, the

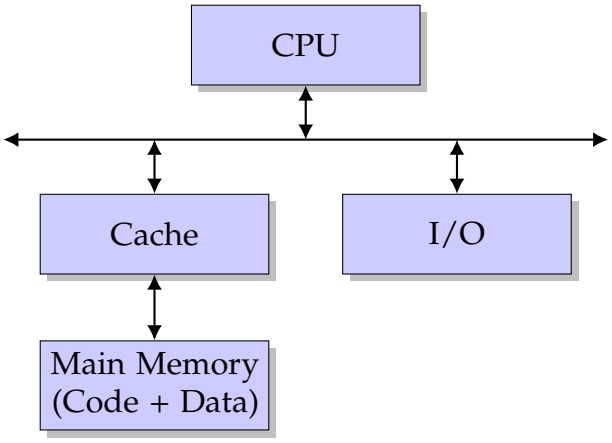


Figure 3.2: Basic structure of an embedded system using the Von-Neumann Architecture.

thesis focuses on both a classical RISC microcontroller, namely the ARM7TDMI architecture represented by the NXP LPC2880 microprocessor and the much more complex Infineon TriCore TC1796 microprocessor whose RISC based central processing unit is accompanied by a DSP instruction set and several instructions for floating point arithmetic.

3.2. Memory Architectures

Besides their differences in the instruction set, the ARM7 and TriCore architecture differ in the way of addressing data and code. The ARM7TDMI is based on the so-called *von-Neumann architecture* [ARM04, p. 3-2] depicted in Fig. 3.2, which uses a common memory for both data and code [Sta16, p. 105].

The Infineon TriCore microcontrollers on the other hand are built according to the so-called *Harvard* architecture, physically separating a program's instructions from its data [HP12, p. L-4] [INF08]. The block diagram of the Harvard architecture is shown in Fig. 3.3.

Due to the separate memories for code and data, the TriCore is able to both fetch a new instruction word and to load application data into a register at the same time. The ARM7TDMI, on the other hand, has to either delay data or instruction retrieval as both code and data reside in one memory accessed over one common bus.

From a hardware point of view, the memory does not distinguish between instructions and data. It solely stores sequences of zeroes and ones which are subsequently interpreted as either data or instructions by the CPU. To ease

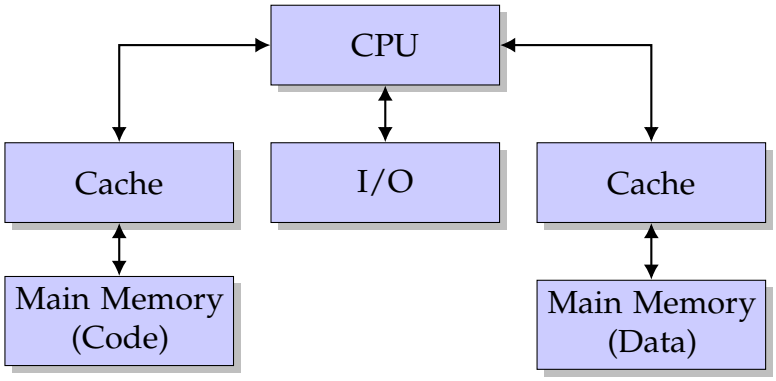


Figure 3.3: Basic structure of an embedded system using the Harvard Architecture.

descriptions, the remainder of this chapter will solely refer to "data" when describing the behavior of memory.

The memory must be *addressable* in order for the CPU to be able to select a given region of the memory where data is stored to or retrieved from. The number of addressable units within the memory is dependent on the architecture. Common bit widths A used to address memory are 8, 16, 32, or 64. This subsequently results in 2^A addressable units in the memory. The size of an addressable unit in the memory is also platform dependent. The microcontrollers which are used for evaluation purposes in this thesis feature 32 bit wide addresses. They address memory at byte-level.

3.2.1. Cache

As explained above, RISC architectures feature a fast execution of one single instruction but need, compared to other architectures, a relatively large number of instructions to solve a given task. To allow for a high computing load and to reduce idle times, it is therefore crucial that instructions can be retrieved as fast as possible from the memory.

Unfortunately, fast memories are expensive with both respect to needed chip size and production costs. *Caches* are often used in order to compensate for this problem. A cache is a small but fast memory which aims at buffering data from main memory in order to provide faster access. Caches are transparent to the processor and any running application and are managed completely autonomously in hardware by the so-called cache controller. The cache controller decides which piece of data from the memory is stored to which part of the cache memory. It also manages which parts of the cache memory are evicted to make room for other data from main memory. If a certain piece of data is present in the

cache, data retrieval can solely take place between the CPU and the cache. There is no need to communicate with the slower main memory.

If data is to be stored to the cache, the cache's behavior may differ. A *write-through* cache will store any new data in the cache and will simultaneously write it directly to the main memory, thus there will be no timing gain by the cache. A *write-back* cache will buffer written data and only write it back if it is to be evicted by other data. This can heavily improve the performance in case of frequent writes to the same address in memory. However, it is hard to predict the cache's timing behavior.

In order to further bridge the gap between access speed and costs, caches can be daisy-chained to form a so-called cache hierarchy. This means that a very fast but very small cache is followed by one or more increasingly slower but larger caches, until finally the actual main memory is accessed. This setup is mainly found in environments which feature very large main memories like general-purpose computers with several gigabytes of memory or in distributed systems with very slow remote memories. The main focus of this work, however, is on small single-core embedded architectures. The platforms analyzed in this work do not feature hierarchical caches. They are therefore not discussed in the following.

An exhaustive introduction into the functioning of caches is given by, e.g., [Sta16, p. 144-188]. This section is based on [Sta16] but solely focuses on those aspects of caches which are needed in the ongoing of this thesis.

Caches are mainly distinguished by their mapping function and their replacement policy. The mapping function determines which block from the main memory will map to which line of the cache. The replacement policy determines which chunk from the cache will be evicted to make room for a new block.

A cache consists of a number of k cache *lines*. Each line contains one *block* of data from main memory, plus additional control information. The control information is used to store which concrete block from main memory is residing in a given cache line, as well as management information needed for the cache controller's replacement policy.

3.2.1.1. Mapping Functions

There are three different approaches for the mapping function: *Direct* mapping, *fully associative* mapping and *set associative* mapping.

In direct mapping, the mapping of memory blocks to cache lines is fixed. For each memory block, there exists exactly one valid cache line where its data may be stored. While being simple to implement, such mapping does usually not offer best performance due to suboptimal usage of the cache's resources.

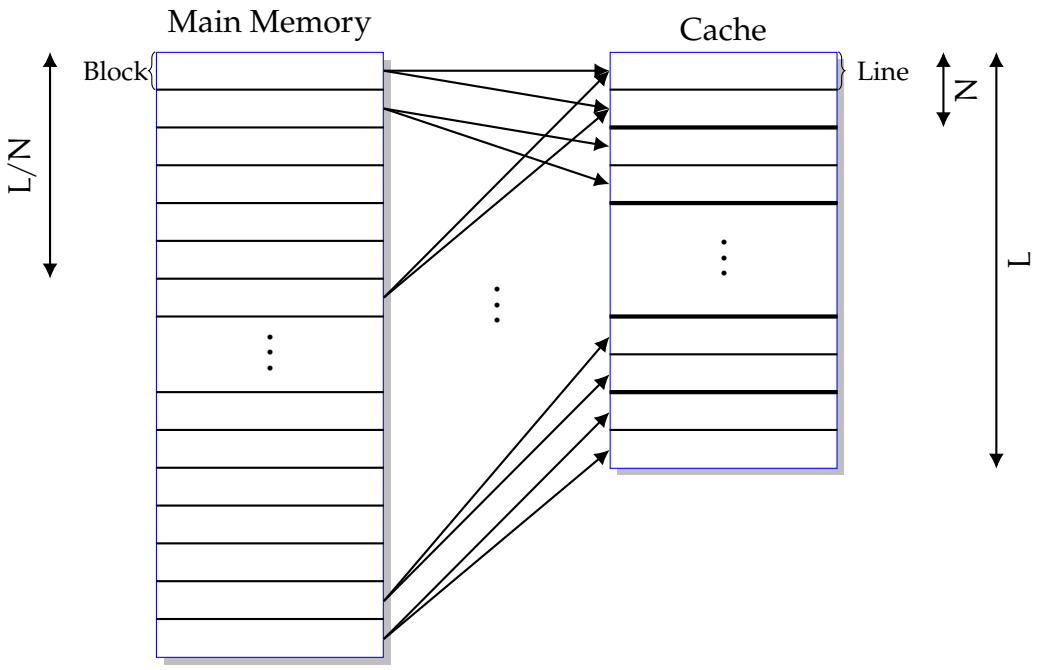


Figure 3.4: Typical structure of a cache. L describes the number of lines in the cache. The cache's associativity is given by N. Figure adapted from [Sta16, p. 165].

In a fully associative cache, a memory block may be assigend to *any* cache line. The cache controller must dynamically keep track of which line contains which memory block. This mapping provides best usage of the cache's resources but also leads to a highly complex cache controller.

As a trade-off between direct and fully associative mapping, set associative mapping can be used. In a so-called N-way set associative mapping, the cache is logically divided into different *sets*. There exists a direct mapping of a memory block to a cache set. Within each set, there exist N cache lines which act like a fully associative cache. Fig. 3.4 shows the typical structure of a set-associative cache. L describes the number of *lines* within the cache. N denotes the *associativity* of the set-associative cache, and therefore equals the number of cache lines within one set. Subsequently, the number of sets S is given by $S = L/N$. It can be seen that each block from memory can be cached in N cache lines. Additionally, due to the fact that the cache is smaller than the main memory, multiple memory blocks map to the same cache lines.

In order to identify to which set a given memory block is allocated to, as well as to identify the concrete block within all lines within one set, the memory address is used. Fig. 3.5 shows how this is achieved. The memory address consisting of γ

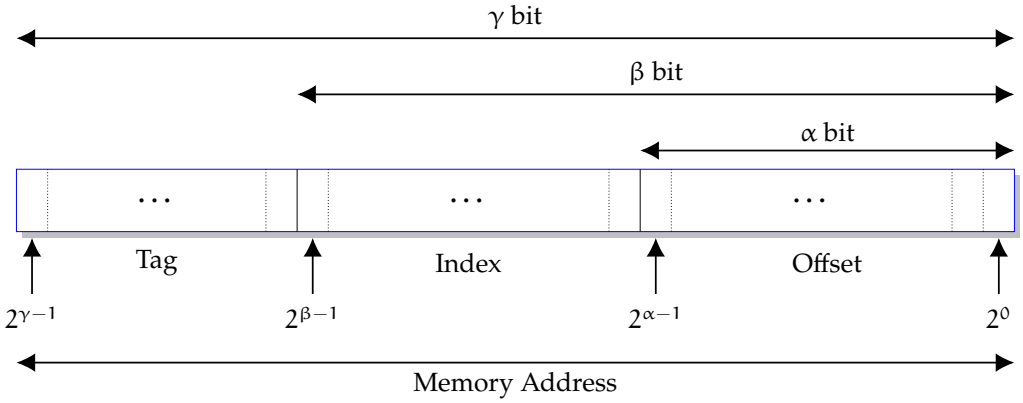


Figure 3.5: Mapping of a memory address with γ bits to its cache parameters [HP12, App. B].

total bits (e.g., $\gamma = 32$ in a 32 bit system) is divided into a *tag*, an *index* and an *offset*. The index bits denote the set which a memory block will be directly mapped to. The tag bits identify the concrete memory block within the set. Finally, the offset bits denote the concrete memory unit (e.g., the byte in a byte addressable memory) which is being addressed. As a side note, an N-way set associative cache with $N = L$ equals a fully associatively mapped cache. On the other hand, the behavior of an N-way set associative cache with $N = 1$ is equivalent to a directly mapped cache.

3.2.1.2. Replacement Policies

Obviously, a cache cannot hold all blocks from main memory at the same time (otherwise, the main memory could simply be removed). The replacement policy is used to decide which block should be evicted from the cache if a new block has to be added. Therefore, the replacement strategy is crucial for the number of cache hits and subsequently for the cache's performance. The most important replacement strategies are:

FIFO First In First Out (FIFO) replacement policy will always evict that block from the cache which was loaded earliest. It does not keep track of the access patterns.

LFU When using the Least-Frequently Used (LFU) replacement policy, the cache controller keeps track of how often a cache block was actually accessed. When a cache block must be evicted, the block which has the least number of accesses is removed from the cache.

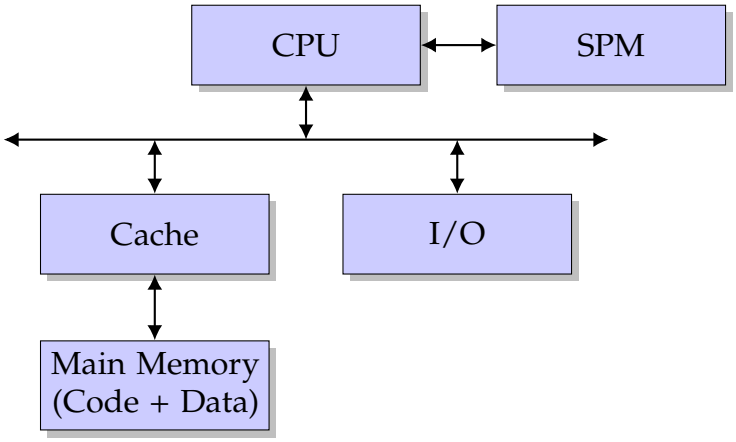


Figure 3.6: Basic structure of an embedded system using the Von-Neumann Architecture with an additional SPM.

LRU The Least-Recently Used (LRU) replacement policy keeps track of the age of the last usage of each cache block. If a cache block is accessed, its age is reset. If a block must be evicted, the block with the highest age is removed.

Random Random replacement policy will select one random block in the cache which is evicted.

Common embedded systems like the Infineon TriCore often use the LRU cache replacement policy. On the one hand, this follows the idiom of the so-called temporal locality. I.e., it is assumed that any instructions or data which have been used recently, might also be likely reused in the future (i.e., due to usage within a computational loop). Data or instructions which have not been used for a long time are thus considered to be unlikely to be reused in the near future and can thus be removed from the cache.

On the other hand, LRU replacement policy has been proven to be optimal with respect to its worst-case behavior analysis. It has been proven that, e.g., caches with FIFO replacement strategy cannot be analyzed as precise as with LRU [Rei08]. In a hard real-time system, whenever some behavior (like a cache hit or miss) cannot be predicted safely, the worst-case behavior must be assumed. Therefore, if the cache replacement policy is not predictable, the cache might bring benefits to the system performance – however, this cannot be accounted for in the system analysis and does therefore not help when it comes to analyzing or optimizing a hard real-time system.

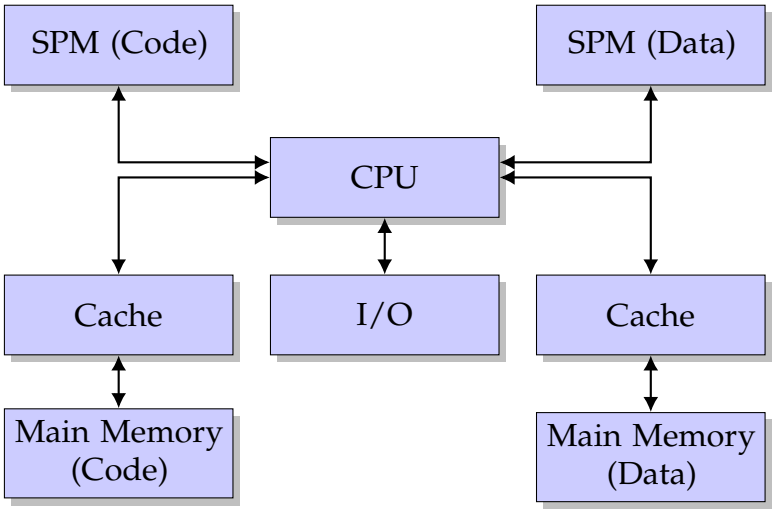


Figure 3.7: basic structure of an embedded system with SPMs using the Harvard Architecture.

3.2.2. Scratchpad Memory

As discussed above, caches can be used to mitigate the effects of slow main memories. Their transparent behavior makes them easy to use. However, this also makes them hard to analyze. Additionally, in a hard real-time system which has to comply with certain timing constraints, the most commonly or recently used data may not be the one which is crucial for the timing-critical parts of the system. Therefore, a Scratchpad Memory (SPM) can be used as an alternative. Fig. 3.6 shows the basic structure of a von-Neumann hardware architecture with an additional SPM. The SPM is directly connected to the CPU and not via the system bus which is used to interface with the "regular" main memory and any I/O devices.

From a hardware perspective, SPMs are usually implemented in the same technology as a cache. In fact, for the Infineon TriCore TC1796 microcontroller, cache and SPM are identical physical memories. The system designer can configure the microcontroller to use the memory as either SPM or cache.

Fig. 3.7 shows the logical structure of a general Harvard architecture with an SPM memory. Because Harvard architectures use separate memories for code and data, this separation can also be found in the design of the SPMs.

From a programming perspective, an SPM is a regular addressable memory. It uses its own address space and is used identically to the system's main memory. This means that the programmer of a system must actively decide which instructions and which data should be allocated to the SPM and which should not

and must adjust the program code accordingly. Because the SPM is just a regular memory, this does not have to be decided statically at compile time, but may even be decided dynamically during runtime of the system. However, because the worst-case behavior in case of a dynamic decision is always hard to predict safely, usually static allocation is preferred in the domain of hard real-time systems.

4. Real-Time Systems

This chapter covers different approaches for modeling and analyzing the timing properties of real-time tasks and systems which are needed in this thesis. After introducing fundamental definitions in Section 4.1, Section 4.2 tackles the analysis of the worst-case runtime behavior of an individual task in a system. Section 4.3 introduces different task models which can be used to formally describe timing properties like activation patterns and the deadline of a task. Afterwards, Section 4.4 gives an overview of important scheduling algorithms in the domain of hard real-time systems. Based on the findings and definitions of the previous sections, Section 4.5 then presents analysis techniques in order to calculate the WCRT of the tasks in a system, and presents schedulability analysis algorithms which predict whether a given system is schedulable or not.

4.1. Fundamentals

To be able to verify the correct behavior of a hard real-time system, a formal model describing both the system itself and its software is needed.

Definition 4.1 (Task)
A task τ_i *is a pre-defined software routine which is executed on a hard real-time system and fulfills a dedicated purpose. Each task is assigned a* deadline D_i *relative to the point in time at which the task becomes ready for execution. The index* i *is used to identify a distinct task. The time interval between the task finishing its execution and the task being ready to be executed* must *be smaller than or equal to* D_i *under any circumstances.*

Using this definition of a task, Definition 3.2 of a real-time system can now be extended with a focus on the software being executed on the system:

Definition 4.2 (Real-Time System)
A real-time system *is a computing system which consists of a predefined task set* Γ *which is executed on a given hardware platform. The task set* Γ *contains a given number* N *of tasks:* $\Gamma = \{\tau_0, \ldots, \tau_{N-1}\}$*. All tasks in the task set must finish their execution within a given time budget.*

Note that the definitions of a real-time system and a task do not yet make any assumptions on *how often* or *when* a task is actually being executed. To cope with multiple tasks which compete for the same resource (e.g., CPU processing time), the real-time system contains a so-called *scheduler* which decides which task is executed at which point in time.

Additionally, the definition of a task does not imply any limitations on the execution pattern of the task. A task in this broad sense may be executed once, periodically or in a completely arbitrary fashion. Formalisms and limitations are introduced in the upcoming sections in order to model the timing behavior of a task in more detail. These are not mandatory for a system to qualify as a real-time system but are solely needed for an analysis to *verify* or *disprove* that a task τ_i will always meet its deadline D_i.

Usually, a task will need a different amount of processing time depending on its input and the initial state of the hardware on which the task is being executed. The *minimum* amount of time until the task returns its result is called the Best-Case Execution Time (BCET). A trivial and safe bound on the BCET is 0.

Definition 4.3 (Best-Case Execution Time)
The Best-Case Execution Time (BCET) c^-_i *of a task* τ_i *is the* minimum *time spent on one execution of the task for any input and initial hardware state, if the task does not have to compete for any resources with other tasks.*

The *maximum* net amount of time the task needs to process its input data is called the Worst-Case Execution Time (WCET).

Definition 4.4 (Worst-Case Execution Time)
The Worst-Case Execution Time (WCET) c_i *of a task* τ_i *is the* maximum *time spent on one execution of the task for any input and initial hardware state, if the task does not have to compete for any resources with other tasks.*

Giving a safe upper bound lower than "infinite" for the WCET of an arbitrary task is impossible, as it would imply solving the halting problem which was proven unsolvable by Turing [Tur37].

However, if computing algorithms are relatively simple and the maximum ranges of input data are known, it is possible to predetermine, e.g., safe upper bounds on the number of loop iterations or recursion depths. Combined with a profound knowledge of the program's target platform, a safe upper bound on the maximum execution time of a program can be obtained.

4.2. WCET Analysis of a Task

A task of an embedded system interacts with its environment by receiving some kind of data, performing certain operations on it and returning some result. Apart from very simplistic tasks, different input data will most likely lead to different paths being taken through the task's algorithmic structure. As a result, different calculations are performed and the execution time will vary. To reduce the analytical overhead, machine instructions which have to be executed consecutively, are often considered as one entity within the analysis. These snippets are called *basic blocks*. The definition of a *basic block* is given by, e.g., Muchnick [Muc14] as follows:

Definition 4.5 (Basic Block)
"A basic block *is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them." [Muc14, p. 173]*

The basic blocks can now be connected as a directed graph structure to model all possible paths through the task. This structure is called the task's Control Flow-Graph (CFG). Based on [All70] and [AG04, p. 218], the following definition can be given:

Definition 4.6 (Control Flow-Graph (CFG))
A Control Flow-Graph (CFG) is a directed graph with one distinguished start node. Each basic block of a task is represented as a node in the graph. If a basic block A *can be directly followed by basic block* B *in the task's execution, then there is an edge from the node representing basic block* A *to the node representing basic block* B*. [AG04, p. 218]*

This CFG can then be used as a basis for all upcoming WCET analyses.

Modern microprocessor architectures may feature, e.g., parallel pipelines, branch prediction units, caches and multi-cycle instructions. As a result, given machine instructions within a basic block may take a different amount of time, depending on their input data and previously executed basic blocks. These differing executions of one basic block are called *execution contexts*.

Fig. 4.1 illustrates the distribution of execution times for a fictional task in an embedded hard real-time system. $WCET_{REAL}$ denotes the actual worst-case timing behavior of the system, as defined by Definition 4.4. Often, this actual case is not triggered during testing and regular operation. Subsequently, $WCET_{OBS}$ marks the worst-case timing behavior which can be *observed* during evaluation and testing.

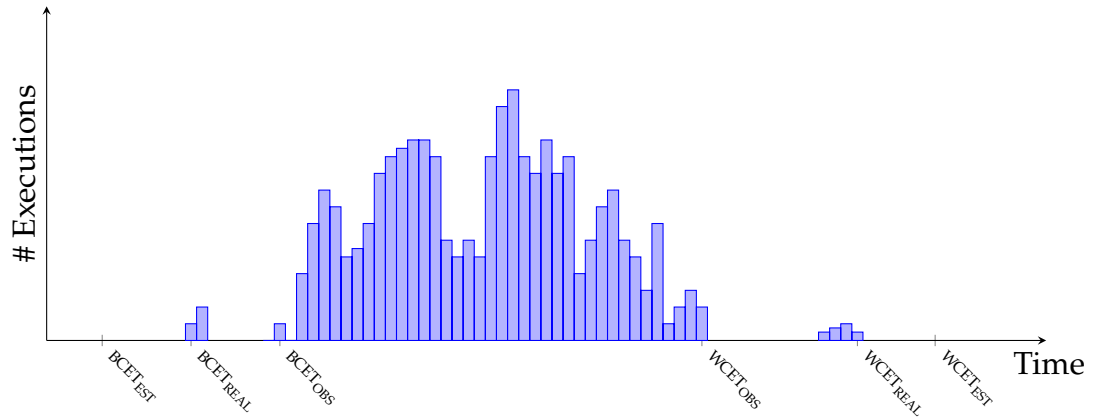


Figure 4.1: Distribution of execution times of a fictional task. The execution time varies depending on the task's input data and initial hardware state. $BCET_{EST}$ and $WCET_{EST}$ are safe bounds on the execution times. $BCET_{REAL}$ and $WCET_{REAL}$ are the actual, but unknown, best- and worst-case execution times. $BCET_{OBS}$ and $WCET_{OBS}$ are the best- and worst- case timings which are observed, e.g., in simulation runs. Figure adapted from [Wil+08].

However, a hard real-time system must always guarantee to comply to all timing constraints, even under extremely rare worst-case circumstances. WCET analysis therefore establishes the notion of the *estimated* BCET and WCET. In Fig. 4.1, these timings are denoted by $BCET_{EST}$ and $WCET_{EST}$. The estimated BCET and WCET are safe under- and over-approximations of the actual best-case and worst-case timing behavior. To avoid that the embedded system's hardware must be largely overdimensioned, it is crucial that these bounds are not only safe but also *tight*.

Although there might be cases in which a system designer is also interested in optimizing the lower bound on the minimum execution time of the task, this work focuses on optimizations and analyses of the *worst-case* timing behavior. The following sections will therefore not discuss BCET analysis in more detail but rather focus on the upper execution bounds, the WCET.

There are two fundamentally different approaches on trying to derive upper or lower bounds on a task's execution behavior: By *observing* the program behavior due to simulation or measurements, or by statically *analyzing* the program code and combining it with a safe formal model of the timing behavior of the underlying hardware [Wil+10].

When using measurement or simulation based techniques, the system designer aims at feeding the task with input data leading to the worst possible timing behavior. Obviously, the more complex a task or the underlying architecture

gets, the more difficult it is to find the sequence of input data leading to the actual $WCET_{REAL}$. Therefore, even with exhaustive testing, the observed WCET will not provide a provably *safe* bound. Instead, the observed WCET, $WCET_{OBS}$, must be considered to be lower than $WCET_{REAL}$. This is also depicted in Fig. 4.1. To counter this uncertainty, safety margins are often added to the observed WCET. The actual amount of safety margin is not scientifically based but is rather determined by the designer's experience or by company design rules or certification standards. More sophisticated measurement techniques execute smaller chunks of the task individually and try to find the input data leading to the worst-case timing behavior of that subset. These partial data can then be combined in order to retrieve some less unsafe upper bound on the WCET [Abs19]. However, despite the fact that evaluating smaller chunks may lead to a better estimation, the underlying problem of not being sure of actually triggering the global worst case of a task's runtime behavior persists.

Static WCET analysis does not base its results on simulation of given input data. Instead, the program code is formally analyzed, giving safe over-approximations on the global worst-case timing behavior. Due to the fact that this thesis tackles hard real-time systems where even one single miss of a task's deadline may lead to catastrophic behavior, the following chapters will solely elaborate on WCET analysis and will not further pursue simulation and measurement based techniques. Since the actual WCET, $WCET_{REAL}$ is unknown, any statement on *the* WCET of a task is considered to refer to the $WCET_{EST}$.

However, it should be mentioned that the optimization frameworks proposed in this thesis do not rely on any specific method to retrieve the WCETs of a task and its parts. If a system designer decides to use measurement based techniques, all proposed methods will work as good as with analysis based techniques – however, the safeness of the results can obviously not be guaranteed.

The following subsections provide a brief introduction to the huge field of static WCET analysis. Section 4.2.1 gives an overview on bounding and analyzing the possible control flows through a task. Section 4.2.2 will then elaborate on micro-architectural analysis, as well as memory access analysis – especially cache analysis. Section 4.2.3 describes the so-called Implicit Path Enumeration Technique which is commonly used in order to find the global worst-case timing behavior out of all previously gathered partial results.

4.2.1. Control Flow Analysis

One of the main challenges in WCET analysis is to find and bound the so-called Worst-Case Execution Path (WCEP) in a given task.

Definition 4.7 (Worst-Case Execution Path (WCEP))
The Worst-Case Execution Path (WCEP) is defined as that execution path through a task's CFG which leads to its WCET. If several paths lead to the WCET, all of these paths are considered equivalent with regard to the task's timing behavior. Then, any one of them may be selected as the *WCEP.*

One key issue in order to bound the WCEP is to give safe bounds on the maximum number of traversals through cyclic paths. The underlying problem is illustrated in the (artificial) example in Listing 4.1: The task to be analyzed consists of a simple function written in C. It retrieves its input from some external variable `input` and calls a recursive function `fac()` from a nested loop.

Both the loops and the recursive function calls lead to cycles in the task's CFG. Without additional knowledge (in this example, the maximum value of `input`), maximum execution bounds on neither the loops nor the recursion depth can be derived. Although, in this very simple example, a safe but very pessimistic upper value on `input` could be given (i.e., $2^{32} - 1$ in a 32 bit environment), upper bounds on the maximum traversion through a cyclic CFG cannot be given in general. Being able to give such assertions in a general program would solve the halting problem which was proven unsolvable by Turing [Tur37].

Therefore, without any additional information, cyclic structures in the CFG must be assumed to be repeated an infinite number of times in a general program, thus leading to an infinite WCET. To tackle this problem, system designers can choose between two different approaches:

The first approach is to avoid any control structures which lead to cyclic CFGs which are not trivially bounded. Some research groups even propose so-called "single-path" code where the control flow through a task does not vary at all [Pus03]. When using single-path code, the problem of bounding the WCEP is easily solved. However, despite being easy to analyze, single-path code by definition performs slow, as it always executes all instructions, even if a computation's result is discarded right afterwards.

The second – and less restrictive – approach is to only use cyclic code constructs where a safe upper bound on the maximum number of iterations can be given. Although not formally required, this often implies the avoidance of complex algorithmic structures like recursions or **`goto`** expressions.

Simple loop structures can often be bounded automatically as shown by, e.g., Lokuciejewski et al. [Lok+09a] and Gustafsson et al. [Gus+03]. When more general control flow structures are needed, the problem of bounding the WCEP can be solved by so-called *flow facts*. These are annotations on source or machine code-level which are given by the programmer. Using these annotations, the execution count of cyclic control flow structures is bounded manually. In Listing 4.1, this

Listing 4.1: Exemplary task, loosely based on the "fac" benchmark from [Gus+10]. The listing illustrates the use of flow facts [FL10] to bound maximum and minimum execution counts on loops and recursions.

```
1  int fac( unsigned int n ) {
2    if( n == 0 )
3      return 1;
4    else
5      return ( n * fac( n-1 ) );
6  }
7
8  // Maximum allowed value is 100
9  extern unsigned int input;
10
11 unsigned int task() {
12
13   unsigned int result = 0;
14   _Pragma("loopbound min 0 max 100")
15   for( unsigned int i=0; i != input; ++i ) {
16     _Pragma("loopbound min 1 max 100")
17     for( unsigned int j=i; j != input; ++j ) {
18       _Pragma( "marker recursivecall" )
19       result += fac( i+j );
20       _Pragma( "flowrestriction 1*fac <= \
21                 198*recursivecall" )
22     }
23   }
24   return result;
25 }
```

has been done by `_Pragma` statements. This was proposed by Falk et al. [FL10] [LM11, p. 19] and is implemented into the `WCC` compiler framework which is used as evaluation platform throughout this thesis. The syntax of `_Pragma` statements is quite straightforward. Loops are simply annotated by minimum and maximum iterations each. For the recursion, a special `marker` pragma is set and accompanied by a `flowrestriction` pragma. This limits the maximum calls to `fac()` to 198 times of each original call within `task()`'s innermost **`for`** loop. Although this provides a safe over-approximation, the approximation is obviously not tight for the given example. The inner loop is only executed once with the maximum loop bound of 100. The next time, the loop is executed only 99 times, down to only 1 single iteration for `i==99`. Accordingly, `fac()`'s recursion depth is much

lower than 198 in the uttermost cases. Further flow restrictions can be used to tighten the approximation as elaborated by Falk et al. [FL10]. Any WCET analysis will then assume that these annotations are correct and use these bounds in the upcoming steps of the WCET analysis.

4.2.2. Micro-Architectural Analysis

The previous section showed how the paths through a task can be bounded on the source code-level. In order to be able to give a safe bound on the actual timing through these paths, a detailed analysis has to be performed on a machine instruction-level, with respect to the concrete hardware on which the task will be executed.

The reasons for this are twofold: First, no timing guarantees can be given before it is known how a statement in a task's source code is actually expressed on the level of machine instructions. Depending on the complexity (e.g., multiplication of two double-precision floating point values), one statement on the source code-level may easily lead to multiple statements and even function calls on machine code-level. Second, even with the knowledge of the concrete assembly instructions being executed, the actual timing behavior is highly dependent on the target architecture. This includes not only properties like the target processor's clock frequency and memory access times but also its internal pipeline structure and many more so-called micro-architectural properties.

Unless explicitly stated otherwise, the upcoming sections subsequently assume that the previously analyzed source code has been translated into a final binary file by a compiler. Then, the analyses operate on the level of machine instructions with the mandatory requirement of exactly knowing the target architecture on which the task is going to be executed.

Section 4.2.2.1 will first introduce the concept of the value analysis as part of the WCET analysis. Section 4.2.2.2 will then provide an introduction into cache analysis. Subsequently, Section 4.2.2.3 covers the analysis of the microprocessor's pipelining behavior. Finally, Section 4.2.2.4 describes call contexts and how they are handled.

4.2.2.1. Value Analysis

The so-called *value analysis* is used to try to determine the contents of each register at each position in the program. If any safe guarantees about the contents of a register can be made, this information can subsequently be used in order to predict the behavior of the machine instructions using these registers. E.g., depending on the target architecture, a floating point multiplication by `0` might be much faster

than a multiplication of two arbitrary real numbers. More importantly the results of the value analysis can be used as the basis to automatically bound the maximum execution counts of loops, as mentioned in the previous section. Additionally, value analysis is needed in order to predict which actual data element is being accessed when using pointer operations (e.g., "`int a = *b;`" on a C code-level) [LM11, p.21f.].

Value analysis can be performed for both the machine instructions or the task's source code. When operating on machine instruction-level, recognizing variable definitions and usages and following the value propagation through the CFG is much more difficult than on a source code-level. However, when operating on source code-level, there must be some kind of compiler support in order to annotate which assignment operations and expressions in the source code will lead to which registers on an instruction-level. Otherwise, the WCET analysis cannot link the findings from the value analysis to the concrete machine instructions whose timing behavior is to be predicted.

Value analysis also plays an important role in compiler optimizations. Constant values can be propagated throughout the program's code, enabling the compiler to simplify statements and thus improve on the program's runtime, whenever assignments and calculations can safely be performed at compile time [Aho+07, p. 632ff.]. Value analysis is not needed explicitly for this thesis but only being used as part of the WCET analysis which is needed as a basis. It will therefore not be discussed any further in detail.

4.2.2.2. Cache Analysis

Section 3.2.1 introduced caches as a way to improve the average-case access times to slow memories. When using caches, access times to the cached memory vary significantly, depending on whether the access leads to a cache hit or miss.

Whether a hit or miss occurs depends on both the initial cache state and any previously accessed memory regions. For simple architectures, assuming each memory access as a cache miss will lead to a safe overestimation of the timing behavior. However, Lundqvist et al. showed that for complex microprocessors, timing anomalies may occur [LS99] when instructions may be executed out of order. In this case, the worst-case timing is actually triggered if a memory access results in a cache hit. Although Lundqvist et al. claim that timing anomalies may only occur for microprocessors featuring out of order execution, Gebhard [Geb10] proved otherwise by demonstrating a timing anomaly for the LEON2 microprocessor. Therefore, a safe analysis on both potential cache hits and potential cache misses is essential for a safe WCET analysis. As result, cache analysis builds its own stage within the WCET analysis of a task.

Cache analysis can be divided into two classes: *Intra* and *inter* task cache analysis. Intra task cache analysis has been researched intensively over the last decades, e.g., by Li et al. [LMW96] and Ferdinand and Wilhelm [FW99]. It classifies cache accesses within one task, assuming the task is executed in a stand-alone manner. Inter task analysis, on the other hand, analyzes the impact on the cache if a task is preempted by another task. In this case, the preempting task may evict parts of the preempted task from the cache, thus leading to timing penalties which were not foreseen by the intra task analysis. Inter task cache analysis is tackled in Section 4.5.3. This section will focus on intra task cache analysis. Thus, it is assumed that the analyzed task is run in isolation and suffers from no outside disruptions.

Intra task cache analysis is split in two parts: A *must* and a *may* analysis. The *must* analysis analyzes which instructions (or data) *must* reside in the cache at any given point of execution and thus lead to definitive cache hits. The *may* analysis, on the other hand, is used to classify whether a memory access *may* result in a cache hit. Subsequently, a cache miss occurs, if a memory block might not be in cache. The combination of both must and may analysis can then be used to give safe upper and lower bounds on the memory access times of any data and instruction access at any point of a task's execution. Due to the fact that the exact control flow through the program is only determined at runtime, both must and may analyses cannot provide exact results but only safe approximations.

For obvious reasons, any concrete analysis of a system's caching behavior depends on the cache properties like its size, associativity, . . . and its replacement policy, e.g., LRU (cf. Section 3.2.1). If the target architecture can be chosen freely, the system designer might refer to, e.g., [Rei08] where different replacement policies are compared with regard to their applicability in hard real-time systems.

As this work focuses on compiler optimizations for commonly found architectures, subsequent chapters will focus on the LRU replacement policy. The Infineon TriCore architecture which is used as one of the evaluation targets throughout the upcoming sections, is also relying on LRU as cache replacement policy.

4.2.2.3. Pipeline Analysis

As part of the micro-architectural analysis, the pipeline analysis focuses on the microprocessor's pipeline. Even rather old architectures like ARM7TDMI do not process each machine instruction separately. Instead, due to the pipelining concept, different processing stages operate on different instructions in parallel. E.g., while the result of an `add` instruction is calculated in the microcontroller's arithmetic pipeline, the previous instruction's result may be written back in the *writeback* phase, and the next instruction's operational code may be retrieved from memory.

As a result, although passing one individual instruction through the CPU might take several cycles, the *average* time an instruction needs can be drastically lower.

This pipelining behavior may be disturbed when executing instructions which may take a varying time in the execution stage, conditional jump instructions are executed where the forthcoming control flow is yet unknown, or subsequently executed machine instructions depend on each other. In these cases, the pipeline either *stalls*, i.e., it does not pass through instructions to the next stage until further proceeding is known. Or, alternatively, it may speculatively proceed fetching instructions. If this speculatively fetch turns out to be wrong, the pipeline must be flushed and the correct instructions must be fetched. If the architecture even allows for speculative *execution*, then the result of these speculatively performed operations must be undone to ensure a functionally correct working program.

WCET-oriented pipeline analysis has been intensively researched by, e.g., Engblom [Eng02]. Boiled down, its goal is to find safe but tight estimates on the execution time of instructions which are definitely executed in sequence. At points where the control flow splits and jumps may occur, it provides safe upper bounds on the additional timing penalties inflicted by a possible misprediction. Pipeline analysis is highly target-specific. Performing a tight but safe pipeline analysis requires detailed knowledge on the internal structure of the target processing unit.

4.2.2.4. Execution Contexts

The previous section explained the necessity of the pipeline analysis which returns the time needed to execute a sequence of instructions. However, in many cases, there is not *the* execution time of a piece of code.

Consider, e.g., a loop which is passed several times. In its first execution, the microprocessor's branch predictor may have mispredicted the loop to not be entered at all, thus resulting in an additional timing penalty. Additionally, the target platform may use instruction and data caches, and the loop's contents may not be in the cache at its first iteration. As a result, instruction fetches and data loads are slow in the first iteration, but fast in subsequent passes. These different access times obviously lead to massive changes in the pipeline analysis.

A safe approximation would be to use the worst possible case (in this example, the loop's first execution) and multiply this timing estimate by the number of loop iterations. This will obviously lead to a major over-approximation of the actual worst-case timing behavior. Modern analysis tools are therefore able to maintain several so-called *execution contexts*. In the case of the loop, each loop iteration may have its own context, with its own pipeline analysis. To seamlessly integrate contexts into the analysis frameworks, analyses typically model contexts by *virtually unrolling* loops or *virtually cloning* functions.

E.g., consider a loop with 10 iterations. The first loop iteration might take 20 time units, and the following 9 iterations may finish within 15 time units each. Without contexts, the loop costs would have to be accounted by $10 \cdot 20 = 200$ time units. With contexts, the analysis will virtually unroll the loop. This means that it is modeled as one single-execution block which needs 20 time units, and a loop block which has 15 time units for each iteration but is only traversed 9 times As a result, the loop's WCET estimate will result in $20 + 9 \cdot 15 = 155$ time units. The unrolling is called "virtual", because it is only performed for the analysis. The actual machine code and the task's execution behavior are *not* changed. Functions which are called multiple times with different (but known) parameter values can be handled accordingly by virtually cloning them. Then each call of the function can be analyzed separately with the respective parameter values. While this allows for a much tighter WCET estimate, it also significantly increases both analysis time and memory consumption.

Therefore, analyzers often compromise by keeping the worst N calling contexts for code constructs, where N is a user-definable number. Therefore, the user may choose, whether a faster but more pessimistic WCET analysis is acceptable or whether to perform a more sophisticated analysis which may take notably more computational resources.

4.2.3. Implicit Path Enumeration Technique

Finally, after analyzing the separate chunks of instructions of the task, the overall WCET must be calculated. This is usually done by using the so-called Implicit Path Enumeration Technique (IPET) proposed by Li and Malik [LM95]. The IPET models the control flow of a task by using flow constraints, similar to Kirchhoff's circuit laws in physics. These constraints are expressed as integer linear equations. An objective function is then defined which aims at maximizing the flow through the task's CFG, while adhering to all constraints. The resulting ILP can then be solved using any ILP solver, finally getting a safe estimate on the WCET of that task. To achieve this, IPET operates on the Control Flow-Graph (CFG) of a task at a basic block-level as defined in Definition 4.5 and Definition 4.6.

Fig. 4.2 shows the CFG of a minimal task consisting of the two functions `main()` and `func()`. The basic blocks are denoted by capital letters, while the edges are represented by a lower-case e with a numerical index. We define e_{main} to denote the flow entering function `main()` and e_{func} to denote the flow entering function `func()`. `func()` can now be expressed by one simple flow constraint with the flows being expressed as non-negative integer variables:

$$e_{\text{func}} = e_7 + e_8 \tag{4.1}$$

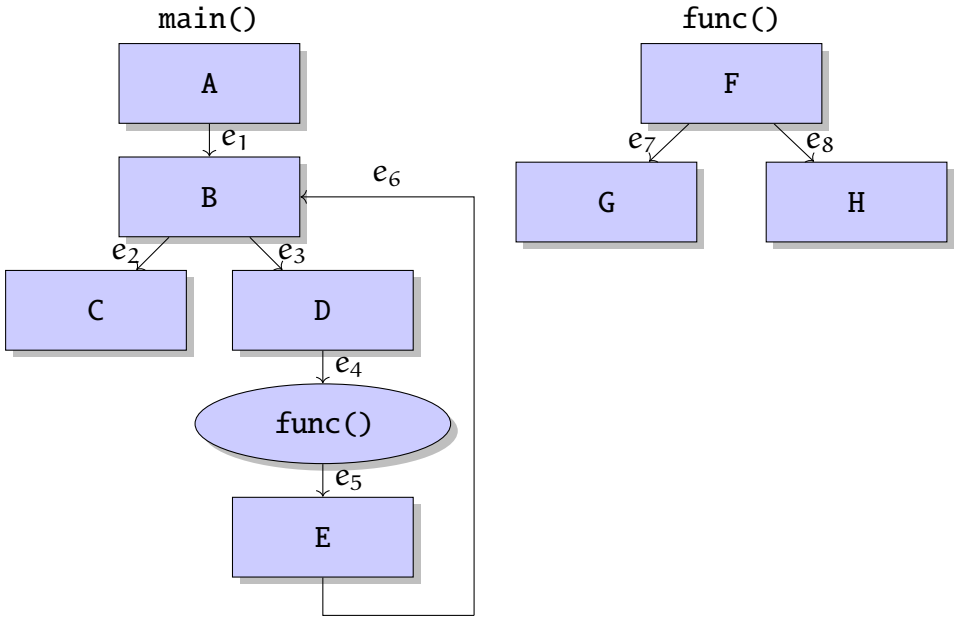


Figure 4.2: CFG of an Exemplary Task.

In other words: The flow going into basic block F must equal the sum of the flows exiting the block. Therefore, for one execution of e_{func}, either basic block G or H but not both must be taken.

Modeling `main()` is a little bit more complex due to the loop. However, the IPET can be built in exactly the same way by simply modeling the sums of ingoing and outgoing edges for each node:

$$e_1 + e_6 = e_2 + e_3 \tag{4.2}$$

$$e_3 = e_4 = e_{\text{func}} = e_5 = e_6 \tag{4.3}$$

Note that this does not yet account for any loop bounds. To prevent an infinite flow, loop iterations must be bounded by user annotations (or automatically by the WCET analyzer framework), as discussed in Section 4.2.1. For this example, assume that the loop defined by the back-edge from basic block E to basic block B is executed a maximum of 10 times. Therefore, the user would have to define

$$e_6 = 10 \tag{4.4}$$

to limit the number of times the back-edge may be taken. Additionally, the number of times the `main()` function is entered must be defined:

$$e_{\text{main}} = 1 \tag{4.5}$$

Given these limits, the total flow into basic block B is set to $1 + 10 = 11$. Due to Eq. (4.3), the flow also restricts the call frequency to `func()` to 10 and e_3 to 10. Thus, the flow e_2 into basic block C is enforced to 1 subsequently modeling the fact that the task must be exited via C.

The presented constraints model the control flow, but they do not yet lead to a WCET estimate. For this, the net worst-case timing behavior of each basic block is introduced as C_x. The index x denotes the corresponding basic block. For example, C_A represents the worst-case execution time for one single execution of basic block A. These timings can be retrieved by micro-architectural analysis as presented in Section 4.2.2.

To simplify mathematical notations, the sum of all flows of all edges going in each basic block is expressed by another ILP integer variable f_x with x representing a basic block. E.g., f_a models the sum of all flows entering basic block A.

$$f_A = e_{\text{main}} \tag{4.6}$$
$$f_B = e_1 + e_6 \tag{4.7}$$
$$f_C = e_2 \tag{4.8}$$
$$f_D = e_3 \tag{4.9}$$
$$f_E = e_5 \tag{4.10}$$
$$f_F = e_{\text{func}} \tag{4.11}$$
$$f_G = e_7 \tag{4.12}$$
$$f_H = e_8 \tag{4.13}$$

The worst-case timing behavior of the complete program is now obtained by maximizing the sum of worst-case flows into each basic block multiplied by their respective worst-case timing:

$$\max \left(\sum_{\forall i, i=A,B,\ldots,H} f_i \cdot C_i \right) \tag{4.14}$$

Additional flow constraints can be used to model recursive calls, infeasible paths and irregular loops. This is not discussed in more detail, as it is not necessary for understanding of the upcoming thesis. A more formal introduction into modeling the IPET is given in the original paper [LM95].

4.3. Task Models

An embedded real-time system may feature multiple tasks which are executed repeatedly. To be able to give any safe estimates on the system's overall behavior,

their maximum execution rates must be bounded. They may either be predefined by one constant period T, or arbitrarily complex execution patterns.

This work uses two different task models. For strictly periodical systems, a simple task model based on the works of Liu and Layland [LL73] is used. For distributed embedded systems with arbitrary activation patterns, event-based task models were introduced by Richter [Ric05], Wandeler [Wan06] and later on modified by Kollmann [Kol+10]. Due to the fact that this thesis focuses on single processing units instead of distributed systems, some simplifications to the model could be made. Especially, best-case execution timing behavior and bounds on the minimum execution rates can be neglected. When necessary, the appropriate proofs for simplifications are given. Additionally, it is assumed that tasks are not dependent on each other. If such dependencies are present, the system designer has to model these by means of clustering two dependent tasks or by assigning task priorities accordingly.

4.3.1. Priorities

In any multi-tasking system, one task, the so-called *scheduler*, acts as a kind of supervisor that decides which task is to be executed next. In real-time systems, this decision is made by evaluating each task's priority. Depending on the scheduling algorithm which was defined by the system designer, priorities may either be fixed at design time, or calculated by the scheduler dynamically at runtime.

In case of dynamic priority assignment, it is neither needed nor possible to express tasks' priorities within the task model. For fixed-priority systems, however, the priority of a task is tied to the task and therefore part of the definition of a task. Although in theory any kind of identifiers may be used to describe priorities, non-negative numeric identifiers became common practice.

Definition 4.8 (Priority of a Task)
A priority P_i is a non-negative integer number associated with a task τ_i. We define that a numerically lower number denotes a logical higher priority. As a result, 0 is the highest priority available in a system.

Throughout the upcoming work, each task τ is assigned a unique identifier i which is denoted as an index variable. Therefore, task i is written as τ_i. Accordingly, τ_i's priority is defined by P_i.

In order to simplify notations, it is common to re-use a task's index i as its priority in case of fixed-priority scheduling. As a result, it can be directly seen that, e.g., τ_1 has a higher priority than τ_2, and that τ_0 is the task with the highest priority in the system, without having to state each task's priority explicitly.

On the downside, this notation does not allow for multiple tasks having the same priority. If multiple tasks with identical priority should be needed in a concrete scenario, formulations may be rewritten using the exhaustive notation with an individual identifier P_i for each task τ_i.

In case of dynamic priority assignment, the index i is solely used to identify and distinguish the tasks without any further meaning.

Example 4.1 (Priority Assignment)
Consider a system Γ consisting of two tasks. Then, these tasks will be assigned the indices 0 and 1:

$$\Gamma = \{\tau_0, \tau_1\} \tag{4.15}$$

In a fixed-priority system, this means that the priority of task τ_0 is 0 and the priority of τ_1 equals 1. Due to the fact that, by definition, lower numeric values are defined as a higher logical priority, a real-time scheduler will choose to execute τ_0 over τ_1.

4.3.2. Periodic Model

If all tasks in a system are triggered for execution strictly periodically, a relatively simple model can be used to describe each task. In this case, each task can be described by its execution frequency and deadline only.

Depending on the scheduling algorithm, a priority indicator may be defined additionally. However, the priority can often be directly derived from the execution frequency or deadline and is not given explicitly. As a result, a periodic task can be defined as follows:

Definition 4.9 (Periodic Task)
A periodic task τ_i is defined as a tuple (C_i, D_i, T_i). C_i denotes the WCET of τ_i and D_i the deadline. T_i is the activation period of the task. If the system is scheduled using fixed priorities, the task's index i is used to denote the task's priority with 0 being the highest priority.

As a special case, the description can be even more simplified if a task's deadline equals the period. In this case, the deadline is called to be *implicit*. If all tasks' deadlines equal their respective periods, the system is called an *implicit-deadline system*.

Definition 4.10 (Implicit Deadline Tasks and System)
A periodic task τ_i with $D_i \equiv T_i$ is called an implicit-deadline task. *If all tasks τ_i in a given task set Γ are implicit-deadline tasks, the system is called* implicit-deadline system.

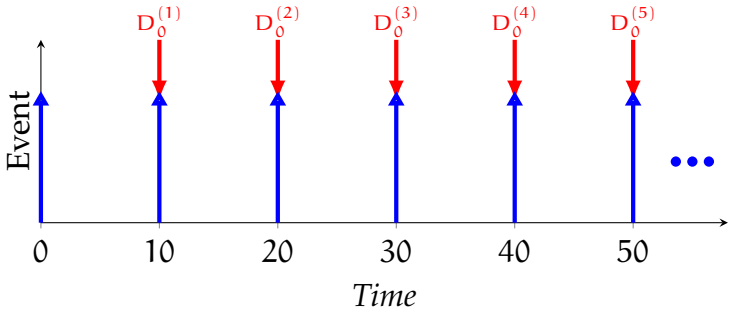


Figure 4.3: Example of a strictly periodically triggered task τ_0 with period $T_0 = 10$ time units and implicit deadline. The blue arrows pointing upwards signify the activation of the task while the red arrows pointing downwards signify the respective deadlines.

Example 4.2 (Strictly Periodical Task)
Fig. 4.3 shows the activation pattern of a strictly periodical task τ_0 with implicit deadlines. This means that one instance of a task must finish at the latest right before the next instance of the task is being triggered. The task has a period of $T_0 = 10$ arbitrary time units. In the figure, this is denoted by the red arrows pointing downwards, which are annotated by D_0^K with K being the corresponding instance of τ_0.

4.3.3. Event-Triggered Modeling

By definition, a periodic task is considered to be triggered for execution in strictly equidistant time intervals. If a task, in fact, is not executed in strictly equidistant time intervals, the periodic task model is not powerful enough to describe the exact task activation pattern. Instead, the shortest possible distance between two subsequent task activations has to be taken and assumed as the period of the task. This will still lead to a safe model with regard to the worst-case timing behavior but introduce a significant amount of pessimism. This is illustrated by Example 4.3.

Example 4.3 (Periodical Task with Recurring Burst)
Fig. 4.4 shows the activation pattern of another task τ_1. There are no deadlines denoted, as it is not important for this example. The task is obviously not triggered with one fixed period. Instead, after 3 instances of the task with a distance of 10 time units, 2 instances follow with only 5 time units distance between each task activation. Then, the pattern repeats with a distance of 10.

With the periodic task model, this cannot be expressed exactly. Instead, the minimal distance between two subsequent task activations has to be taken as the task's period in order to be able to provide a safe *estimation on the task's activation pattern. I.e., $T_1 = 5$.*

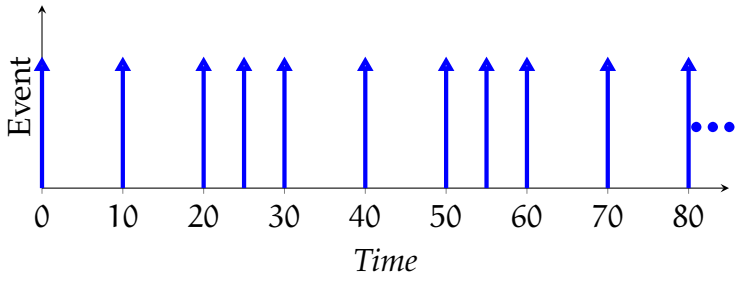


Figure 4.4: Example of an aperiodically triggered task τ_1. There are always 3 instances of the task with a distance of 10 time units and 3 jobs with a distance of 5 time units. This pattern is repeated infinitely.

This obviously leads to significant pessimism in a schedulability analysis, as the task is actually not being executed that often.

To counter these shortcomings, Gresser introduced an event-triggered task model [Gre93a]. In such a system, a task's execution is triggered by *events* which can occur at arbitrary rates. In order to mathematically describe these activation patterns, an event density function $\eta(\Delta t)$ is introduced. The density function does not operate on absolute time stamps but on time *intervals*. It returns the maximum number of events which trigger a given task within a window of Δt time units. More formally, the density function may be defined as:

Definition 4.11 (Density Function)
The density function $\eta(\Delta t)$ *denotes the maximum number of events in any interval of length* Δt *[Gre93a].*

In order to associate a given event function with a given task τ_i, the event function is annotated using the index i: $\eta_i(\Delta t)$. Note, that in this thesis when talking of a *time interval* Δt, this refers to an interval with the *length* of Δt.

The *interval function* is the inverse of the density function. It returns the minimum distance in which a given number of events may occur:

Definition 4.12 (Interval Function)
The interval function $\delta(n)$ *describes the minimal distance in which* n *events are generated [Gre93a].*

Analogously to the density function, the interval function for a given task τ_i can be denoted by $\delta_i(n)$. The minimum distance between two subsequent events, i.e., $\delta(2)$ is called the *inter-arrival time*. Subsequently, for a strictly periodically triggered

task, the task's inter-arrival time equals its period. Using these definitions, an event-triggered task τ_i can be defined as follows.

Definition 4.13 (Event-Triggered Task)
An event-triggered task τ_i is defined as a tuple $(c_i, D_i, P_i, \eta_i(\Delta t))$.

Identically to Definition 4.9, c_i is the WCET of the task, D_i its deadline. P_i signifies the task's priority in case of fixed-priority scheduling. As described above, we define $P_i \equiv i$ in order to achieve nicer mathematical notations in the upcoming chapters. However, the density function $\eta_i(\Delta t)$ is used instead of the period T_i in order to describe the task's activation pattern. The interval function $\delta_i(n)$ is not included in the task's definition, as it may be derived from the event density function.

Example 4.4 (Strictly Periodic Task Using Event Model)
Example 4.2 modeled a strictly periodic task τ_0 with a period of $T_0 = 10$ time units and implicit deadlines. When transforming this into the interval domain, the density and interval functions η_0 and δ_0 are:

$$\eta_0(\Delta t) = \left\lceil \frac{\Delta t}{10} \right\rceil \tag{4.16}$$

$$\delta_0(n) = \max((n-1) \cdot 10, 0) \tag{4.17}$$

The rationale behind these functions is straightforward: One new event is triggered every 10 time units. Therefore, in an interval of 10 time units, at most 1 event may occur. In an interval of 20 time units, at most 2 events may occur, ... The interval function – forming the inverse of the density function – can be deduced accordingly. A maximum of 0 events is triggered in a time interval of exactly 0. A negative number of events does not exist, and the interval function is defined to 0 for such input. For any interval length which is an infinitely small amount larger than 0, at least one event may happen. However, the minimum interval between 2 subsequent events is at least $(2-1) \cdot 10$, as this is defined by the task's period. Figs. 4.5 and 4.6 show the plots of the resulting event curves.

Example 4.5 (Periodic Task with Recurring Burst using Event Model)
Example 4.3 introduced task τ_1 which is not triggered by one fixed period. The activation pattern was shown in Fig. 4.4.

In order to derive the density function $\eta_1(\Delta t)$, the minimum distances between subsequent task activations (at an arbitrary point in absolute time) have to be analyzed. In this example, the minimum number of tasks in a time interval of up to 5 time units is 1. In any time interval which is slightly larger but still lower than 10 time units, the maximum number of task activations is 2. This can be deduced graphically from Fig. 4.4

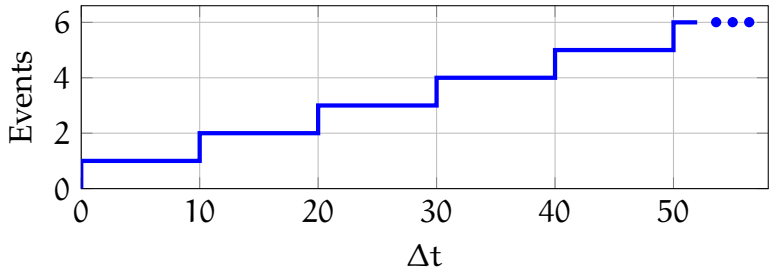


Figure 4.5: Example density function $\eta_0(\Delta t)$ of a strictly periodically triggered task τ_0 with period $T_0 = 10$ time units.

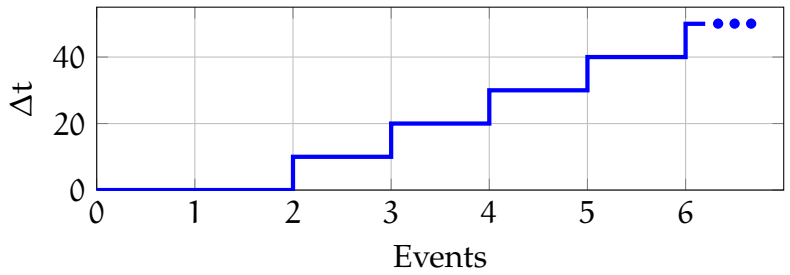


Figure 4.6: Example interval function $\delta_0(n)$ of a strictly periodically triggered task τ_0 with period $T_0 = 10$ time units.

by sliding a frame slightly smaller than 10 time units over the occuring events. It is not possible to capture more than 2 events within the window.

Accordingly, for a time interval of 10 time units, 3 instances of τ_1 can be triggered (E.g., between 20 and 30 in absolute time in Fig. 4.4). However, a time interval of at least 20 is needed in order to capture 4 events. Thus, $\eta_{20} = 4$. The resulting density function is depicted in Fig. 4.7.

The interval function $\delta_1(n)$ can be derived accordingly: The minimum interval between 2 subsequent events at any position in the time stream occurs at a burst and is 5 time units. Subsequently, $\delta_1(3) = 10$. Because the burst is limited to 3 subsequent events (by definition of the example), 4 events occur in a minimal interval of 20. Therefore, $\delta_1(4) = 20$. Fig. 4.8 shows the graph of the resulting interval function.

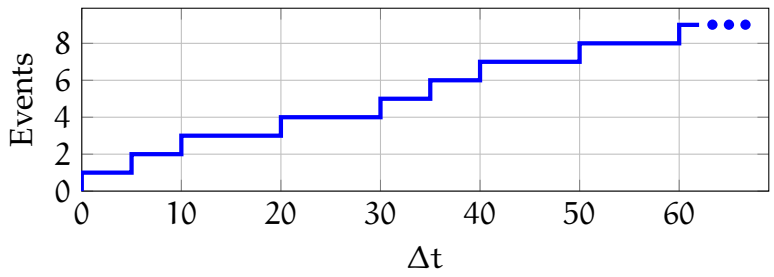


Figure 4.7: Example density function $\eta_1(\Delta t)$ of a periodically triggered task τ_1 with recurring bursts.

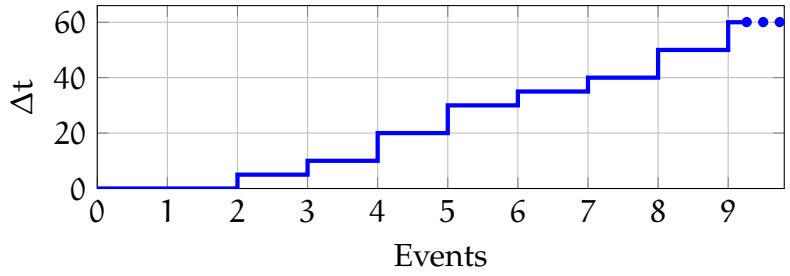


Figure 4.8: Example interval function $\delta_1(n)$ of a periodically triggered task τ_1 with recurring bursts.

The density function can easily be given as an explicit formulation by adding each periodically recurring "step" in the density function:

$$\begin{aligned}\eta_1(\Delta t) = &\left\lceil \frac{\Delta t}{30} \right\rceil + \\ &\left\lceil \frac{\max(\Delta t - 5, 0)}{30} \right\rceil + \\ &\left\lceil \frac{\max(\Delta t - 10, 0)}{30} \right\rceil + \\ &\left\lceil \frac{\max(\Delta t - 20, 0)}{30} \right\rceil\end{aligned} \tag{4.18}$$

The interval function is calculated accordingly:

$$\begin{aligned}\delta_1(n) = 5 \cdot \left\lceil \frac{\max(n-2,0)}{4} \right\rceil + \\ 5 \cdot \left\lceil \frac{\max(n-3,0)}{4} \right\rceil + \\ 10 \cdot \left\lceil \frac{\max(n-4,0)}{4} \right\rceil + \\ 10 \cdot \left\lceil \frac{\max(n-5,0)}{4} \right\rceil\end{aligned} \tag{4.19}$$

These two functions precisely model the behavior of the task's activation pattern.

One key property of the density function is that it operates on time *intervals* and no longer on wall-clock time. This inherently leads to two important properties: First, the density function is monotonically increasing and second, it is subadditive (i.e., larger time intervals must not outweigh smaller ones):

Proposition 4.1 (Density Function)
Any density function must meet the following two properties:

- *The density function is monotonically increasing:*

$$\forall \Delta t_1 \leqslant \Delta t_2 : \eta(\Delta t_1) \leqslant \eta(\Delta t_2) \tag{4.20}$$

- *The density function is subadditive: For a time interval Δt_2, with $\Delta t_2 \geqslant \Delta t_1$ and $\Delta t_3 = \Delta t_2 - \Delta t_1$, it holds that:*

$$\eta(\Delta t_1) \geqslant \eta(\Delta t_2) - \eta(\Delta t_3) \tag{4.21}$$

Proof. We prove both statements by deducing the property from Definition 4.11.

The first property can be deduced directly from the definition. If for a given time interval Δt_2 the maximum number of events is N, it is not possible to have more events in a smaller interval Δt_1. Otherwise, these additional events which occur in the small interval would by definition also have to be present in the larger interval.

The second property is proven by contradiction. Eq. (4.21) can be rewritten as:

$$\eta(\Delta t_1) + \eta(\Delta t_3) \geqslant \eta(\Delta t_2) \tag{4.22}$$

Assume the statement does not hold, i.e.:

$$\eta(\Delta t_1) + \eta(\Delta t_3) \overset{!}{<} \eta(\Delta t_2) \tag{4.23}$$

Consider an arbitrary task with an arbitrary density function. For a worst-case scenario, it can be assumed that at any point in time, $\eta(\Delta t_1)$ events occur within Δt_1 and then exactly right after in an interval of Δt_3, $\eta(\Delta t_3)$ events occur. Due to the strict "lower than" formulation in Eq. (4.23), this would imply that when looking at the complete time interval Δt_2 as a whole, more events might occur in total than when looking at the very same time interval Δt_2 but counting the number of events in two sub-intervals and adding the partial results. This is obviously logically infeasible, thus Eq. (4.21) holds. □

Due to the generality of this approach, density and interval functions can be used to describe any arbitrary task activation pattern. In practice, this may happen by any means, ranging from simple tables to arbitrary symbolic formulations. For some models like periodically occuring bursts, the event and interval functions can easily be given directly as mathematical formulas, so-called event functions. However, this is not a requirement on using event-based task modeling. Density and interval functions may be given by a graphic representation, a lookup table or some other means.

4.4. Scheduling Algorithms

If several tasks are executed on one processing unit, some mechanism must be established in order to decide which task is to be executed. There are two fundamental concepts: Off-line and on-line scheduling [Gaj+09, p. 161]. In off-line scheduling, the system's execution pattern is pre-calculated at system design time and then statically executed at runtime. This makes the system easy to analyze and verify, yet drastically restricts the flexiblity of the system at runtime. The system cannot quickly react to, e.g., a high priority event like an interrupt or retreived sensor data. Instead, all – already scheduled – tasks are always executed in the predetermined manner - no matter what the current state of the system is. Therefore, off-line scheduling will not be considered within this thesis.

In on-line scheduling, the decision which task is to be executed at a given point in time is made at runtime. Again, there are two different approaches: Cooperative scheduling and scheduler based execution. In cooperative scheduling, a running task decides by itself if and when execution is handed over to another task. In hard real-time systems, this again may drastically increase latencies before execution is actually handed over to a task with a high priority. This can significantly decrease the schedulability of the overall system [DB11]. This thesis will therefore not cover cooperative scheduling schemes.

As an alternative, a special task, the so-called *scheduler*, can be introduced into the system. The scheduler can be triggered by a hardware interrupt and be

executed in an arbitrary fashion. On execution, the scheduler has to decide if the currently running task may presume operation or whether it is interrupted by another task. The complexity of this scheduler task can vary significantly depending on the system's purpose.

On general-purpose systems, the scheduler is a small part of the enclosing operating system, and its decision on which task is assigned computing resources can be based on various different parameters. On the one hand, the user can often select certain preferences, as, e.g., by using the well-known `nice` levels under Unix based operating systems. Apart from this, the scheduler usually tries to split the available computing resources as fair as possible between all tasks to provide the feeling of a seamless operation to the end user. However, there are no hard criteria or constraints enforcing the execution of one special task over another.

In contrast, as defined in Definition 3.2, a hard real-time system features fixed timing constraints which must be adhered. As a result, the scheduler's sole purpose is to execute tasks in an order that ensures compliance with these timing constraints. User experience or any other scheduling strategies based on heuristics are not applied as they would prevent the software designer from being able to analyze whether the system will provably hold all constraints. However, there is not *the* perfect scheduling strategy. Although theoretically optimal schedulers have been designed, the computational overhead of the implementation of the scheduler itself is usually neglected. As a result, a complex scheduler in a real-existing system may be able to optimally determine which task to execute next, yet lead to an unschedulable system due to the sheer overhead of the computational costs of this calculation itself. On the other hand, a sub-optimal but computationally cheap scheduling algorithm may lead to a provably schedulable system.

The question which scheduling algorithm to use, is not only decided based on scientific reasons. In practice, this may also be determined by legacy code, hardware platform restrictions or safety regulations. This work is therefore not prefering one scheduling algorithm over another. Instead, the system designer may freely decide on an algorithm (and its implementation) which fits his needs. Still, when optimizing multi-tasking systems with a focus on their overall schedulability, optimization strategies must be aware of the underlying scheduler in order to achieve best results. The following subsections will therefore give a brief overview of the most common scheduling algorithms in the domain of embedded hard real-time systems. Scheduling analysis techniques of the presented algorithms will then be discussed in Section 4.5.

4.4.1. Preemptive and Non-Preemptive Systems

The first differentiation between schedulers is to distinguish between *preemptive* and *non-preemptive* systems.

Definition 4.14 (Preemptive System)
A system is called preemptive, *if a running task can be interrupted and suspended by another task. [LL73]*

One form of preemption is the handling of hardware-triggered interrupts in a system where an external signal triggers the execution of a so-called Interrupt Service Routine (ISR). These are usually small functions written in regular C code. A typical use case of an ISR is the notification of an important incoming event or a notification about the pass of a given timing interval by a hardware clock.

In a preemptive system, the scheduling task itself could be executed by such an ISR which is triggered by a hardware clock every couple of milliseconds. Then, the scheduling task may decide to return without any changes, or to execute a different task instead of the currently running one. In a non-preemptive system, a task which currently runs is guaranteed to be executed without interruption by any other task until it finishes. Only after the task has finished, the scheduler is re-executed and may decide which task should be executed next.

The benefits of a preemptive system are the possibility to react much faster if an important task wants to be executed. In contrast, the overhead due to the scheduler being activated is significantly higher in a preemptive system, as it is often triggered in a pre-defined periodic fashion, whether any other task is waiting for execution or not. Purely non-preemptive systems are usually found in very small-scale hard real-time systems with only very few tasks. The more tasks exist in a system and the more complex the activation patterns of these tasks can be, the more likely it is to use a preemptive scheduler.

Often, systems are neither fully preemptive nor fully non-preemptive but only allow preemptions upon specific times. In the simplest case, this stems from the fact that the scheduler is called in a time-triggered, periodical fashion independently of the arrival of any task-triggering events. This way, a new task has to wait until the scheduler is re-executed before being able to run. In more complex scenarios, system designers define critical sections within their programs in which a task cannot be interrupted, thus allowing preemption only at given points in the control flow of the currently running task. This can help reducing the WCET of a task but can also drastically increase complexity of system analysis.

In both preemptive and non-preemptive systems, a priority is assigned to each task. This priority is the key indicator for the scheduler to decide which task

should be executed next. The priority can either be fixed at system design time or can be calculated dynamically by the scheduler at runtime.

4.4.2. Fixed-Priority Scheduling

If a system is scheduled using a fixed-priority scheduling algorithm, each task is assigned a fixed priority which will not be changed during the runtime of the system. There are several strategies on how to decide which task will be assigned which priority.

Definition 4.15 (Rate-Monotonic Scheduling)
In a system scheduled under Rate-Monotonic Scheduling (RMS), each task is assigned its priority according to its period. The task with the smallest period is assigned the highest priority [LL73].

RMS has been proven an optimal fixed-priority scheduling algorithm in case that the system is strictly periodic and each task's deadline equals its respective period. This means that, in this case, if there is any fixed-priority schedule which leads to a schedulable system, then RMS will also provide a valid schedule. In order to be able to quantify the quality of a scheduling algorithm, the notion of the maximum *load* of a system is introduced:

Definition 4.16 (Maximum System Load of Periodic Systems)
The maximum load u *of a system with a task set* Γ *is defined as the sum over all tasks' WCET divided by their respective period [LL73]:*

$$u = \sum_{i \in \Gamma} \frac{C_i}{T_i} \tag{4.24}$$

Liu and Layland [LL73] derive a safe lower bound on the system load u up to which a system consisting of periodically triggered tasks with implicit deadlines is definitely schedulable using a fixed-priority scheduler:

Definition 4.17 (Utilization bound on Fixed-Priority Scheduling Algorithms)
A system consisting of n *tasks with implicit deadlines, which are all scheduled strictly periodically, is provably schedulable under an optimal fixed-priority scheduling algorithm with a system utilization* u *up to [LL73]:*

$$u \leqslant n \cdot \left(2^{\frac{1}{n}} - 1\right) \tag{4.25}$$

For an infinite number of tasks, this results in a maximum load of $u = \ln 2 \approx 0.693$.

For any system load beyond this boundary, a fixed-priority scheduler *may* be able to provide a valid schedule in which all tasks provably meet their deadlines. However, depending on WCETs and periods of the tasks, it is not guaranteed that such a schedule exists.

RMS has been designed for strictly periodic systems where each task is considered to be executed every fixed time interval (cf. Section 4.3.2). If two tasks have the same period, they are assigned the same priority. The implications of this special case for the concrete scheduling algorithm are implementation-specific. If multiple tasks are assigned the same priority, it is not generally defined whether the tasks can preempt each other or not. To prevent such disambiguities, one of the tasks can be assigned a higher priority over the other.

In the course of thesis, a task τ_i can only be preempted by another task τ_j if $P_i > P_j$ (cf. Definition 4.8 which defines that higher numerical values denote a logically lower priority). This implies that a task cannot preempt another already running instance of the same task.

Deadline-Monotonic Scheduling (DMS) can be seen as an extension over RMS in case that the minimum period of at least one task in the system differs from its respective deadline. In a strictly periodic system, this means that for at least one $\tau_i \in \Gamma$, $D_i \neq T_i$.

Definition 4.18 (Deadline-Monotonic Scheduling)
In a system scheduled under Deadline-Monotonic Scheduling (DMS), each task is assigned its priority according to its deadline. The task with the smallest deadline is assigned the highest priority [LL73].

DMS has been proven an optimal *fixed-priority* scheduler in case of a periodic system by Liu and Layland [LL73]. Both derivation and proof are given in [LL73]. This means that if there exists a valid schedule with fixed priorities, DMS will be able to provide a valid schedule.

4.4.3. Dynamic-Priority Scheduling

The aforementioned scheduling algorithms have in common that the priority of a task is fixed over the whole runtime of the system. These fixed-priority scheduling algorithms may be able to provide a schedule even for high system loads. However, as described in the previous section, they do not guarantee schedulability at high system loads.

Dynamic-priority schedulers, on the other hand, dynamically determine each task's priority at runtime, depending on the current state of the system. Using a dynamic-priority scheduler, a given τ_i may preempt another task τ_j at one

instance in time, while being preempted by another instance of this task τ_j at some later point. This allows for guaranteed schedulability for system loads of up to 100 % in case of implicit deadline systems, at the cost of a much more complex scheduler implementation. One of the most popular dynamic-priority scheduling algorithms is EDF.

Definition 4.19 (Earliest Deadline First Scheduling)
In a system scheduled under Earliest Deadline First (EDF), each task's priority is calculated at runtime according to its absolute deadline. The task with the smallest deadline is assigned the highest priority [LL73].

This means that, each time a new task is ready for execution, the scheduler will calculate the wall-clock time at which the task has to be finished and compare it to the wall-clock time at which each other currently running task must have finished its execution. Then, the task which must be finished soonest will be scheduled for execution and any currently running other task will be suspended. This comparison of a task's timing properties with the actual wall-clock time leads to the fact that priorities are no longer known at system design time. Instead, each task may have the highest or lowest priority at some point of the system execution.

EDF has been proven to be an optimal scheduling algorithm [LL73]. This means that, if any scheduling algorithm is able to provide a schedule which leads to a schedulable system, then EDF will also provide a valid schedule.

4.4.4. Limitations

All these scheduling strategies and their proofs of optimality feature one common limitation: They neglect both the cost of the scheduler itself and the costs needed to switch between different tasks.

The first limitation can easily be circumvented by introducing the scheduler as another task. The implications and impact of this will be shown in the evaluation of this thesis. Therefore, if a multi-tasking system features a dedicated scheduler then, by definition, this scheduler is the highest-priority task within the system.

Neglection of context switching costs is not that simple to tackle. In fact, it may lead to the result that in practice sometimes a dynamic-priority scheduler like EDF may not provide a valid schedule due to a high number of preemptions while a fixed-priority scheduler will. While accounting for these timing penalties within the upcoming chapters, this work is focusing on code-level optimizations. Thus, the question of choosing the right scheduler, as mentioned before, is not tackled explicitly throughout the upcoming chapters. However, of course, an optimization may be (manually or automatically) re-run with different scheduling

strategies to find out whether one scheduling strategy outperforms the other in a given specific scenario.

4.5. Schedulability Analysis

This section provides an overview over different analysis techniques for the scheduling algorithms presented in the previous section. It is not meant as a complete guide to all existing methods on schedulability and response time analysis. Rather, only those techniques are presented which are needed and used in the upcoming framework to provide a holistic model for code-level optimizations of embedded hard real-time multi-tasking systems. Additionally, the analysis of periodic fixed-priority systems is slightly improved over previous approaches in order to be able to provide a tighter WCRT estimate in case of realistic systems with non-negligible preemption costs.

The so-called Worst-Case Response Time (WCRT) of a task is the maximum time span from the moment the task is triggered to be executed until its end including all preemptions and blocking times due to interrupts or higher-priority tasks.

Definition 4.20 (Worst-Case Response Time)
The Worst-Case Response Time (WCRT) of a task is the maximum possible interval between any activation of the task and its respective finishing, including all blocking times and interferences of other tasks. We denote the WCRT of a task τ_i *using the symbol* r_i*.*

Definition 4.21 (Schedulability)
A system is called schedulable if and only if the WCRT r_i *of each task* τ_i *in the system is lower than or equal the task's respective deadline* D_i*:*

$$\forall \tau_i : r_i \leqslant D_i \tag{4.26}$$

Generally speaking, schedulability analysis provides methods to verify if a given real-time system will provably meet all timing constraints under any circumstances, while response time analysis returns distinct values on the WCRT of each task. Therefore, both approaches can be used to determine whether a system is schedulable, but only WCRT analysis techniques can give information about the actual worst-case timing behavior of a given task.

4.5.1. Analysis of Strictly Periodic Systems

The following subsections tackle the analysis of strictly periodically triggered systems which follow the task model from Definition 4.9. Section 4.5.1.1 introduces

the response time analysis for systems using a fixed-priority scheduling algorithm. Section 4.5.1.2 proceeds with schedulability analysis techniques for dynamic-priority systems with implicit deadlines.

4.5.1.1. Fixed-Priority Systems

The WCRT analysis for periodical fixed-priority systems was originally proposed by Joseph et al. [JP86] and Lehoczky et al. [LSD89] as a fixed-point iteration formula:

$$r_i = c_i + \sum_{j=0}^{i-1} \left(\left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j \right) \tag{4.27}$$

The WCRT of task τ_i is composed of the WCET of τ_i itself, plus the time for which τ_i is blocked due to the execution of higher-priority tasks τ_j, $j < i$. This is reflected by Eq. (4.27). The first term c_i denotes the WCET of τ_i. The summation term adds up all the blocking times due to higher priority tasks. The ceil function within the summation calculates the number of times τ_i may be preempted by a given higher-priority task τ_j. The maximum number of interruptions is given by dividing the WCRT of τ_i by the period of the higher priority task τ_j, T_j. The preemption time is then calculated by multiplying this maximum number of preemptions with the WCET of τ_j.

Due to the fact that the summation in turn depends on the response time r_i of task τ_i, this formula must be re-calculated iteratively. The idea is to start calculating r_i for each task starting with the highest-priority task τ_0. As a safe lower bound on the WCRT of each task – and therefore the starting point of the fixed-point iteration – task i's WCET may be used. If the fixed-point iteration converges to a value lower than or equal to the respective deadline D_i, the task is said to be schedulable and the WCRT of the task with the subsequently lower priority is calculated.

If all WCRTs of all tasks are below their respective deadlines, the system is schedulable. If the response time of at least one task is greater than its respective deadline, the analysis is aborted for all remaining tasks and the system is considered to be not schedulable.

This approach assumes an ideal system where an evicting task can preempt any lower-priority task instantaneously without any blocking times b. Additionally, it is silently assumed that the process of preempting a task does not inflict additional timing delays. If these additional penalties are not negligible, a safe over-estimation of the additional delays has to be added to each task's WCET, obviously resulting in large pessimism. Staschulat et al. [SSE05] present an extended WCRT analysis to explicitly account for blocking delays and eviction penalties:

$$r_i = c_i + b_i + \sum_{j=0}^{i-1} \left(\left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j + e_{i,j}^{(r_i)} \right) \tag{4.28}$$

b_i describes the so-called *blocking time*. This is the time that the task's execution may be delayed, even if the task actually has the highest priority and would be scheduled for execution. In practice, blocking times may be inflicted by, e.g., hardware interrupt latencies.

The term $e_{i,j}^{(r_i)}$ describes the overall inter-task penalties (e.g., context switching costs) inflicted by τ_j and all tasks τ_n, $j < n < i$ to τ_i. It is therefore dependent on the current WCRT r_i, which is indicated by the $^{(r_i)}$.

For each individual preemption of a task τ_i by another task τ_j, a safe approximation $e_{i,j}$ can be defined which describes the maximum timing penalty inflicted to τ_i. Using this eviction penalty $e_{i,j}$, Eq. (4.28) can be extended as follows:

Proposition 4.2 (Reformulated WCRT analysis)

$$r_i = c_i + b_i + \sum_{j=0}^{i-1} \left\{ \left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j + e_{i,j}^{(r_i)} \right\} \tag{4.29}$$

$$e_{i,j}^{(r_i)} = \sum_{n=j+1}^{i} \left[\min \left(\left\lceil \frac{r_i}{T_n} \right\rceil \cdot \left\lceil \frac{r_n}{T_j} \right\rceil, \left\lceil \frac{r_i}{T_j} \right\rceil \right) \cdot e_{n,j} \right] \tag{4.30}$$

Tightening $e_{i,j}^{(r_i)}$ by using the $\min()$ term was not proposed by Staschulat et al. but has been introduced as part of this thesis.

Proof. The basic structure of Eq. (4.29) is the same from Eq. (4.27). The WCRT r_i of task τ_i is calculated by summing up the WCET c_i of τ_i itself and the WCETs c_j of all higher priority tasks $\tau_j, j < i$.

This proof shows that Eq. (4.30) gives a safe upper bound on the preemption costs inflicted to τ_i by each higher-priority task τ_j, including additional preemption overheads caused by subsequent preemptions of "middle" priority tasks, i.e., tasks with a priority lower than j but higher than i.

In a worst-case scenario, τ_j will not directly preempt τ_i. Instead, τ_i will be preempted by τ_{i-1}, which is preempted by τ_{i-2}, ... until finally τ_j preempts τ_{j+1}. Thus, when considering the preemption penalties inflicted to τ_i by τ_j, any possible nested preemptions must be considered as well. This is expressed by the sum in Eq. (4.30) which iterates over all tasks with a priority n, $j < n \leqslant i$, thus including both the direct preemption of τ_i by τ_j and all nested preemptions.

In the following, it is shown that both terms within the min() expression of Eq. (4.30) are always safe bounds on the number of possible preemptions of τ_i by τ_n with $j < n \leqslant i$. As a result, the min() expression also always provides a safe bound on the number of preemptions. Therefore, in conclusion, the term $e_{i,j}^{r_i}$ provides a safe upper bound on the complete preemption overhead.

Obviously, the maximum interval in which any preemptions may occur is the WCRT of τ_i. Therefore, any task τ_n with $j < n \leqslant i$ cannot be preempted more often by τ_j, than τ_j is actually being triggered within τ_i. Therefore, for any given τ_n, $\left\lceil \frac{r_i}{T_j} \right\rceil \cdot e_{n,j}$ is a valid upper bound for τ_n's preemption costs inflicted by task τ_j.

This yields:

$$r_i = c_i + b_i + \sum_{j=0}^{i-1} \left\{ \left\lceil \frac{r_i}{T_j} \right\rceil \cdot (c_j) + \sum_{n=j+1}^{i} \left\lceil \frac{r_i}{T_j} \right\rceil \cdot e_{n,j} \right\} \tag{4.31}$$

which is equivalent to Eqs. (4.29) and (4.30) for $\left\lceil \frac{r_i}{T_j} \right\rceil \leqslant \left\lceil \frac{r_i}{T_n} \right\rceil \cdot \left\lceil \frac{r_n}{T_j} \right\rceil$.

It is now shown that $\left\lceil \frac{r_i}{T_n} \right\rceil \cdot \left\lceil \frac{r_n}{T_j} \right\rceil$ always guarantees a safe upper bound for the preemption penalties as well: As described above, any task τ_n with $n > j$ may be evicted by τ_j a maximum of $\left\lceil \frac{r_n}{T_j} \right\rceil$ times. Obviously, τ_j can only preempt τ_n if τ_n is currently executed. Within τ_i's WCRT r_i, τ_n is executed a maximum of $\left\lceil \frac{r_i}{T_n} \right\rceil$ times. For a safe upper bound, it is assumed that at each execution it will be preempted the maximum amount of times by τ_j. Therefore, the product $\left\lceil \frac{r_i}{T_n} \right\rceil \cdot \left\lceil \frac{r_n}{T_j} \right\rceil$ is a safe bound on the number of preemptions.

As a result, both terms within the min() operation always provide sound upper bounds on the number of evictions of τ_n by τ_j. Thus, it is safe to choose the smaller of both terms in order to get a tighter estimate on the WCRT r_i. □

Obviously, Eq. (4.29) will never return higher WCRT bounds than Eq. (4.31). However, in cases where middle-priority tasks are executed with very low frequency, it is able to reduce the bound for the WCRT. As a result, this approach only adds additional penalties if both the higher and the respective lower priority task are executed in the given interval r_i multiple times.

4.5.1.2. Dynamic-Priority Systems

This section briefly addresses the schedulability analysis of periodically triggered tasks which are scheduled using dynamic-priority scheduling algorithms like, e.g., EDF with implicit deadlines. This means that only systems with $T_i = D_i$ are

considered in this section. In this case, schedulability analysis can be simplified enormously. This was originally described by Liu and Layland [LL73]. An optimal dynamic-priority scheduling algorithm is, per definition, able to schedule any implicit-deadline system with a maximum workload u below 100 %.

Proposition 4.3 (Schedulability of Periodic Dynamic-Priority Systems)
Given a task set Γ consisting of N periodic tasks τ_i, $i \in [0, \ldots, N-1]$. If each task has a fixed activation period T_i and an implicit deadline $D_i \equiv T_i$, then the system is schedulable if and only if:

$$u = \sum_{i=0}^{N-1} \frac{c_i}{T_i} \leqslant 1 \tag{4.32}$$

Proof. The proof is given in [LL73]. □

This schedulability test, however, does not account for any timing penalties due to, e.g., context switches on task preemptions. If these costs cannot be neglected, a safe over-approximation has to be added to each task's WCET, leading to high pessimism in the analysis.

Despite these drawbacks, the test provides a simple necessary condition for the schedulability of periodical systems. In this work, no more sophisticated schedulability tests for strictly periodical systems are used, as the schedulability test for arbitrary systems shown in the next subsection can obviously be applied to strictly periodic systems with arbitrary deadlines as well.

4.5.2. Analysis of Arbitrarily Triggered Systems

To analyze multi-tasking systems which are triggered in an arbitrary fashion, event functions which were introduced in Section 4.3.3 are used. Analogous to strictly periodically triggered systems, a maximum system load u can be defined.

Definition 4.22 (Maximum Load of Arbitrarily Triggered Systems)
Given a task set Γ consisting of N tasks, where the density function $\eta_i(\Delta t)$ models the activation pattern of each task $\tau_i, i \in [0; N-1]$, the maximum load u of the system is defined as:

$$u = \lim_{\Delta t \to \infty} \sum_{i=0}^{N-1} \frac{c_i \cdot \eta_i(\Delta t)}{\Delta t} \tag{4.33}$$

Practically spoken, the maximum system load is defined as the fraction of an infinitely large time interval in which the system is *not* idle. For periodical

systems, the definition is reduced to the formula given in Definition 4.16. For periodically recurring activation patterns, the system's behavior is repeated after a finite amount of time. This time interval is called the *hyper-period* of the system.

Definition 4.23 (Hyper-Period)
The hyper-period *of a system is the least common multiple of all fundamental periods which occur within a task set* Γ*. Due to the definition as least common multiple over all fundamental periods, the system's worst-case behavior repeats after each hyper-period.*

If the task set contains some aperiodical activations like, e.g., an initial burst, their share in the system load will diminish for an infinitely large time interval Δt. To illustrate this, imagine that a task τ_i is only triggered a fixed number of Y times. Then, obviously, $u = \lim_{\Delta t \to \infty} \frac{Y}{\Delta t} \cdot c_i = 0$

Therefore, in practice, instead of using the lim calculation, the system load can be obtained by using the least common multiple over all periodical activation patterns as time interval Δt and an adapted density function $\eta_i'(\Delta t)$, which only describes the periodically recurring activation patterns.

The following sections present schedulability analysis techniques which can be used to analyze arbitrarily triggered systems running on one computational unit using event streams for systems scheduled under both fixed- and dynamic-priority scheduling with arbitrary deadlines. Despite the different approaches, a maximum system load below 100 % is always a necessary requirement for the system to be schedulable. In case of $u > 1$, some task will inevitably miss its deadline at some point. Therefore, the system is unschedulable without any further tests being necessary.

In the following, the term *busy window* is used in order to describe the length of the time interval in which the system is not idle. Formally, the busy window can be defined as follows:

Definition 4.24 (Busy Window)
The busy window ΔT_B *is the maximum length of the interval in which only the currently analyzed task or any task with a higher priority is executed and the processor is not idle [Leh90].*

The busy window can be used as an upper bound for the time frame in which a system's schedulability must be verified in order to prove the schedulability of the system.

4.5.2.1. Fixed-Priority Systems

The original WCRT analysis for strictly periodical systems using fixed-priority scheduling, as shown in Eq. (4.27) was later extended by Lehoczky in order to support arbitrary deadlines [Leh90]. The approach was then adapted by Tindell et al. [TBW94] for tasks executed with jitter or bursts. Kollmann et al. [Kol+10] then reformulated the analysis for event-triggered systems using event density functions. They showed that the WCRT r_i of a task τ_i can be calculated as:

$$r_i = \max_{\forall K \in [1,\ldots,\eta_i(\Delta T_B))]} \{r_{i,K} - \delta_i(K)\} \tag{4.34}$$

$$r_{i,K} = \min \left\{ \Delta t \,|\Delta t = K \cdot c_i + \sum_{j=0}^{i-1} [\eta_j(\Delta t) \cdot c_j] \right\} \tag{4.35}$$

Eq. (4.35) is analogous to the previously presented WCRT analysis for fixed-priority systems in Eq. (4.27). The differences are solely that the ceil function which could be used to calculate the maximum number of preemptions of task τ_i by a higher priority task τ_j is substituted by the event density function $\eta_i(\Delta t)$. Additionally, the fixed-point iteration is explicitly expressed by the $\Delta t|\Delta t = \ldots$ term.

The additionally introduced factor K accounts for multiple instances of the currently analyzed task τ_i itself. It denotes the number of times that τ_i is executed within the analyzed time interval. This is important in case that τ_i's deadline D_i is greater than its minimal inter-arrival time $\delta_i(2)$, meaning that the task is allowed to finish its execution even after another instance of the same task has already been triggered. Calculating the WCRT of τ_i solely for one instance (i.e., using the original equation Eq. (4.27) by Joseph et al.) may not correctly predict the worst-case timing behavior of τ_i over the whole system runtime. The reason for this is that subsequent instances of a task may suffer from additional blocking times inflicted by earlier instances which are still running when the subsequent instance is triggered for execution.

Eq. (4.35) is used to determine the worst-case time that passes when executing K consecutive instances of a given task τ_i. E.g., for $K = 2$, $r_{i,K}$ will return the maximum time needed until two subsequent instances of τ_i have finished their execution. Eq. (4.34) is then used to find *the* maximum WCRT of τ_i over all of its instances: $r_{i,K}$ gives the accumulated worst-case timing to execute K consecutive instances of task τ_i. In order to determine the WCRT of the K'th instance, the time overhead inflicted due to all $K-1$ previous instances must be eliminated. This is achieved by subtracting the minimum interval needed in order for the K'th instance to be activated from the overall time $r_{i,K}$ which will pass in order to

execute all K instances (Eq. (4.35)). The result is the WCRT of the K'th instance of task τ_i. The maximum over all of these WCRTs is then *the* WCRT of τ_i.

Eqs. (4.34) and (4.35) do not account for eviction penalties due to context switching costs, CRPD, ... In order to account for these costs, the accounting of eviction penalties from Eq. (4.29) can easily be integrated into Eq. (4.35) by substituting the periods with the respective density functions. This leads to the final equations for the WCRT analysis of arbitrarily triggered task sets:

Proposition 4.4 (WCRT Analysis of Arbitrarily Triggered Systems Using Fixed-Priority Scheduling)
The WCRT of an arbitrarily triggered task τ_i can be calculated as:

$$r_i = \max_{\forall K \in [1,\ldots,\eta_i(\Delta T_B)]} \{r_{i,K} - \delta_i(K)\} \tag{4.36}$$

$$r_{i,K} = \min\left\{ \Delta t \,|\Delta t = K \cdot c_i + \sum_{j=0}^{i-1} \left[\eta_j(\Delta t) \cdot c_j + e_{i,j,\Delta t}\right] \right\} \tag{4.37}$$

$$e_{i,j,\Delta t} = \sum_{n=j+1}^{i} \left[\min\left[\eta_n(\Delta t) \cdot \eta_j(D_n), \eta_j(\Delta t)\right] \cdot e_{n,j} \right] \tag{4.38}$$

To simplify notations, we express the maximum possible K for a specific task τ_i from Eq. (4.34) as $\hat{K}_i$:

$$\hat{K}_i \coloneqq \eta_i(\Delta T_B) \tag{4.39}$$

ΔT_B denotes the so-called *busy window*.

The system is not schedulable if $r_i > D_i$ for any $\tau_i \in \Gamma$. By definition, the maximum time interval Δt which has to be analyzed is the busy window defined in Definition 4.24.

The hyper-period from Definition 4.23 can also be used as a safe upper bound on the busy window for systems with periodically recurring activation patterns. This stems from the fact that the system's behavior repeats after the hyper-period. If the busy window is larger than the hyper-period, the system is definitely not schedulable, as the system load is beyond 100 %, and thus - in the long run - a task will have to miss its deadline. Therefore, for a system to be schedulable, the busy window must be smaller than or equal to the hyper-period. Thus, using the hyper-period as an upper bound is safe.

Proposition 4.5
For systems with periodically recurring activation patterns, a safe upper bound on ΔT_B is given by the hyper-period of the task set, as long as the system load $u \leqslant 1$ and $r_i \leqslant D_i$.

For a task τ_i with a deadline D_i lower than or equal to its respective minimal period, i.e., $\eta_i(2) \geqslant D_i$, it is sufficient to use D_i as maximum Δt.

Proof. The proof for the first part is given in [Kol+10] but may also easily be deduced: By definition, the system's behavior repeats after the hyper-period. The only way the system behavior does not exactly repeat after the hyper-period is if the resource demand within one hyper-period exceeds the hyper-period (i.e., $u > 1$). However, the system is trivially unschedulable in this case.

For the second part of the proposition, assume that the currently analyzed task τ_i is schedulable, i.e., $r_i \leqslant D_i$. Then, by definition, the busy window must be smaller than or equal to the task's deadline: $\Delta T_B \leqslant D_i$. If $r_i > D_i$, then the system is not schedulable, and no bigger time interval has to be investigated. □

For systems with a partially non-periodical activation pattern (e.g., if at least one task may have an initial burst), the complete system behavior does obviously not repeat. However, the periodical part of the system will repeat after the hyper-period composed of all periodic parts of the activation pattern. As a result, the maximum busy window is the least common multiple of all fundamental periods in the system, plus the maximum time interval in which non-periodical activation patterns may occur. For systems which do not have *any* periodical activation pattern, the complete system behavior must be analyzed, as the system behavior cannot be described in a compressed fashion.

Unfortunately, due to the fact that the hyper-period is the least common multiple of all fundamental periods within the task set, it may easily become very big. Approaches to reduce the maximum interval to be analyzed exist [BMR90; Pol+09; SH98]. However, these approaches assume that the WCET is fixed and that the actual WCRT is below the maximum allowed one. This basically means that, in case of analyzing a schedulable system, the analysis can be ended prior to analyzing the task set up to its hyper-period. Due to the nature of the schedulability-oriented WCET optimizations tackled in this thesis, these approaches cannot be applied. On the one hand, the WCETs can *not* be assumed to be fixed, as it is exactly the WCETs which the framework will aim to optimize. On the other hand, if the system is already schedulable prior to the optimization, the optimization does not have to be applied in any way. Therefore, for the systems which are analyzed in this thesis as part of the optimizing compiler framework, it is always assumed that the unoptimized system is *not* schedulable, i.e., at least one task's WCRT exceeds its respective deadline. Therefore, approaches to reduce the busy window or to determine an initial minimal interval cannot be applied.

However, if a task's deadline is lower than or equal to its minimal period (i.e., $D_i \leqslant \delta_i(2)$), then $\hat{K}_i \equiv 1$, as the task cannot be blocked by itself. In this case, the task's deadline D_i can be used as a safe upper bound on ΔT_B (cf. Proposition 4.5).

As a practical solution for systems where the busy window is defined by the hyper-period, tasks' fundamental activation periods may be tightened in order to reduce the least common multiple, thus reducing the hyper-period. This has been shown by Xu [Xu10] for strictly periodical systems. Of course, tightening activation patterns will introduce more pessimism, thus decreasing the amount of task sets which can be repaired and turned into provably schedulable systems by the optimization model. The user has to choose a trade-off between analysis quality and analysis time depending on the specific problem.

The remaining question to tackle when investigating Eq. (4.35) is to determine the actual values $\Delta t \in [0, \Delta T_B]$ which have to be analyzed. Eq. (4.35) does not explicitly limit these, such that in case of real-valued time units, an infinite number of time intervals would have to be tested, and in case of an integer-valued time unit (e.g., if time units are given in CPU clock cycles) still ΔT_B time intervals would have to be investigated which is easily computationally infeasible. However, the event density function $\eta_j(\Delta t)$ from Definition 4.11 is piecewise constant. For a periodic system, a new event is triggered at a fixed rate and even for arbitrary systems, the density function will not increase strictly monotonically. Therefore, Eq. (4.35) has to be evaluated only at the points of discontinuity. These points of discontinuity can easily be retrieved by building the union over all interval functions of the tasks $\tau_l, 0 \leqslant l < i$. Tasks with a priority lower than or equal to τ_i do not need to be considered as they cannot preempt τ_i and do therefore not contribute to its WCRT r_i.

4.5.2.2. Dynamic-Priority Systems

For systems scheduled under an optimal dynamic scheduler like EDF, schedulability analysis can be performed using the so-called processor demand test proposed by Baruah [Bar03]:

Proposition 4.6 (Schedulability Analysis of Arbitrary Systems with Dynamic-Priority Scheduling)
A system is schedulable under EDF if the following condition holds for any time interval Δt *[Bar03]:*

$$\Delta t \geqslant \sum_{\forall \tau_i \in \Gamma} \left[\eta_i (\Delta t - D_i) \cdot c_i \right] \tag{4.40}$$

Proof. The proof is given in [Bar03]. □

Due to the nature of the *dynamic* priority scheduling algorithm EDF, a schedulability analysis cannot easily predict which task is preempted by which other task at runtime in a worst-case scenario. However, due to the optimality of EDF,

if there is any way to execute the tasks such that they all meet their deadline, this will also apply for EDF. As a result, the schedulability test in Eq. (4.40) calculates the *resource demand* of the system within a given time interval Δt. The resource demand is calculated by summing up the number of executions of each task which have to finish its execution within Δt, multiplied with its respective WCET. If this resource demand is smaller than or equal to the analyzed time interval Δt, the system meets all timing constraints in this specific time interval. When repeating this analysis for *all* time intervals, the system is guaranteed to be schedulable.

The maximum time interval Δt to analyze is bound by the maximum busy window of the system. For periodical systems, Proposition 4.5 can be applied directly to EDF as well. Also, the reasoning from the previous section on systems which are not fully periodical are independent from the used scheduling algorithm and and are thus also valid for EDF.

The question left open is which concrete values of $\Delta t \in [0, \Delta T_B]$ must be analyzed in order to determine whether a given system is schedulable or not. Obviously, the event density function is not dependent on whether fixed or dynamic priorities are used. Therefore, the density function $\eta_i(\Delta t)$ and the corresponding interval function $\delta_i(k)$ are still piecewise constant. In order to comply with its timing requirements, a given task τ_i demands to finish its execution at the latest D_i time units after it has been triggered for execution. At any point in time prior to its deadline, the resource *demand* of this instance of τ_i is zero, as it does not matter *when* the task finishes its execution, as long as it is not after the respective deadline. Subsequently, it is sufficient to evaluate these values for Δt for which the resource demand of a task increases:

$$\Delta t = \{\delta_i(k) + D_i\}, k = 0, \ldots, \eta_i(\Delta T_B) \tag{4.41}$$

To verify schedulability of the complete system, this must be performed for any task $\tau_i \in \Gamma$.

Eq. (4.40) neglects any context switching costs e. In contrast to fixed-priority scheduling, in dynamic-priority scheduling, each task may have the highest or lowest priority at some point in time. Therefore, it is non-trivial to determine a feasible worst-case pattern of evictions between each task. However, a safe over-approximation can be given: The preemption penalty e_j is defined as the maximum penalty which task τ_j may inflict by one preemption to any other task τ_j in the system. If $e_{i,j}$ is the preemption penalty inflicted to task τ_i by τ_j (as introduced for fixed-priority scheduling algorithms in Eq. (4.28)), e_j can be obtained by:

$$\forall i \neq j : e_j = \max(e_{i,j}) \tag{4.42}$$

For EDF, each task τ_j cannot preempt τ_i more often than the number of times that τ_j is actually being executed within the analyzed time interval Δt. Thus, a safe schedulability test which accounts for preemption penalties is given by:

$$\Delta t \geqslant \sum_{\forall \tau_i \in \Gamma} \left[\eta_i \left(\Delta t - D_i \right) \cdot \left(c_i + e_i \right) \right] \tag{4.43}$$

4.5.3. Analysis of Context Switching Costs

In the previous sections, context switching costs inflicted by a task τ_j to another task τ_i were included in the schedulability analyses by a dedicated term, $e_{i,j}$. This allows for the decoupling of scheduling analysis from context switching cost analysis. I.e., the analysis of the penalty which is inflicted to a task τ_i by another task τ_j due to a preemption is not dependent on the scheduling algorithm.

There are two sources for context switching costs: One are fixed costs which are created due to, e.g., hardware interrupt latencies. These are platform-dependent but do usually not vary between tasks. Timing costs to save and restore the preempted task's context do not have to be accounted as context switching costs. The scheduling task has to take care of saving and restoring task contexts. Thus, this timing overhead is already included in the WCET of the scheduler.

The other major source for context switching costs is CRPD. If the system features caches, a preempting task may evict cache lines which were loaded by the preempted task. If the preempted task resumes, it may suffer from additional cache misses which were not accounted for by a single-tasking WCET analysis.

The general idea to tackle this issue is to perform a Useful Cache Block (UCB)/Evicting Cache Blocks (ECB) analysis: First, the set of *useful* cache blocks of the preempted task is created. This set comprises all blocks in memory which would lead to a cache hit if the preempted task was executed in a stand-alone manner. Then, a set of *evicting* cache blocks of the preempting task is created. This set contains all blocks of the evicting task which may subsequently purge the preempted task's cache line from the cache.

The actual analysis heavily depends on the cache's replacement policy. In an N-way set-associative cache using LRU replacement policy (cf. Section 3.2.1), a mere conflict between one ECB and one UCB may not yet lead to a CRPD, due to the cache's associativity. Previous works exist which perform sophisticated analyses in order to tighten these results [ADM12; KFM11; Kle15]. One commonly cited technique is the so-called "resilience analysis" by Altmeyer et al. [ADM12]. This operates on program traces and uses the notion of "resilience" to determine a useful block's age and whether it may be evicted by a preempting cache block.

If a task τ_i is preempted by another task τ_j more than once, the subsequent preemptions may have a smaller CRPD than the first one. This stems from the fact, that the task is preempted at different points of its execution, and therefore naturally the CRPD can be different. Sophisticated analysis techniques could be

thought of that find the feasible worst-case preemption points for each task and subsequently tighten the timing penalty inflicted by CRPD by using only these feasible values. Staschulat et al. [SSE05] analyzed this behavior and found that the improvements by such an analysis are quite small. In this work, it is therefore assumed as a safe over-approximation that each preemption of a given task τ_i by another task τ_j will inflict the worst-case CRPD.

For set-associative caches with an associativity larger than 1, another issue arrises: In case of nested preemptions (e.g., τ_0 preempts τ_1 which has previously preempted τ_2), the CRPD due to the nested preemptions may be larger than the sum of the pair-wise CRPDs.

Introducing such nested preemptions into the schedulability analysis comes with a major increase in complexity. Evaluations in Chapter 12 will show that even a simple CRPD-aware optimization drastically increases an optimization's solution search space. Thus, in the following, these nested preemptions are not going to be considered. Instead, as a safe over-approximation, a cache conflict between a preempting and a preempted task is considered to always cause a cache eviction, despite the cache associativity. Therefore, the eviction penalties $e_{i,j}$ can safely be considered pair-wise only.

5. Integer-Linear Programming

Integer-Linear Programming (ILP) expresses an optimization problem as set of linear constraints and an objective function. The constraints are equations or inequations with real-valued constant factors and integer variables. The objective function is a possibly weighted sum of integer variables which is to be either minimized or maximized. As shown in Chapter 2, ILPs have been proven a powerful tool for modeling code optimizations targeting a system's worst-case timing behavior. *Integer*-linear programming is used, because the optimization can usually be reduced to a combinatorial problem expressed by integer or even binary values. E.g., the decision whether a loop should be unrolled or whether a given piece of code is to be allocated to this or that memory region is a binary decision. Modeling timing behavior can also easily be expressed by integer numbers if CPU clock cycles are used as a time base.

Section 5.1 gives a brief overview of linear programming and integer-linear programming as its special case. Section 5.2 continues by showing how Boolean expressions like a logical AND or OR, as well as conditional dependencies may be expressed as a set of ILP constraints and variables. These formulations will be used frequently in the upcoming chapters.

5.1. Introduction

Many optimization problems can be modeled as a set of linear equations combined with an objective function. Modeling and solving such systems of equations is called linear programming. When the variables may be integer only, the problem is subsequently called *integer*-linear programming.

Mathematically, an ILP may be expressed as a real-valued coefficient matrix $\mathbf{A}$, a variable vector $\mathbf{x}$ with values in $\mathbb{N}^+, 0$ and a real-valued constant vector $\mathbf{c}$. These equations are accompanied by a minimization function which consists of the variable vector $\mathbf{x}$ multiplied with another real-valued constant vector $\mathbf{b}$. Thus, a generic ILP is usually expressed as:

$$\mathbf{A}\boldsymbol{x} = \mathbf{c} \tag{5.1}$$

$$\min\left(\mathbf{b}^{\mathrm{T}}\mathrm{x}\right) \tag{5.2}$$

At first sight, this formulation brings some limitations: First, the notation restricts the equation system to a strict "equal". However, often it is to be expressed that a certain variable should be "greater than or equal to" or "lower than or equal to" some constant value. This can be expressed in the matrix formulation by adding an additional slack variable. Second, the notation does not allow for negative values for the solution vector. This can be countered by substituting an x_i which allows for negative values by two non-negative variables x_i^+ and x_i^- with $x_i = x_i^+ - x_i^-$ [PLB12, p. 139f.]. Finally, the objective function is usually a minimization only. In case of a maximization problem, this may easily be expressed as a minimization problem by negating Eq. (5.2) and minimizing the negated expression.

Expressing an ILP using the matrix formulation from Eq. (5.1) may be sensible for describing solving algorithms but is usually impractical in order to discuss applied ILP modeling in practice. Instead of using one vector $\mathbf{x}$ holding the ILP variables, variable values are denoted by lower-case letters. Constant values are denoted by upper-case letters. The ILP is not described as one huge matrix with its full rank. Instead, stand-alone (in)equations are used in order to restrict the ILP as needed. Variables which are not relevant for one constraint are not printed. In the matrix formulation from above, the corresponding values of the $\mathbf{A}$ matrix will be chosen as 0. For example, an ILP consisting of three (in)equations would be written as follows:

$$a_0 \cdot A_0 + a_1 \cdot A_1 + \cdots + a_I \cdot A_I \leqslant D_0 \tag{5.3}$$

$$b_0 \cdot B_0 + b_1 \cdot B_1 + \cdots + b_J \cdot B_J \geqslant D_1 \tag{5.4}$$

$$c_0 \cdot C_0 + c_1 \cdot C_1 + \cdots + c_K \cdot C_K = D_2 \tag{5.5}$$

The lower-case a_i, b_j and c_k denote integer-valued variables in the (in)equations. A_i, B_i and C_i denote real-valued constant factors which each respective integer-variable is multiplied with. Finally, the D_0, D_1, D_2 are also real-valued constants. Depending on the logical expressions, the relational operators $\geqslant$, $\leqslant$ and $=$ may be used in order to link both sides of an (in)equation. Each (in)equation may consist of an arbitrary number of summation terms, and the ILP may consist of an arbitrary number of (in)equations. Finally, for a minimization problem, the objective function is denoted like:

$$\min\left(o_0 \cdot O_0 + o_1 \cdot O_1 + \cdots + o_L \cdot O_L\right) \tag{5.6}$$

With o_l being integer-valued variables and O_l being their corresponding constant real-valued multiplication factors. In case of a maximization problem, the objective is written as $\max(\dots)$, accordingly.

A special case of integer variables are the so-called *binary variables*. These are integer variables which are bound to the interval $[0, 1]$, i.e., they may only hold the

values 0 and 1. These variables are usually treated specially by an ILP solver as it may apply different solving strategies when encountering such variables. Apart from this, from a mathematical point of view, there is no difference between an integer variable bound to $[0, 1]$ and a binary variable which is explicitly specified as *binary* for the ILP solver. Subsequently, binary variables are also notated by lower-case roman letters in the upcoming sections. The describing text as well as the defined valid range defines whether a variable is "integer" or "binary".

The probably most famous algorithm used for solving ILPs is the *simplex* algorithm (also called the *simplex method*), propsed by Dantzig et al. [DOW55]. Although primarily used for solving linear problems with real-valued variables, the algorithm can also be applied to integer-linear problems by introducing additional slack variables which are bound to sufficiently small values. The ILP solver tool or the user must make sure that these slack variables' bounds are small enough to avoid numerical issues.

The simplex algorithm guarantees to find the global optimum of the underlying minimization or maximization problem after a finite number of optimization steps. Although simplex usually performs quite well, the number of steps needed grows exponentially in a worst-case scenario with the number of variables. As a countermeasure, Karmarkar [Kar84] proposed an alternative solution approach whose solving complexity depends only polynomially on the number of variables.

An ILP can have none, one or multiple optimal solutions. No solution exists in case of contradicting constraints (then, the ILP is said to be infeasible) or in case of unbounded variables (thus, the solution is infinite). In the case that exactly one optimal solution exists, this solution is returned by an ILP solver. If multiple optimal solutions exist leading to the same optimal value of the objective function, the ILP solver does not differentiate between these solutions. Although some solvers try to provide stable results, the user should expect any aribtrary solution from the set of optimal solutions to be returned.

5.2. Expressing Logical and Mathematical Relations

When using ILP constraints to model schedulability-aware optimizations, several logical relationships have to be modeled frequently. These range from relatively simple Boolean logic like a logical AND of two binary ILP variables, conditional constraints up to modulo calculations and bitwise operations.

This section gives an overview of all these formulations. In subsequent chapters, these logical expressions are used in equations in order to ease readability of formulations. Most of these formulations were used and (re)implemented in the past by different authors. Mostly, however, the concrete formulation is omitted in

publications. This work provides references to existing implementations when an existing implementation is known or will otherwise provide a proof of correctness. The formulations have been published in [LOF18] prior to submitting this thesis.

As a side note, modern ILP solvers like `Gurobi` or IBM `CPLEX` sometimes offer special expressions in order to directly model some logical expressions. However, tests showed that at least for the use case within this work, those special implementations were either slower or at least not faster than modeling the constraints manually within the ILP as regular inequations [LOF18]. Therefore, and to provide a solver-independent framework, this work will not use those so-called "general constraints".

5.2.1. Boolean Expressions

5.2.1.1. Boolean AND

Given three binary decision variables x, a and b, the goal is to model

$$x = a \wedge b \tag{5.7}$$

Proposition 5.1
The logical AND $x = a \wedge b$ *of two binary ILP variables* a *and* b *can be expressed as:*

$$x \geqslant a + b - 1 \tag{5.8}$$
$$x \leqslant a \tag{5.9}$$
$$x \leqslant b \tag{5.10}$$
$$a, b, x \in [0, 1] \tag{5.11}$$

Proof. We show that the proposition holds for all combinations of a and b.

$a = 0$, $b = 0$:

$$x \geqslant a + b - 1 \Rightarrow x \geqslant -1 \tag{5.12}$$
$$x \leqslant a \quad \Rightarrow \quad x \leqslant 0 \tag{5.13}$$
$$x \leqslant b \quad \Rightarrow \quad x \leqslant 0 \tag{5.14}$$

Because x is a binary variable, it is restricted to 0, thus the proposition holds for this case.

$a = 0$, $b = 1$:

$$x \geqslant a + b - 1 \Rightarrow x \geqslant 0 \tag{5.15}$$
$$x \leqslant a \quad \Rightarrow x \leqslant 1 \tag{5.16}$$
$$x \leqslant b \quad \Rightarrow x \leqslant 0 \tag{5.17}$$

x is forced to 0 by the last equation. Thus, the proposition holds for this case.

$a = 1$, $b = 0$: This case can be covered identically to the previous one.

$a = 1$, $b = 1$:

$$x \geqslant a + b - 1 \Rightarrow x \geqslant 1 \tag{5.18}$$
$$x \leqslant a \Rightarrow x \leqslant 1 \tag{5.19}$$
$$x \leqslant b \Rightarrow x \leqslant 1 \tag{5.20}$$

Eq. (5.18) forces x to 1, thus the proposition holds for this last case either. □

5.2.1.2. Boolean OR

Proposition 5.2
The logical OR $x = a \vee b$ of two binary ILP variables b and c can be expressed as:

$$x \geqslant a \tag{5.21}$$
$$x \geqslant b \tag{5.22}$$
$$x \leqslant a + b \tag{5.23}$$
$$a, b, x \in [0, 1] \tag{5.24}$$

Proof. For $a \equiv 1$, Eq. (5.21) enforces $x \equiv 1$. Respectively, for $b \equiv 1$, Eq. (5.22) enforces $x \equiv 1$, as well. For $a \equiv b \equiv 0$, Eq. (5.23) forces $x \leqslant 0$, thus x must be 0. In case that $a \equiv b \equiv 1$, x is forced to 1 by both Eqs. (5.21) and (5.22) and Eq. (5.23) is trivially fulfilled. □

5.2.1.3. Boolean XOR

Proposition 5.3
We define the logical XOR of two binary ILP variables a and b as follows:

$$x := a \oplus b \tag{5.25}$$

This can be expressed as follows:

$$x \geqslant a - b \tag{5.26}$$
$$x \geqslant b - a \tag{5.27}$$
$$x \leqslant a + b \tag{5.28}$$
$$x \leqslant 2 - a - b \tag{5.29}$$
$$x, a, b \in [0, 1] \tag{5.30}$$

Proof. For $a \equiv 0$ and $b \equiv 0$, Eq. (5.28) forces $x \leqslant 0$. Eqs. (5.26), (5.27) and (5.29) are trivially fulfilled.

For $a \equiv 1$ and $b \equiv 0$, Eq. (5.26) ($x \geqslant 1$) forces x to 1. Eqs. (5.27) to (5.29) do not contradict this. Thus, $x \equiv 1$. For $a \equiv 0$ and $b \equiv 1$, Eq. (5.27) forces $x \equiv 1$ accordingly.

Finally, for $a \equiv b \equiv 1$, Eq. (5.29) forces $x \equiv 0$: $x \leqslant 2 - 1 - 1 = 0$. Eqs. (5.26) to (5.28) do not contradict this. □

5.2.2. Conditional Expressions

5.2.2.1. Either-or Constraints

Consider the following two constraints:

$$\sum_i A_{1,i} \cdot a_i \leqslant B_1 \tag{5.31}$$

$$\sum_i A_{2,i} \cdot a_i \leqslant B_2 \tag{5.32}$$

$$A_{1,i}, A_{2,i}, B_1, B_2 \in \mathbb{R} \tag{5.33}$$

$$\forall i : a_i \in \mathbb{Z}, a_i \in [\check{A}_i, \hat{A}_i] \tag{5.34}$$

$A_{1,i}$, $A_{2,i}$, B_1 and B_2 are arbitrary constants. The a_i are integer variables. $\check{A}_i$ and $\hat{A}_i$ are arbitrary constants which bound the upper and lower limit of a_i.

The either-or constraint shall ensure that at least one of the constraints must hold. This way, one of both constraints may be violated without rendering the ILP infeasible. Note, that the either-or constraint is not modeling an exclusive XOR condition. Therefore, both constraints *may* hold but at least one of them *must* hold in order for the ILP to be feasible. This problem was described by Bisshop [Bis17] and the solution is taken from there:

Proposition 5.4
An either-or constraint between two given constraints may be achieved by introducing a binary decision variable y *and two sufficiently large constants* M_1 *and* M_2*:*

$$\sum_j A_{1,j} \cdot x_j \leqslant B_1 + M_1 \cdot y \tag{5.35}$$

$$\sum_j A_{2,j} \cdot x_j \leqslant B_2 + M_2 \cdot (1 - y) \tag{5.36}$$

$$M_1, M_2 \in \mathbb{Z} \tag{5.37}$$

M_1 and M_2 must be sufficiently large to ensure that each of the equations will always hold if y is 1 or 0, respectively.

Proof. The derivation and proof are given in [Bis17, pp. 77-78]. □

Valid bounds on M_1 and M_2 can be calculated by summing up the maximum values of the summation terms. This implies that upper bounds must exist for all x_j variables.

5.2.2.2. Conditionally Enabled Constraints

When logical conditions are described, it is rather common that a constraint only has to hold in certain cases. To express this, the following syntax is introduced:

$$b \equiv 0 \Rightarrow a_0 \cdot A_0 + a_1 \cdot A_1 + \cdots \leqslant B \tag{5.38}$$

$$b \in [0, 1] \tag{5.39}$$

$$\forall i : a_i \in \mathbb{Z} : a_i \in [\check{A}_i, \hat{A}_i] \tag{5.40}$$

$$A_i \in \mathbb{R} \tag{5.41}$$

This formulation denotes that the inequation only has to hold if the binary variable b equals 0. If b is 1, the constraint may be violated. This can be expressed using the following inequations:

$$a_0 \cdot A_0 + a_1 \cdot A_1 + \cdots - b \cdot M \leqslant B \tag{5.42}$$

$$b \in [0, 1] \tag{5.43}$$

$$\forall i : a_i \in \mathbb{Z} : a_i \in [\check{A}_i, \hat{A}_i] \tag{5.44}$$

As long as M is large enough, Eq. (5.42) will always hold if $b \equiv 1$. For $b \equiv 0$, $b \cdot M \equiv 0$, thus the constraint is not trivially fulfilled. A safe lower bound on M can easily be determined by reformulating Eq. (5.42):

$$M \geqslant B - \sum_i \check{A}_i \cdot A_i \tag{5.45}$$

This idea can be reused to formulate if-then dependencies between constraints. Consider the following two constraints:

$$\sum_i A_{1,i} \cdot a_i \leqslant B_1 \tag{5.46}$$

$$\sum_i A_{2,i} \cdot a_i \leqslant B_2 \tag{5.47}$$

$$A_{1,i}, A_{2,i}, B_1, B_2 \in \mathbb{R} \tag{5.48}$$

$$\forall i : a_i \in \mathbb{Z} : a_i \in \left[\check{A}_i, \hat{A}_i\right] \tag{5.49}$$

The if-then constraint shall ensure that if Eq. (5.46) holds, then Eq. (5.47) must hold, too. In the course of this thesis, this is depicted by:

$$\sum_i A_{1,i} \cdot a_i \leqslant B_1 \Rightarrow \sum_i A_{2,i} \cdot a_i \leqslant B_2 \tag{5.50}$$

Proposition 5.5
An if-then condition between two given constraints may be modeled by introducing an additional binary variable y *and two sufficiently large constants* M_1 *and* M_2*:*

$$\sum_i A_{1,i} \cdot a_i \geqslant B_1 + \epsilon - M_1 \cdot y \tag{5.51}$$

$$\sum_i A_{2,i} \cdot a_i \leqslant B_2 + (1 - y) \cdot M_2 \tag{5.52}$$

ϵ *is used to express a greater-than relationship using the greater-equal operator* $\geqslant$*. If all values are integer only,* ϵ *may be chosen as* 1*.* M_1 *and* M_2 *are sufficiently large constants.*

Proof. The derivation and proof are both given in [Bis17, pp. 79-80]. □

By reformulating both equations, safe bounds on M_1 and M_2 can easily be retrieved:

$$M_1 \geqslant -\sum_i A_{1,i} \cdot \check{A}_i + B_1 + \epsilon \tag{5.53}$$

$$M_2 \geqslant \sum_i A_{2,i} \cdot \hat{A}_i - B_2 \tag{5.54}$$

5.2.3. Minimum and Maximum

At several occasions, the minimum or maximum of two or more ILP integer variables is to be calculated:

$$x \coloneqq \max(a_0, a_1) \tag{5.55}$$

$$y \coloneqq \min(a_0, a_1) \tag{5.56}$$

$$a_i \in \left[\check{A}_i, \hat{A}_i\right], a_i \in \mathbb{Z} \tag{5.57}$$

$$x \in \left[\check{X}, \hat{X}\right], x \in \mathbb{Z} \tag{5.58}$$

$$y \in \left[\check{Y}, \hat{Y}\right], y \in \mathbb{Z} \tag{5.59}$$

$$\tag{5.60}$$

Proposition 5.6
The maximum $x = \max(a_0, a_1)$ *of two ILP integer variables* a_0 *and* a_1*, as well as the minimum* $y = \min(a_0, a_1)$ *can be expressed using the following constraints and variables:*

$$a_1 \leqslant a_0 + b \cdot M \tag{5.61}$$
$$a_0 \leqslant a_1 + (1 - b) \cdot M \tag{5.62}$$

$$b \equiv 1 \Rightarrow x = a_1 \tag{5.63}$$
$$b \equiv 1 \Rightarrow y = a_0 \tag{5.64}$$
$$b \equiv 0 \Rightarrow x = a_0 \tag{5.65}$$
$$b \equiv 0 \Rightarrow y = a_1 \tag{5.66}$$

$$b \in [0, 1] \tag{5.67}$$
$$Z \in \mathbb{Z} \tag{5.68}$$

with M *being a sufficiently large constant.*

The formulations above describe that the integer variable x holds the maximum of the integer variables a_0 and a_1 while the variable y models the minimum of both values.

Proof. If $a_0 < a_1$, the binary variable b in Eq. (5.61) is forced to 1. Subsequently, Eq. (5.63) sets $x = a_1$ and Eq. (5.64) forces $y = a_0$. Therefore, for $a_0 < a_1$, x models $\max(a_0, a_1)$ and y models $\min(a_0, a_1)$.

In case that $a_0 > a_1$, Eq. (5.61) does not restrict the binary variable b. However, Eq. (5.62) only holds, if $(1-b) = 1$. Therefore, b must equal 0. For $b \equiv 0$, Eq. (5.65) sets $x = a_0$ and Eq. (5.66) enforces $y = a_1$, accordingly. As a result, for $a_0 > a_1$ the proposed equations also model the max and min relationships correctly.

Finally, for $a_0 = a_1$, the ILP solver is free to choose b to either 1 or 0. As a result, x and y may be set to either a_0 or a_1. However, since both a_0 and a_1 are identical in this case, the proposition still holds. □

A safe bound on the sufficiently large constant M is calculated by taking the maximum distance between the minimum and maximum values of both a_i variables:

$$M \geqslant \max\left(\hat{A}_0 - \check{A}_1, \hat{A}_1 - \check{A}_0\right) \tag{5.69}$$

If the maximum or minimum over multiple integer variables is to be modeled, the max and min formulations introduced above may easily be chained:

$$\min(a_0, a_1, \ldots, a_n) = \min(a_0, \min(a_1, \min(\ldots, a_n))) \tag{5.70}$$

5.2.4. Bit-Wise Operations

Any integer number may also be interpreted as a bit vector. In some cases, it is useful to be able to retrieve a subset of this vector. To describe these bit-wise operations as a formula, we use a C++ like notation.

Example 5.1
Consider the 8 bit *wide number* $n = 52 = 00110100_2$.

The operation

$$i = (52 \;\&\; 00011100_2) \gg 2 \tag{5.71}$$

will return the bits 2-4 (starting at the right side of the number and counting the lowest bit as bit number 0) as new integer:

$$i = 52 \;\&\; 00011100_2 \gg 2 = 101_2 = 5 \tag{5.72}$$

This sequence of operations is, e.g., handy when calculating the cache index for a given memory address (cf. Fig. 3.5 on page 24). The solution to express this problem using ILP was developed as part of this thesis and has previously been published in [LKF16]:

Given a non-negative integer variable n and the maximum number of bits $\hat{N} = \lceil \log_2(n) \rceil$ needed to express the number as binary. Then, its base number decomposition to the base of 2 can be given as an ILP formulation by using one single equation:

$$n = 2^0 \cdot b_0 + 2^1 \cdot b_1 + \cdots + 2^{N-1} \cdot b_{N-1} \tag{5.73}$$

with $b_i \in [0, 1], b_i \in \mathbb{Z}\, \forall i \leqslant N - 1$. To illustrate this, e.g., the dyadic base factor decomposition of 52 may be written as:

$$52 = 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \tag{5.74}$$

It is obvious that there exists exactly one and unambiguous base factor decomposition of a given number for a given base as long as the b_i coefficients are limited to 0 and 1. Without loosing the unambiguity of the base factor decomposition,

subsequent summands can easily be combined into one term. Consider a shift by 2 bits and a bit-mask of $\mathtt{00011100}_2 = 28$ from the example above. This leads to:

$$52 = 2^0 \cdot o + 2^2 \cdot i + 2^5 \cdot t \tag{5.75}$$
$$0 \leqslant o \leqslant 2^2 - 1 = 3 \tag{5.76}$$
$$0 \leqslant i \leqslant 2^{5-2} - 1 = 7 \tag{5.77}$$
$$0 \leqslant t \leqslant 4 \tag{5.78}$$

Eq. (5.75) expresses Eq. (5.71) by introducing the three integer variables o, i, and t. o models the two lowest-order bits 0 in the given bit-mask. i models the three bits which are selected in the bit-mask. And finally, t expresses the three highest-order bits (which are also 0 in the bit-mask).

Providing upper bounds on o, i and t ensures that a low-order term cannot take values leading to a number which should only be expressible by a higher-order term in the base-number decomposition. Therefore, this inequation system has exactly one valid solution:

$$52 = 2^0 \cdot o + 2^2 \cdot i + 2^5 \cdot t \tag{5.79}$$
$$= 2^0 \cdot 0 + 2^2 \cdot 5 + 2^5 \cdot 1 \tag{5.80}$$

And the variable $i \equiv 5$ equals the result of Eq. (5.71):

$$(52 \;\&\; \mathtt{00011100}_2) \gg 2 = i = 5 \tag{5.81}$$

6. The WCET-Aware C Compiler

The WCET-Aware C Compiler (WCC) is a C cross-compiler framework written in C++ targeting the Infineon TriCore architecture and the ARMv4 instruction set based ARM7TDMI [FL10]. It currently features support for TriCore TC1796 and TC1797 microprocessors as well as the NXP LPC2880 ARM7TDMI microprocessor. Additionally, for scientific evaluations, an ARMv4 based multi-core system is supported which consists of a configurable amount of cores connected by a common bus.

First of all, WCC is a regular C compiler featuring well-known ACET optimizations like function inlining, loop unrolling, constant propagation, . . . With this respect, it is comparable to any other compiler framework like GCC or LLVM. Apart from this, WCC features several unique features targeting at the optimization of a system's worst-case runtime behavior. WCC seamlessly integrates the static WCET analyzer aiT into the compile flow. Moreover, it features an internal WCET analysis framework for the ARM7TDMI architecture, supporting multi-core setups with shared memories connected by a common bus [Kel15]. The analysis results are then integrated into the internal data structures of the WCC compiler framework, allowing for WCET-aware code optimizations which are completely transparent to the end user.

The optimization techniques presented in this thesis were integrated into this WCC compiler framework. As a result, WCC now allows for the automated schedulability-aware optimization of multi-tasking hard real-time systems.

The following sections give an overview of WCC's features as well as of its internal structure as far as related to this thesis. A sophisticated introduction into WCC is given in detail by, e.g., [FL10] and [LM11, p. 23ff]. An overview of recently added features and capabilities is given in [OLF18a]. Section 6.1 gives an overview of the internal structure of WCC. Section 6.2 continues by introducing the timing model used by WCC for its analyses and optimizations. In Section 6.3, WCC's internal WCET analysis framework is briefly discussed. Building upon the previous sections, Section 6.4 continues by presenting the integration of WCET oriented code optimizations into the compiler framework.

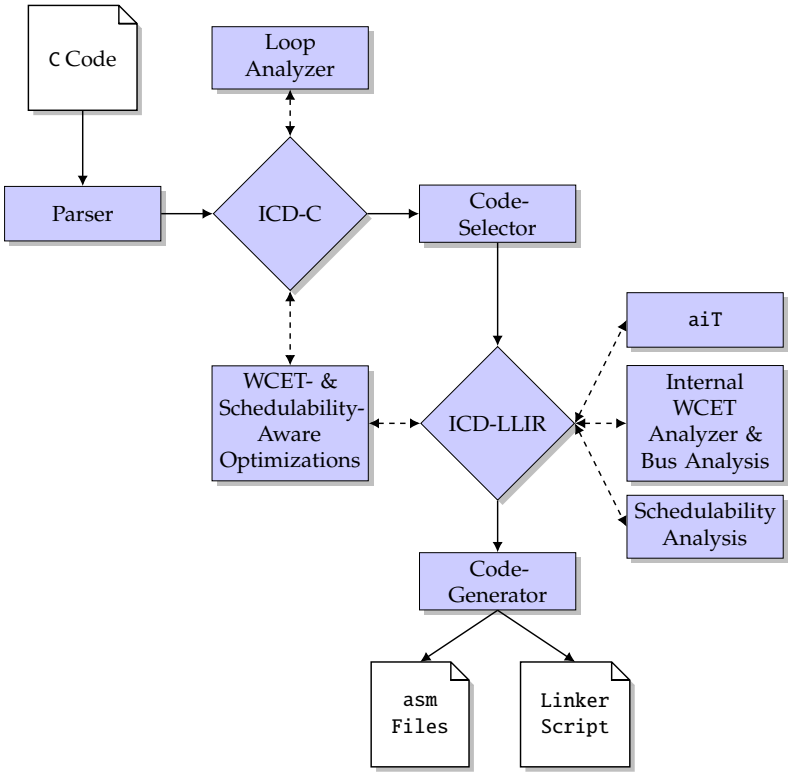


Figure 6.1: Structure of the WCC. Based on [FL10], updated. The solid lines denote the classical control flow of an optimizing compiler from C source code to a set of assembly files and a linker script. The dashed lines denote WCC-specific extensions.

6.1. Internal Structure

Fig. 6.1 shows an overview of the internal structure of WCC. The solid lines in the figure denote the control flow which is very similar to any modern optimizing compiler: A set of C files is input into the compiler and transformed into a high-level Intermediate Representation (IR) by a parser. WCC uses the commercial High-Level Intermediate Representation (HIR) *ICD-C* provided by Informatik Centrum Dortmund e. V. The high-level representation is then transformed into the low-level IR. In WCC, the low-level IR is called *ICD-LLIR* which is also provided by the Informatik Centrum Dortmund e. V.

Several ACET optimizations can be applied to both high-level and low-level IRs. They are not depicted explicitly but are integrated in the intermediate representations. Exemplary, WCC supports loop unrolling, constant propagation and folding, redundant code elimination, function elimination and many more traditional optimizations. These are very similar to the optimizations which

can be found in common compilers like GCC or LLVM. The low-level IR is then transformed into a set of assembly files and a linker script. For the Infineon TriCore architecture, WCC features its own code selector and register allocator. For the ARM7TDMI target, WCC uses GCC for code selection and register allocation. For both target architectures, the final assembly files are then assembled and linked using the GNU assembler gnu-as for the appropriate target.

Blocks connected by dashed lines in Fig. 6.1 show the unique parts of the WCC compiler framework which distinguish it from other compilers. A loop analyzer [Lok+09a] interfaces with ICD-C. It can be used to try to automatically bound the maximum number of loop iterations as discussed in Section 4.2.1.

The integrations of aiT and WCC's own internal WCET analyzer for multi-core platforms [Kel15] are interfacing with the ICD-LLIR. They are interfacing on the low-level IR, since WCET analysis has to be performed on machine-code-level in order to provide safe and tight WCET estimates for each basic block of a task. The results of the analyses are annotated to each basic block in the ICD-LLIR. An optional backannotation (which is not explicitly depicted in the figure) can be triggered in order to annotate the WCET timings back to the corresponding ICD-C IR expressions.

In the course of this thesis, a set of C++ classes was developed in order to perform schedulability analysis. These classes retrieve the WCET timing information for all tasks from the LLIR and use these in order to calculate whether the system is currently schedulable or not. The schedulability analysis can handle arbitrarily event-triggered systems with arbitrary deadlines. Both fixed-priority scheduling and EDF can be analyzed using the techniques previously presented in Section 4.5.

Finally, WCC features multiple WCET and schedulability-aware optimizations. Depending on the optimization, these may either operate on an IR-level (like, e.g., loop nest splitting [FS06]) or on the LLIR (e.g., WCET-aware register allocation [FSS11]).

6.2. Timing Model

In order for WCET-aware optimizations to be able to actually optimize towards the worst-case timing behavior of a task, WCC's intermediate representations need a notion on the timing behavior of the task. One of the distinct features of WCC therefore lies in its timing model which is integrated into the Low-Level Intermediate Representations (LLIRs). Every basic block in the LLIR can be annotated with arbitrary and detailed information regarding its timing properties. The results from supported WCET analysis tools are automatically parsed and annotated to the basic blocks. As a result, detailed information like the number of

Listing 6.1: Exemplary annotations for a system consisting of two tasks. The `entrypoint` keyword tells `WCC` that this function is the entry of a task. The additional keywords can be used to specify the task's timing requirements. CPU cycles are usually being used as time unit.

```
1
2  void _Pragma( "entrypoint period=1000 \
3                 deadline=100000 priority=0" ) task0 {
4    ...
5  }
6
7  void _Pragma( "entrypoint period=5000 \
8                 deadline=200000 priority=1" ) task1 {
9    ...
10 }
```

cache hits or misses, worst-case execution timing behavior for different execution contexts and the worst-case execution count are all annotated to each basic block. This information may then be used by the analyses and optimizations in `WCC` to specifically analyze or optimize the worst-case behavior of the complete system. In addition to these fine-grained annotations, `WCC` has a detailed knowledge on cache and memory sizes and memory access times for its supported platforms which may either be supplied by the user or be read from predefined configuration files.

System properties with an impact on the WCET or the system's scheduling behavior can be annotated by the user into the C code using `_Pragma` directives. For loop bounds and general flow restrictions, this has previously been discussed in Section 4.2.1. Additionally, the `_Pragma` directives can be used to annotate task deadlines, priorities, periods, and so on at source-level. An example is given in Listing 6.1. For annotations that specify timing requirements, the underlying time unit is basically arbitrary. If no time unit is given, the underlying time unit is considered as one CPU cycle, i.e., the inverse of the CPU's frequency.

Properties like, e.g., the scheduling algorithm which is to be used can be supplied to `WCC` by either a configuration file or command line options. Support of DMS, RMS, fixed-priority scheduling with user-defined priorities and EDF scheduling was integrated into `WCC` in the course of this thesis. Furthermore, for fixed-priority scheduling algorithms, `WCC` can automatically generate a minimal scheduler task for a more realistic evaluation of its schedulability-aware optimizations. This feature will be discussed in more detail in Section 11.5.

6.3. WCET Analysis Framework

For the ARM7TDMI platform, `WCC` features an internal WCET analyzer which was developed by Timon Kelter [Kel15] as part of his PhD thesis on the analysis of multi-core systems. It is able to analyze a task for its WCET and to precisely account for additional costs which are introduced due to accesses to a shared memory bus.

The analysis combines the bus analysis into the micro-architectural WCET analysis to allow for tight estimates of the bus overhead. Multiple tasks on each core are only handled rudimentary, such that each task is analyzed after each other, considering its WCET if executed stand-alone. For inter-core conflict analysis, additional timing penalties due to preemptions of tasks running on the same core are not considered. Additionally, inter-task conflicts on each core like CRPD are also not considered.

Apart from its own internal analyzer, `WCC` is tightly coupled to AbsInt `aiT` (cf. Section 2.1). `WCC` automatically generates all configuration files and annotations (like, e.g., recursion depths and loop bounds) in `aiT`'s configuration file format, calls `aiT`, and back-annotates `aiT`'s results to its own intermediate representations. This flow happens transparently for the user such that it allows for a seamless WCET analysis during compilation. Due to the back-annotation, `aiT`'s analysis results may be used for WCET and schedulability-oriented compiler based optimizations. For evaluations of the TriCore architecture, `WCC` was used with `aiT` version 18.10 in this thesis.

Neither the internal analyzer nor `aiT` are able to analyze a given multi-tasking system for its schedulability. Instead, they both solely analyze each task separately and return the WCET of each task if it is executed without any preemptions by other tasks. Therefore, the schedulability analyses proposed in Section 4.5 were integrated into `WCC` from scratch as part of this thesis. Additionally, a UCB/ECB CRPD analysis as proposed in Section 4.5.3 was implemented into `WCC` as part of this thesis. As a result, `WCC` now features a framework which can automatically analyze each task's stand-alone WCET, additional penalties due to preemptions, and include them all into a holistic schedulability analysis. The compiler can subsequently safely estimate whether all tasks in a given task set are schedulable or not.

In case that all tasks are schedulable, no further actions have to be taken from the real-time system designer's perspective. Otherwise, optimizations can be performed directly targeting at improving the worst-case timing behavior of the system's tasks.

6.4. WCET Optimizations

Prior to this thesis, WCC almost completely focused on a system setup with exactly one single task running on each processing unit. In such a scenario, optimizing towards schedulability is equivalent to minimizing *the* WCET of the a task. These optimizations have been previously discussed in Section 2.2.2.

This scenario used to be both sufficient and realistic in hard real-time setups. However, as it has been discussed throughout the previous chapters, modern real-time systems have shifted towards complex multi-tasking setups with preemptive schedulers. Only few works in WCC targeted at multi-tasking sytems prior to this work. Most notably, Plazar et al. proposed a cache partitioning optimization [PLM09]. In this optimization, the basic blocks of each task are allocated in such a way that the tasks cannot evict each other's basic blocks from the cache upon preemption. This circumvents CRPD effects at the cost of a smaller cache and more intra-task cache conflicts. Additionally, it does not account for the actual system schedulability. Instead, the problem was avoided by separating all tasks from each other, trying to mitigate any inter-task effects. This basically reduces the the multi-tasking optimization problem to multiple single-tasking optimizations. If no side effects exist between the tasks and the tasks do no longer compete for one common resource, due to, e.g., a previously applied cache partitioning scheme, then each task's WCET can be optimized separately without any notion of priorities, deadlines or scheduling algorithm.

While this simplifies the complexity of the problem, these approaches are by design not able to specifically optimize towards the system's schedulability. To counter these shortcomings, a schedulability-aware optimization framework was introduced into WCC as part of this thesis. This comprises an ILP-based optimization framework which allows for an optimal solution within the respective optimization model (cf. upcoming Chapter 8) as well as an approach based on a genetic algorithm (cf. upcoming Chapter 10).

7. ILP-based Model for WCET-Aware Single Tasking Optimizations

Trying to minimize a program's WCET within the compiler usually imposes a large combinatorial problem. Exemplary, assume an instruction SPM optimization where parts of a single-tasking program are to be assigned to a fast but small memory region in order to minimize the WCET. If the whole program fits into the SPM, the solution to the problem is obviously trivial. If not, the question arises which parts should be moved to the SPM and which should stay in the slow memory.

The complexity of the problem obviously depends on the granularity of the optimization. Moving whole functions may lead to a relatively small problem, however the granularity is so coarse that the limited resources of the embedded system are not used efficiently. Additionally, for a program with large functions, no single function might fit into the SPM completely. Thus, no improvement might be possible at all on a function-level. As a result, optimizations tend to use basic blocks as a minimum atomic entity [FK09]. However, this also significantly increases the complexity of the underlying combinatorial problem. For complex tasks, the number of basic blocks may easily get into hundreds.

A trivial idea for an optimization would be to use a greedy heuristic. The optimization would then successively move that basic block with the greatest timing gain to the SPM, until the SPM is full. Such optimizations have been demonstrated successfully, e.g., for basic block reordering [FK11]. However, apart from mitigating the complexity of the combinatorial problem, such heuristic approaches come with several drawbacks. First of all, they do not guarantee an *optimal* solution. Moving one block with the biggest improvement might have a smaller improvement than, e.g., moving two blocks which have a smaller improvement each, but a larger improvement in total. Additionally, these heuristics are *iterative*. Due to the fact that moving one basic block might have unforeseen side effects on the rest of the program, a full-fledged time intensive WCET analysis has to be performed after each iteration. The reason for this is that the WCEP may change unpredictably after performing a local modification. Example 7.1 illustrates this phenomenon.

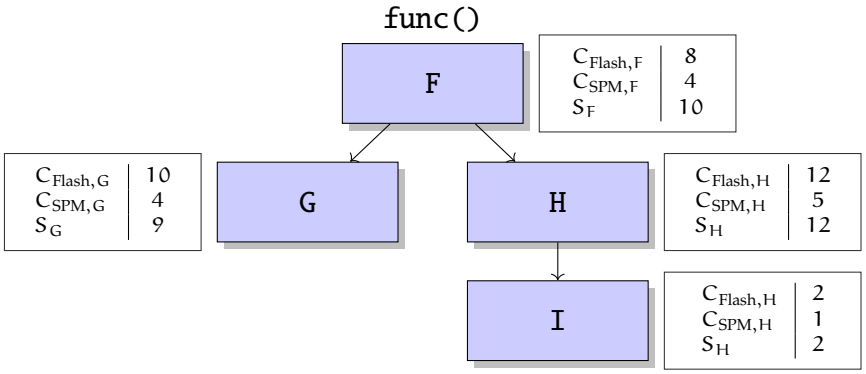


Figure 7.1: Exemplary CFG of a very simplistic function annotated with execution times for each basic block. The timings express the execution times C for each basic block if it is either allocated to SPM or to the slow Flash memory, as well as the size S of each basic block. The units may be chosen arbitrarily.

Example 7.1 (Changes in WCEP)
Fig. 7.1 shows the CFG of a very minimalistic function `func()`*. As an example, consider a static SPM allocation as WCET optimization: Each basic block's worst-case execution time is denoted by* C^{Flash}*, if left in main memory, and* C^{SPM}*, if assigned to the SPM. Timings are given in an arbitrary time unit. For this example, consider a total SPM size of* 20 *units.*

Consider the whole program is located in main memory. Then, the program's WCEP is $F \to H \to I$ *and the corresponding WCET is* 22*.*

The maximum gain for one single block being moved into SPM will be achieved by moving block H*. Thus, the execution time of path* $F \to H \to I$ *is reduced to* 15*. However, now the WCEP switches, as the longest path through the function is now* $F \to G$*, resulting in a WCET of* 18*. Thus, the WCET of the function is* 18*. Due to the SPM size of* 20*, only block* I *could still be moved into SPM, which, however, will not improve on the WCET.*

Instead, moving F *and* I *into the SPM would result in a WCET of only* 17 *time units.*

It can be seen that even for a very minimalistic function consisting of only 4 basic blocks with one branch and no loops or function calls, finding an optimal solution is not that obvious. Suhendra et al. decided to use Integer-Linear Programming (ILP) to formulate the optimization problem for a data SPM allocation [Suh+05]. However, the underlying approach is much more flexible and can be applied to many different optimization problems. The idea was picked up later by Falk et al. and used to model a WCET-aware instruction SPM allocation [FK09] as well as a WCET-aware register allocation [FSS11]. In the course of this thesis it has

also been shown that the ILP formulation is able to handle modern embedded systems featuring instruction caches [LKF16].

As a basis for the upcoming contributions presented in this work, this chapter first gives a brief introduction into the underlying ILP model by Suhendra et al. [Suh+05] and Falk et al. [FK09] without focusing on any concrete WCET optimization technique. The second part of the chapter focuses on modeling the previously mentioned static instruction SPM allocation which proves to be a powerful example for the upcoming schedulability-aware optimization framework. This chapter will leave out any multi-tasking and schedulability details. These are then handled in the upcoming Chapter 8.

7.1. ILP-Based WCET Modeling of a Single Task

This section introduces an ILP formulation to optimize the WCET of one single task. The model was initially proposed by Suhendra et al. [Suh+05] and used for a static data SPM allocation. The model was subsequently adapted for a single-tasking static instruction SPM allocation by Falk et al. [FK09].

The model operates at a basic block-level on the CFG of a program. In contrast to the IPET model previously presented in Section 4.2.3, this approach formulates the WCET calculation as a *minimization* problem. Therefore, it allows for an easy integration of code optimization techniques.

This section covers the pure ILP model which constrains the WCET of a single task. It does not yet cover any concrete optimization techniques. Adding ILP variables and constraints to perform an actual optimization will be discussed in the upcoming sections.

7.1.1. Sequential Code

Prior to modeling the WCET using the ILP formulation, the timing behavior of each individual basic block has to be evaluated. To provide a consistent model for different optimizations, the ILP integer variable c_i is introduced which is defined to model the WCET of one single execution of a distinct basic block i. In the simplest case, a static WCET analysis can be used to retrieve the constant worst-case execution time C_i for each basic block i. Subsequently, the ILP variables can be set as:

$$\forall i \in \mathcal{B} : c_i = C_i \tag{7.1}$$

where $\mathcal{B}$ denotes the set containing all basic blocks. In this case, the ILP framework will not provide any optimization but is basically "degraded" to a

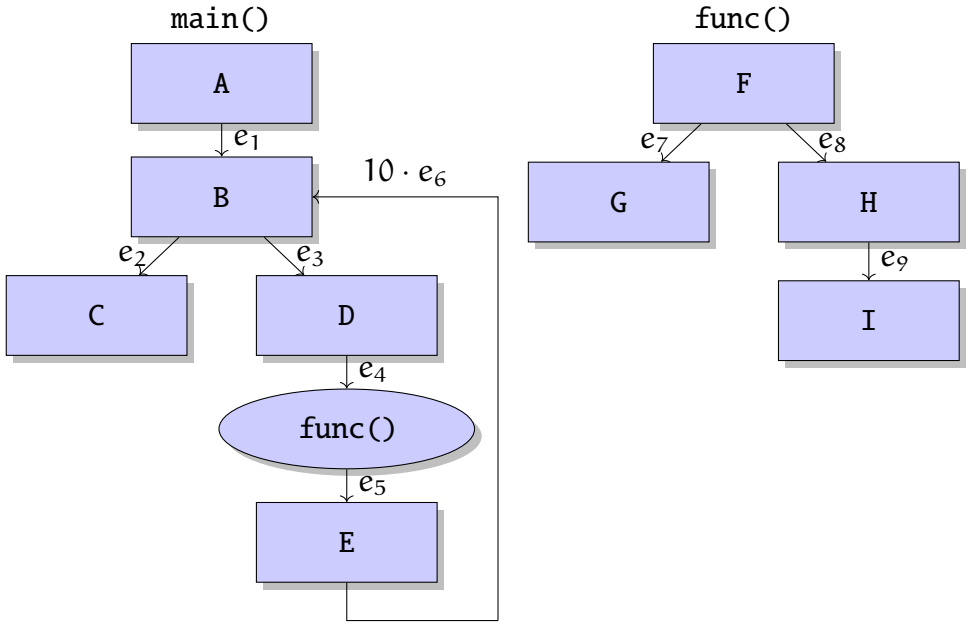


Figure 7.2: CFG of an exemplary task. Each box represents a basic block.

WCET analysis. For optimizations, arbitrary additional constraints can be used in order to model the c_i variables. This way, the WCET of a basic block can differ depending on whether it is optimized or not.

The general idea of the ILP framework is explained using the exemplary CFG depicted in Fig. 7.2. The minimal program represented by this graph consists of two functions: `main()` and `func()`. The `main()` function features one loop with 10 iterations.

The ILP model is built on a function-level, starting at the exit nodes of the CFG, implicitly accumulating the WCET of the whole program starting at each basic block. If a basic block has no successor, the accumulated execution time w of that basic block is defined as the execution time of one single execution of that block. E.g., for the basic blocks G and I of function `func()` shown in Fig. 7.2, this results in:

$$w_G = c_G \tag{7.2}$$
$$w_I = c_I \tag{7.3}$$

For basic blocks which have one successor, the accumulated execution time is defined as the time for one execution of the block itself plus the accumulated execution time of its succeeding basic block:

$$w_H = c_H + w_I \tag{7.4}$$

If multiple successors exist, separate constraints are created for each possible successor. For `func()`, this leads to:

$$w_F \geqslant c_F + w_G \tag{7.5}$$
$$w_F \geqslant c_F + w_H \tag{7.6}$$

Due to the greater-equal operator, w_F is forced to be as large as defined by that succeeding basic block which leads to a larger execution time. This implicit formulation basically forms the key concept why this ILP formulation allows for modeling the WCET of the given function.

It should be noted that, wherever an optimization which is attached to the model modifies the execution times, any changes in the WCEP are inherently reflected in the model. This stems from the fact that valid paths through the task's CFG and their respective execution times are not listed explicitly, but are implicitly modeled by inherently accumulating the costs of each of the block's successors. This way, the possibly complex switches of the WCEP are automatically handled within an optimization.

The equations presented above are sufficient to model the very simple function `func()`. Solely for the sake of better readability, we add another constraint which bounds the maximum execution time of one single execution of `func()`:

$$c_{\text{func}} = w_F \tag{7.7}$$

7.1.2. Function Calls

When looking at the successor of block `D` in Fig. 7.2, it can be seen that it calls function `func()`. Since the function's maximum execution time has already been modeled, the ILP variable modeling the maximum execution time of one call to function `func()` can simply be added to the accumulated execution time of the calling basic block:

$$w_D = c_D + c_{\text{func}} + w_E \tag{7.8}$$
$$\tag{7.9}$$

7.1.3. Loops

Loops are handled by converting them into meta blocks. This is shown in Fig. 7.3. All basic blocks which form the loop are included in that meta block, and the back-edge which originally formed the loop is removed. Instead, the execution of that loop's meta block is annotated by the number of taken loop iterations.

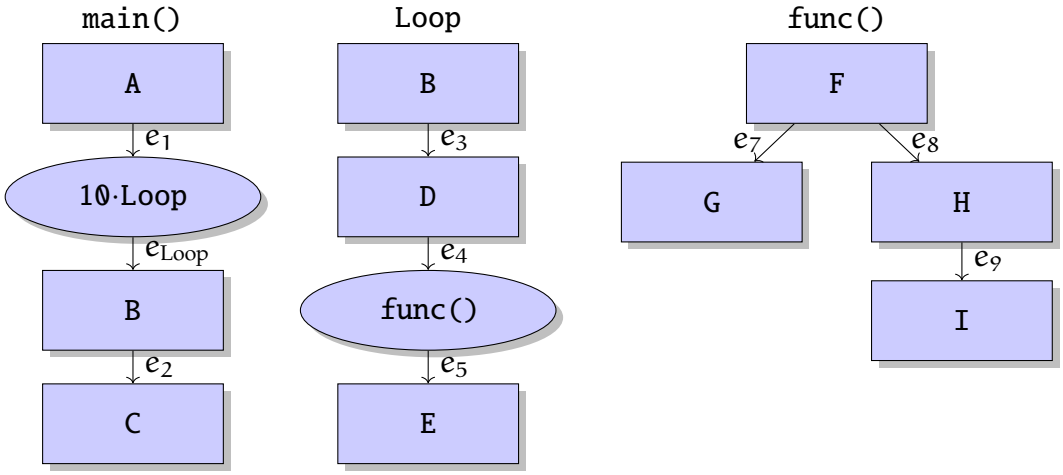


Figure 7.3: CFG for exemplary code with loop meta block.

The net execution time of one single execution of the loop is described by c_{Loop}. The accumulated execution time w_{Loop} models the complete execution time over all iterations of the loop plus the accumulated execution time of the loop's successor. Analogously to the previous description for modeling sequential code, c_{Loop} now results in:

$$c_{Loop} = c_B + w_D \tag{7.10}$$
$$w_D = c_D + c_{func} + w_E \tag{7.11}$$
$$w_E = c_E \tag{7.12}$$

After modeling the WCET of one single loop iteration, the accumulated execution time w_{Loop} can subsequently be described as follows:

$$w_{Loop} = 10 \cdot c_{Loop} + c_B + w_C \tag{7.13}$$

Due to the fact that the loop depicted in Fig. 7.2 is head-controlled (i.e., the loop condition is checked prior to entering the loop), basic block B is executed 11 times. In the final execution of B, the loop body starting at block D is not entered but the loop is exited through block C. This leads to the additional term $+c_B$ in Eq. (7.13). For tail-controlled loops (e.g., a **do ... while**() loop in C), the loop is always executed at least one time and the head is evaluated at the end of the loop iteration. Therefore, the head is executed as often as the loop body, and

this additional accounting is not necessary. With these constraints, the overall execution time of the `main()` function with the loop can now be modeled:

$$c_{\text{main}} = w_A \tag{7.14}$$

$$w_A = c_A + w_{\text{Loop}} \tag{7.15}$$

$$w_C = c_C \tag{7.16}$$

In some cases, a loop header may consist of more than one basic block. This happens, e.g., in case of multiple conditional expressions within the loop conditions. In these cases, the loop head (`B` in the given example) is not only one single basic block but forms yet another meta block. The challenge when accounting for such complex loop heads is, that on the low-level abstraction on which the ILP is built on, it is impossible to differentiate between a loop head which consists of multiple possible exits (e.g., **`while`**`( a && b ){ ... }`) or a **`break`** statement which is executed conditionally within the loop's *body* (i.e., **`while`**`( a ) { `**`if`**`( ! b ) `**`break`**` ...}`. However, from a timing analysis point of view, both statements must be accounted for in the final execution of the loop head if they are in the loop head, whereas in the latter case the test whether `b` is not zero does not have to be accounted for.

Due to the similarity of both constructs on the low-level CFG, these constructs were previously not accounted for. However, due to the implementation as part of the `WCC` compiler, it is possible to reliably trace back a low-level basic block to its corresponding C statement by using `WCC`'s internal data structures. Subsequently this improvement was implemented newly when creating the ILP framework for this thesis.

Note, that commerical state of the art WCET analyzers like `aiT` are *not* able to reliably determine these constructs. In order to provide safe upper estimates, the too pessimistic worst-case is assumed, and both statements are always checked. This may lead to severe over-estimation of the WCET in case of conditional **`break`** statements within nested loops. Despite `aiT`'s precision, for some benchmarks this can lead to cases where the ILP-based optimization gives a tighter yet safe estimate on the WCET than `aiT`.

The previous constraints modeled so-called *natural* loops, which are characterized by having one single loop entry. For arbitrary loops with multiple entries, the loop meta block can be split into multiple smaller meta blocks to cover the first and only partly executed loop body separately. Then, the accumulated execution time for complete loop iterations for the remaining number of iterations of the loop can be added to each of these partially executed loop bodys. However, these so-called non-natural loops with multiple entries are very rare in the domain of hard real-time systems, so they will not be considered in the following for the sake of a simpler notation.

Listing 7.1: Triangular nested loop with simple loopbound annotations. The execution count of the inner loop depends on the counting variable of the outer loop. This is not explicitly restricted by the loopbounds.

```
1  int main() {
2
3    int i;
4    int j;
5    int res = 0;
6
7    _Pragma("loopbound min 10 max 10")
8    for( i = 0; i != 10; ++i ) {
9
10     _Pragma("loopbound min 1 max 10")
11     for( j = i; j != 10; ++j ) {
12       ++res;
13     }
14   }
15   return res;
16 }
```

Nested Loops

In case that loops are nested, they can be converted iteratively as described above, starting with the innermost loop. This way, nested loops can be modeled without any further adjustments of the model. The total number of maximum executions of the inner loop subsequently resolves to a multiplication of the maximum loopbound of the outer loop and the maximum loopbound of the inner loop. This approach may drastically over-approximate the maximum execution count of nested loops in case that the Worst-Case Execution Count (WCEC) of the inner loop depends on the counting variable of the outer loop. Such an example is depicted in Listing 7.1. The corresponding CFG is shown in Fig. 7.4.

It is easy to see that the innermost loop starts with 10 loop iterations during its first execution. In the last execution of the outer loop, the inner loop is only executed once.

Fig. 7.5 shows the control flow with the transformed loops as previously described for non-nested loops. If this loop were modeled by the approach shown above, the maximum loop bound must be annotated to the accumulated execution time variables of both inner and outer loops. The ILP formulation would subsequently be as follows:

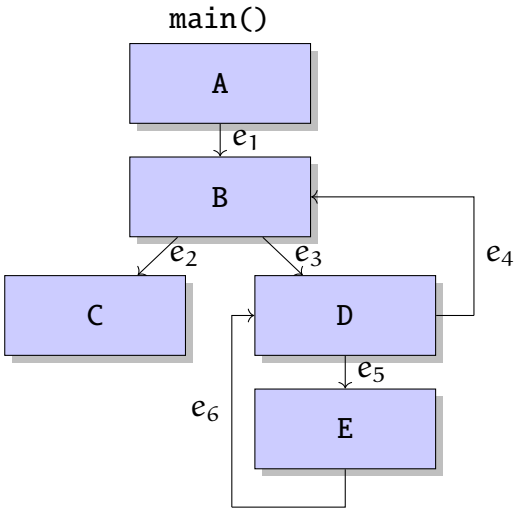


Figure 7.4: CFG for Listing 7.1, depicting a nested loop.

$$c_{\text{main}} = w_A \tag{7.17}$$
$$w_A = w_{\text{Loop0}} + c_B + w_C \tag{7.18}$$
$$w_C = c_C \tag{7.19}$$
$$w_{\text{Loop0}} \geqslant 10 \cdot c_{\text{Loop0}} \tag{7.20}$$
$$c_{\text{LoopO}} \geqslant c_B + w_{\text{LoopI}} + w_D \tag{7.21}$$
$$w_D = c_D \tag{7.22}$$
$$w_{\text{LoopI}} \geqslant 10 \cdot c_{\text{LoopI}} \tag{7.23}$$
$$c_{\text{LoopI}} \geqslant c_D + w_E \tag{7.24}$$
$$w_E = c_E \tag{7.25}$$

This, however, leads to a significant over-approximation of the WCET as now, the ILP model will account for 100 executions of the inner loop, although the actual number of executions is obviously only 55.

The `WCC` compiler framework in which this optimization model is embedded, features so-called "flow restrictions" which allow for a tighter model of complex execution behavior (cf. Section 4.2.1). Listing 7.2 shows the nested loop from Listing 7.1, but this time with additional flow restrictions which explicitly model the triangular structure of the inner loop's execution count.

The expressions `inner` and `outer` define labels which can be referenced in the flow restrictions. The expression restricts the number of executions of the statement in the inner loop (marked by `inner`) to be at most 55 times the number

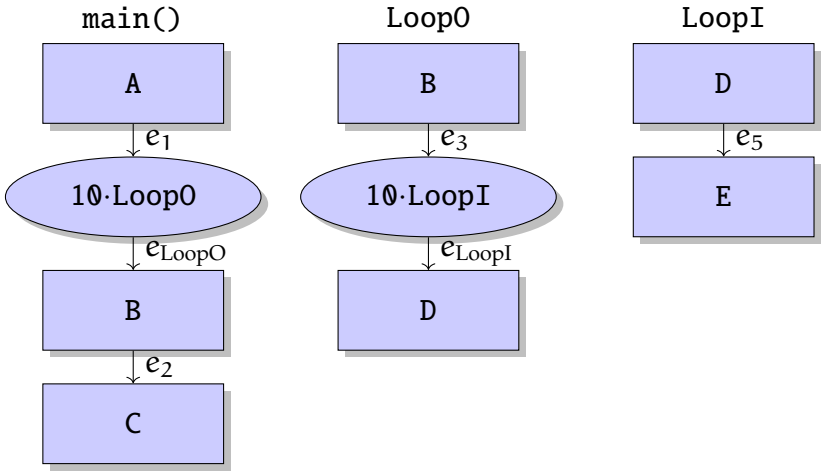


Figure 7.5: CFG for Listing 7.1, depicting a nested loop using meta blocks.

of executions of the outer statement (marked by `outer`). In this case, the position marked by the `outer` marker is executed only once, as it is not part of a loop. Therefore, the statement marked by the inner loop's marker and subsequently the inner loop must be executed at most 55 times in total.

Using flow restrictions to tighten the execution count of nested loops has not been covered in previous approaches like [Suh+05] or [FK09] and has been newly realized as part of this thesis. To tackle nested loops, the maximum iteration count of the inner loop, as modeled by Eq. (7.23), is adapted to reflect the actual upper execution count:

$$w_{\text{LoopI}} \leqslant \frac{55}{10} \cdot c_{\text{LoopI}} \tag{7.26}$$

The maximum iteration count of the inner loop for one execution of the outer loop is modeled as the maximum iteration count of the inner loop divided by the maximum iteration count of the outer loop. This may lead to fractional execution counts for the inner loop. Consequently, this introduces some pessimism due to rounding errors: As w_{LoopI} is integer, its integer value may be up to one time unit larger than its actual real value. However, the approach still provides much tighter WCET estimates than simply multiplying the worst-case execution counts for both outer and inner loops.

Although nested loops and flow restrictions can become very complex, code structures within embedded hard real-time systems are mostly relatively simple in practice. In this work, we therefore only explicitly consider these relatively simple versions of nested loops with dependent loop conditions. For more complex loops which cannot be expressed in this model, the initially sketched

Listing 7.2: Triangular nested loop with flow restrictions. The execution count of the inner loop depends on the counting variable of the outer loop. This is explicitly modeled by the annotated flow restrictions.

```
1  int main() {
2
3    int i;
4    int j;
5    int res = 0;
6
7    _Pragma("loopbound min 10 max 10")
8    for( i = 0; i != 10; ++i ) {
9
10     for( j = i; j != 10; ++j ) {
11       _Pragma( "marker inner" );
12       ++res;
13     }
14   }
15
16   _Pragma( "marker outer" );
17   _Pragma( "flowrestriction 1 * inner <= 55 * outer" );
18   return res;
19 }
```

over-approximation of the WCET by transforming all loops to meta-nodes and using their respective upper loop bound as multiplication factor is used.

7.1.4. Recursive Function Calls

Previous approaches ([Suh+05] and [FK09]) did also not support recursive function calls. Programs containing recursions could not be optimized by these formulations. Recursive function calls are usually classified as *direct* or *indirect* recursions. In case of a direct recursion, a function directly calls itself repeatedly. In case of an indirect recursion, two (or more) functions call each other in a nested fashion.

Generally, recursions are quite uncommon in the domain of safety-critical embedded systems. However, some commonly found benchmarks make use of direct recursion. Therefore, support for direct recursion was added to the ILP framework as part of this thesis.

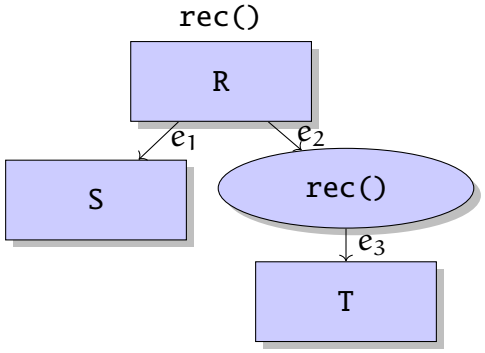


Figure 7.6: CFG depicting a simple directly recursive function.

Fig. 7.6 shows the CFG of a simple directly recursive function `rec()`. From the CFG itself, it cannot be deduced how many times `rec()` may call itself in a worst-case scenario. This information has to be provided additionally by the system designer by the means of, e.g., flow facts. For this example, it is assumed that `rec()` may call itself at most 10 times. Considering the recursion as a regular function call will lead to an infeasible ILP as it would lead to equations

$$c_{rec} = w_R \tag{7.27}$$
$$w_R \geqslant c_R + c_{rec} + w_T \tag{7.28}$$

The only way that these inequations hold is that both c_R and w_T equal 0. However, in this case, the recursion would not even exist in practice, since this would imply that the whole recursive function (c_R) would be executed in zero time units. For any real existing basic blocks, however, w_R can obviously not be greater than or equal to itself plus another positive integer, thus the inequation contradicts itself.

To tackle this issue, the recursive call to `rec()` is neglected when calculating the accumulated cost w_R of basic block R, and then the costs due to multiple nested calls are accounted for when modeling c_{rec}. This leads to the following inequations:

$$c_{rec} \geqslant 11 \cdot w_R \tag{7.29}$$
$$w_R \geqslant c_R + w_S \tag{7.30}$$
$$w_R \geqslant c_R + w_T \tag{7.31}$$
$$w_S = c_S \tag{7.32}$$
$$w_T = c_T \tag{7.33}$$

Eq. (7.29) accounts for all 11 executions of `rec()` (the direct call, plus the maximum of 10 recursive calls as annotated by the user). As a result, the calls to

the recursive function `rec()` in an arbitrary program can be modeled like a call to any non-recursive function. Indirect recursive function calls can be modeled accordingly, if needed.

7.2. ILP-based SPM Allocation for a Single Task

Memory-based optimizations have proven to provide significant impact when it comes to the reduction of the WCET of a task [FK09]. Additionally, as shown in the introduction of this chapter, SPM allocation is combinatorially complex which comes in handy for the evaluation of practical usability of the upcoming schedulability-aware optimization framework.

This section will describe the general idea of the single-tasking ILP-based static instruction SPM allocation as previously proposed by Falk et al. [FK09]. The approach was originally presented for the Infineon TriCore architecture but is platform-independent per se. In this thesis, the optimization is applied to both TriCore and ARM7TDMI architectures.

The optimizations and techniques described in this section purely focus on the minimization of the WCET of one single task. It does neither account for multiple tasks nor scheduling effects. Additionally, caches are not considered but the approach assumes that the unoptimized program resides in non-cached Flash memory. The optimization builds on the underlying framework presented in Section 7.1. To focus on the optimization specifics, the underlying idea will therefore be explained using the very simple CFG depicted in Fig. 7.2. Additional tweaks and technical implications both covering the TriCore architecture and systems based on ARM7TDMI, which were not covered in the original publication by Falk et al., are covered in Section 7.3.

To be able to perform an SPM allocation optimization which specifically targets improving the WCET, for each part of the task which can either be moved to the SPM or not, the possible timing gain must be known. Therefore, the complete task will be analyzed twice using an existing WCET analyzer like, e.g., `aiT` by AbsInt. The first analysis is performed when the task completely resides in the slow main memory (e.g., a Flash memory). Then, the WCET of each basic block is retrieved from the analysis tool and stored. We denote the net execution time of one execution of a basic block i which is located in Flash memory as C_i^{Flash}.

The second analysis is performed when the task completely resides in SPM. Obviously, normally the SPM will be too small to hold the whole task. Otherwise, the optimization can be trivially performed without any ILP model by simply assigning all basic blocks to the SPM. However, for the initial analysis only, the SPM memory can be virtually increased such that the whole program fits into

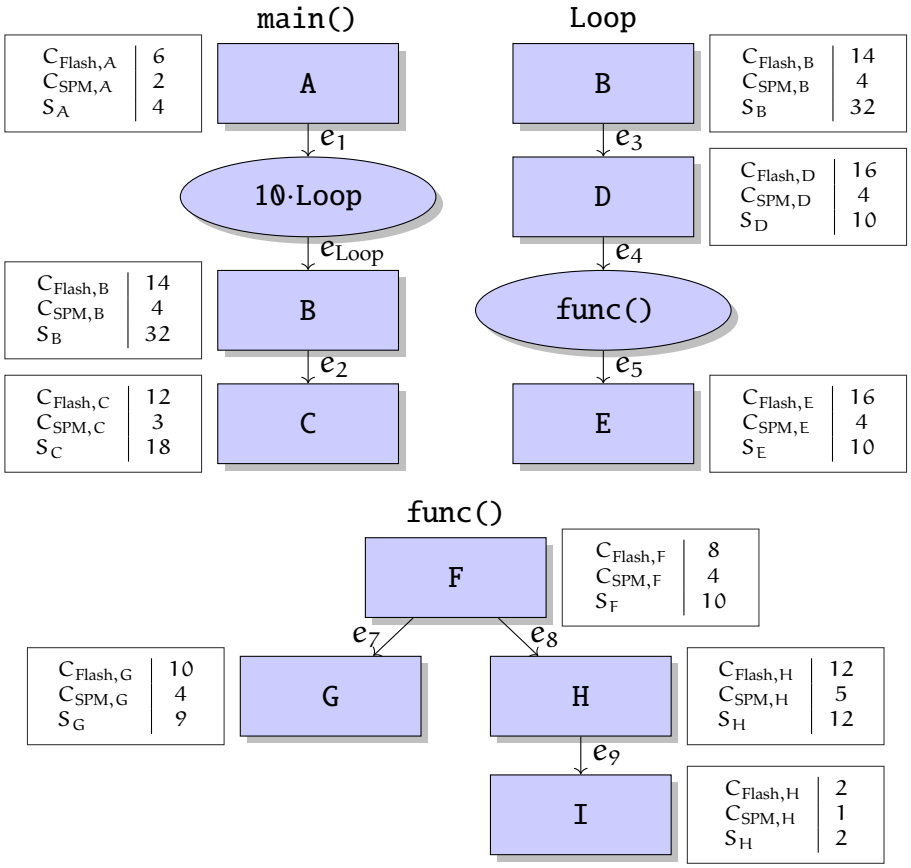


Figure 7.7: CFG of an exemplary task. Each box represents a basic block. The worst-case execution times for one single execution of each block in case of both SPM and Flash assignment are annotated. Also, the size of each basic block is annotated. Units are arbitrary.

it. At all later stages in the optimization and allocation, the correct sizes for the SPM memory will be used. After this second analysis, the maximum execution time of each individual execution of each basic block if residing in the SPM can be retrieved. For a given basic block i, it is denoted by C_i^{SPM}.

Fig. 7.7 shows the CFG from Fig. 7.3 with annotated WCETs and sizes for each basic block. Given the execution times for execution on SPM and Flash memory, a possible gain G_i can be calculated. This represents the timing gain achieved for one execution of one distinct basic block i, if this block is allocated to the SPM:

$$G_i = C_i^{Flash} - C_i^{SPM} \tag{7.34}$$

The integer variable c_i which constrains the net execution time of any given basic block i is extended as follows:

$$c_i = C_i^{\text{Flash}} - x_i \cdot G_i \tag{7.35}$$

x_i is a binary decision variable which is set to 1 by the ILP solver to denote that basic block i is to be assigned to the SPM or 0 in case that the basic block should stay in Flash memory. For the given example, this leads to:

$$c_A = C_A^{\text{Flash}} - x_A \cdot G_A = 6 - 4 \cdot x_A \tag{7.36}$$
$$c_B = C_B^{\text{Flash}} - x_B \cdot G_B = 14 - 10 \cdot x_B \tag{7.37}$$
$$c_C = C_C^{\text{Flash}} - x_C \cdot G_C = 12 - 9 \cdot x_C \tag{7.38}$$
$$c_D = C_D^{\text{Flash}} - x_D \cdot G_D = 16 - 12 \cdot x_D \tag{7.39}$$
$$c_E = C_E^{\text{Flash}} - x_E \cdot G_E = 16 - 12 \cdot x_E \tag{7.40}$$
$$c_F = C_F^{\text{Flash}} - x_F \cdot G_F = 8 - 4 \cdot x_E \tag{7.41}$$
$$c_G = C_G^{\text{Flash}} - x_G \cdot G_G = 10 - 6 \cdot x_E \tag{7.42}$$
$$c_H = C_H^{\text{Flash}} - x_H \cdot G_H = 12 - 7 \cdot x_E \tag{7.43}$$
$$c_I = C_I^{\text{Flash}} - x_I \cdot G_I = 2 - 1 \cdot x_E \tag{7.44}$$

Given these formulations, the ILP model from Section 7.1 now models the possible timing gains due to the SPM allocation optimization. To finalize the ILP, an objective function is set in order to minimize the accumulated WCET of the `main()` function:

$$\min\left(c_{\text{main}}\right) \tag{7.45}$$

Given only these constraints and the objective, the ILP solver will allocate those basic blocks to the SPM that lead to the minimum overall WCET. However, the solver does not yet honor the actual size of the SPM, thus more basic blocks may be moved to SPM than actually fit into it. To tackle this issue, the constant size of the SPM is denoted as S_{SPM}. This is considered to be a hardware constant determined by the chosen target platform.

In addition, each basic block's size is known to the optimization, as it is embedded into a compiler framework. We denote the size of a given basic block i by S_i. For the example used throughout this section, sizes for each basic block are given in Fig. 7.7.

Now, one single additional constraint can be used to limit the number of basic blocks which may be allocated to the SPM in correspondence with the physical size of the SPM S_{SPM}:

$$S_{\text{SPM}} \geqslant \sum_{i \in \mathcal{B}} x_i \cdot S_i \tag{7.46}$$

When exemplary assuming an overall SPM size of $S_{SPM} = 20\,B$, the example used in this section resolves to:

$$20 \geqslant 2 \cdot x_A + 32x_B + 18 \cdot x_C + \cdots + 2 \cdot x_I \tag{7.47}$$

For basic blocks whose memory allocation will not result in a change of the WCET, the ILP solver is free to choose whether the allocation variable x is set to 1 or 0. From a user perspective, this should be considered arbitrary. If SPM memory should be used as much or as little as possible, many ILP solvers feature secondary objectives which could then maximize or minimize the used SPM. However, since this work solely focuses on optimizing schedulability of hard real-time systems and does not consider any secondary objectives, this is not considered in the following.

These inequations already provide a basic SPM allocation, which, however, is not yet safe. The problem is that allocating one basic block to the SPM while its predecessor(s) and/or successor(s) stay in Flash memory, will invalidate the control flow at the assembly-level, as a basic block's previously implicit successor in memory will then no longer be the implicit successor afterwards. To retain a valid functional behavior, additional jump instructions must be inserted. These, however, come at both timing and size costs which are not yet modeled. The concrete modeling of these additional costs differs depending on the actual target architecture for which the optimization is carried out. These additional costs and further architecture-dependent tweaks are discussed in the next section.

7.3. Technical Implications

The previous section gave a basic overview of the general idea behind ILP-based WCET optimizations and static instruction SPM allocation as one powerful example. For the sake of simplicity, several significant side effects were neglected when describing the basic SPM optimization in Section 7.2. This section will discuss the necessary extensions for the SPM optimization and introduce the modified (in)equations.

7.3.1. Jump Correction

The by far most important addition is the accounting for jump correction costs. This is mandatory for a correct optimization and was implemented originally by Falk et al. [FK09] for the TriCore target architecture. Later on, Oehlert et al. described and implemented the jump correction for the ARM7TDMI architecture [OLF16]. This section will therefore only give a brief overview of the necessary techniques.

When re-allocating a basic block to the SPM, the basic block's memory address obviously changes. For example, consider basic block F from function `func()` depicted in the CFG in Fig. 7.7. Consider that this block F is moved to the SPM, while its succeeding block H stays in Flash memory,

This leads to the following issue: Block F has two successors: G and H. Let us assume that G used to be the so-called *implicit* successor of F, i.e., it follows F directly in memory. Subsequently, there was no need for an explicit assembly instruction jumping from the end of block F to the beginning of block G. Accordingly, block H is called the *explicit* successor. This means that the very last instruction of F contains an assembly instruction which, depending on some condition, will perform an explicit jump in order to reach block H.

After moving F to Flash memory, block G will occupy the original position of F. F may or may not have a physical successor in the SPM, but it will not be its logical successor G. As a result, without any further modifications, the control flow of the program will be changed and the program will be functionally broken. Therefore, two issues must be resolved: First, an explicit jump must be added after the last instruction of block F, correcting the control flow to the previously implicit successor G.

Second, the address of the explicit jump target in the final instruction of F must be updated to point to the new location of block H. In theory, at least this second issue could be solved without adding any further assembly instructions. However, in practice the address space of the SPM is far off the address range of the Flash memory. As a result, the relative displacement of H compared to the new location of F is so large that it does not fit into the existing assembly instruction. To handle the large displacement between different memories, the jump target's address is usually loaded into a register, and then a special assembly instruction is used to jump to the target address denoted in that register. If at that position within the control flow of the program no free registers are available, additional spill code must be added to save the content of a currently used register on the stack and restore it after the jump was taken.

Neglecting the overhead in execution time of these additional spilling costs may degrade the quality of the SPM allocation. By not accounting for these additional timing costs, the ILP model will assume a higher gain of moving a block to the SPM than it actually provides. More importantly, neglecting the additional code size due to the newly inserted instructions may even lead to broken solutions where the SPM assignment suggested by the ILP solver does not fit into the actual SPM once the control flow has been fixed.

The problem can be tackled by introducing a new binary ILP variable $z_{i,j}$. This variable is defined to be set to 1 in case that additional jump correction code must be added at the end of block i in order to reach block j. $z_{i,j}$ can be defined by

connecting the SPM assignment variables of blocks i and j by a logical XOR (cf. Section 5.2.1.3), which enforces $z_{i,j}$ to 1 if blocks i and j are in different memories, or 0 otherwise:

$$z_{i,j} = x_i \oplus x_j \tag{7.48}$$

The costs of additional jump correction instructions, both respecting additional code size and additional timing penalties, differ not only depending on the target architecture but also on the last instruction of the original basic block. E.g., if a basic block already has jump code in order to reach a memory location far away in the original code, then the overhead of fixing this concrete jump may be 0 even in a worst-case scenario.

As the target architecture is known at compile time, these costs can be calculated prior to creating the ILP and then be added to the ILP as constants. Because basic block i may have multiple successors, the additional jump costs are added to the accumulated WCET w_i which accounts for the execution path with successor j:

$$\forall j \in \text{succ of } i : w_i \geqslant C_i + C_{i,j}^{\text{jump}} \cdot z_{i,j} + w_j \tag{7.49}$$

For the updated SPM size constraint from Eq. (7.46), the additional code size of the jump correction must only be accounted for if a jump correction is necessary, and the block is actually assigned to the SPM. This can be modeled by another binary ILP variable $s_{i,j}^{\text{jump}}$:

$$s_{i,j}^{\text{jump}} = z_{i,j} \wedge x_i \tag{7.50}$$

Accordingly, when denoting the additional code size costs of a jump correction from block i to block j by $S_{i,j}^{\text{jump}}$, Eq. (7.46) can be updated to:

$$S_{\text{SPM}} \geqslant \sum_{i \in \mathcal{B}} \left[x_i \cdot S_i + \sum_{j \in \text{succ of } i} \left(s_{i,j}^{\text{jump}} \cdot S_{i,j}^{\text{jump}} \right) \right] \tag{7.51}$$

In conclusion, if a basic block is moved to a memory location different from its successor(s) or predecessor(s), additional instructions have to be added leading to both penalties in execution time and needed space. These penalties have been modeled in the SPM optimization which is used as an example to illustrate the upcoming schedulability-aware WCET optimization framework. The calculation of the concrete constant values for jump costs has been previously published and discussed [OLF16; FK09]. Due to lack of novelty, this will not be further discussed in this thesis.

7.3.2. Miscellaneous Technical Implications

This section contains short overviews of further techncial implications of the instruction SPM allocation optimization. Since they are merely technical issues, they will not be discussed in detail.

Branch Prediction The Infineon TriCore architecture features a static branch prediction. Depending on whether the explicit target of a conditional jump has a negative or positive displacement (i.e., whether it is a backward or forward jump), the TriCore will predict the branch as either taken or not-taken. Moving basic blocks in memory and subsequently applying the previously discussed jump correction may change a negative displacement into a positive one or vice versa. A change in branch prediction has a direct impact on the execution times of the implicit and explicit successors. The original single-tasking instruction SPM optimization by [FK09] did not account for this. In order to provide tighter WCET estimates, the SPM optimization was extended to account for such changes in the course of writing this thesis [LF15a]. Due to the fact that knowledge of the concrete realization of branch prediction is not necessary for the further work, this will not be discussed in detail.

Literal Pools ARM compilers may put data objects in the `.text` section which is normally reserved for instructions. This is possible since ARM follows the Von-Neumann Architecture with one common physical memory for instructions and data. These data areas which are mixed in between regular basic blocks are called "literal pools". They are used to, e.g., store data needed for modeling large displacements for jump instructions. When moving a basic block to the SPM, it must be ensured that any literal pool that a basic block relies on is also moved to the SPM. This can be achieved easily by setting the relevant binary decision variables x equal. E.g., if a basic block i relies on another block j which is, in fact, a literal pool, keeping both blocks in the same physical memory region can be achieved by adding $x_i = x_j$ to the ILP. Handling those literal pools has previously been discussed in detail in [OLF16].

7.3.3. Simplifications

Independently from any concrete optimization, the ILP framework presented in this chapter features some simplifications in order to reduce the computational complexity.

Contexts When the same basic block is executed multiple times, the basic block may have different execution times for each execution. This may be due

to different data the instruction operates on, but also due to pipelining or caching effects. The ILP model as presented in this chapter does not model these so-called execution *contexts*. Instead, for each each execution of a basic block, its global worst-case execution time is taken. Depending on how much execution contexts differ for a concrete architecture and on the concrete program being optimized, the WCET estimate in the ILP model may be significantly more pessimistic than an estimate by an analysis which honors different contexts.

If more precise WCET estimates are needed, this issue can easily be solved as described in Section 4.2.2.4. If, e.g., a loop with 10 iterations has 3 different contexts, the loop is split into 3 distinct meta blocks in the ILP model. Each block is executed sequentially after each other. Depending on how many times each context is executed, the distinct loop meta blocks are multiplied by these execution times. E.g., if loop `Loop` is split into the contexts `LoopC0` which is executed only 1 time, `LoopC1` (also only 1 execution) and `LoopC2` (8 executions), the resulting accumulated execution time of the loop could be expressed as:

$$w_{\text{Loop}} = 1 \cdot c_{\text{LoopC0}} + 1 \cdot c_{\text{LoopC1}} + 8 \cdot c_{\text{LoopC2}} \tag{7.52}$$

However, this obviously comes at the cost of a more complex ILP formulation. For this thesis, contexts were not considered, as this precision was not needed in order to achieve profound results during the evaluation.

Inter-Basic Block Effects The ILP framework considers each basic block to be independent. For modern micro-architectures, this is not fully correct. Instead, succeeding basic blocks may feature synergetic effects due to memory block fetches and pipelining effects. These effects could be modeled in the ILP by contexts and thus be considered in the ILP model. However, as described in the previous section, this would come at the cost of a more complex ILP, inevitably leading to higher solving times. Similarly to "regular" contexts discussed above, modeling these effects was not needed in order for the optimization framework to achieve profound results. Therefore, this will not be covered any further within this thesis.

8. ILP-Based Schedulability-Aware Optimization Framework

The framework presented in Chapter 7 has been proven to provide a good basis for single-tasking WCET optimizations. However, it does not provide any means to optimize a hard real-time system featuring multiple tasks. A trivial solution to this problem would be to optimize each task separately. However, in this case, each optimization does not know of resource requirements by the other tasks in the system.

E.g., for the SPM allocation discussed in Section 7.2, each task would then make use of the whole SPM. As a result, at any time a task is preempted during runtime, the SPM has to be be reloaded. This leads to a large implementation overhead, as moving code to the SPM might not be that easy depending on the used hardware platform. Additionally, reloading the SPM implies loading its new contents from the slow Flash memory, resulting in large preemption penalties and diminishing the positive effects of executing code from the SPM. The issue can be tackled by sharing the resources which are allocated by the optimization between all tasks. Then, however, it is non-trivial to determine which amount of the overall resources should be reserved for which task.

As a solution, this chapter introduces a schedulability-aware optimization framework. The framework is able to model the worst-case scheduling behavior of all tasks in a multi-tasking hard real-time system as part of one common ILP. This allows for an optimal solution of the optimization problem, leading to an optimized task set in which each task will use the available resources to an extent which will provably result in a schedulable system, if such a solution exists. If no such solution exists (e.g., due to computing demands which simply exceed the target hardware's computational capabilities), the ILP model is rendered infeasible.

Section 8.1 introduces the general ideas of the optimization framework. Section 8.2 continues by showing how simple and strictly periodical systems can be modeled. Section 8.3 then extends these findings and shows how arbitrarily triggered systems with arbitrary deadlines can be modeled for both fixed- and dynamic-priority scheduling algorithms. The model of inter-task timing penalties for one individual preemption is discussed in Section 8.4. Finally, the chapter closes with a discussion of the limitations of the presented approach in Section 8.5.

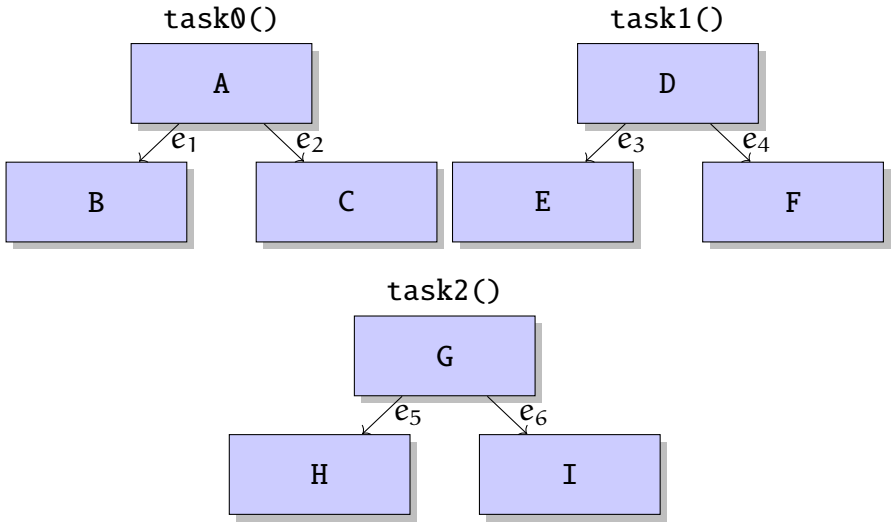


Figure 8.1: CFGs of a task set containing three tasks. Each task is depicted at basic block-level.

8.1. General Idea

The basic prerequisite for the schedulability-aware optimization framework is that the WCET of each task τ_i in the system's task set Γ is expressed in the ILP. For the upcoming sections, it is therefore assumed that there exists an integer variable c_i for each task $\tau_i \in \Gamma$ which provides a safe upper bound on the WCET of τ_i.

The framework abstracts from any concrete optimization or WCET modeling. It is therefore not relevant *how* these WCET variables are obtained. However, compiler-based optimizations often work at a basic block-level. For such optimizations, the ILP model by Suhendra et al. [Suh+05] (cf. Chapter 7) can be used as a means to model the WCET of each individual task, as illustrated in Example 8.1.

Example 8.1 (Simple Multi-Tasking System)
Fig. 8.1 shows the CFG structure of a task set containing three tasks. In a hard real-time system with static task allocation, a task can be seen as a regular function. Using the ILP model from Section 7.1, the WCETs c_0, c_1 and c_2 of tasks τ_0, τ_1 and τ_2 can be expressed by the following set of ILP equations:

$$c_0 = w_A \tag{8.1}$$
$$c_1 = w_D \tag{8.2}$$
$$c_2 = w_G \tag{8.3}$$

Modeling w_A, w_D and w_G can then be achieved as described in Section 7.1.

Once an integer variable c_i exists for each τ_i of the task set Γ, there is no longer *the* WCET which can be optimized as the ILP's objective function. A trivial new objective function to optimize the overall system would be to minimize the sum over all WCETs:

$$\min \sum_{\tau_i \in \Gamma} c_i \tag{8.4}$$

However, this naive approach does neither account for task activation patterns, deadlines nor priorities which are crucial when it comes to determining whether a system is schedulable or not (cf. Section 4.5). Additionally, the scheduling algorithm and thus the number of possible preemptions of each task by another task cannot be accounted for. This leads to the fact that any preemption penalties will also not be modeled.

The approach presented in this thesis tackles these shortcomings by providing a *constraint-based* framework. In the course of this chapter, several approaches are discussed in order to extend the ILP by sets of integer-linear (in)equations which precisely model the scheduling behavior of a task set, depending on the scheduling algorithm being used. Any optimization formulations are inherently integrated into the schedulability constraints.

This means that *any* solution returned by the ILP solver will lead to a schedulable system. The user is no longer bound to a specific optimization goal as in the single-tasking case. Instead, any arbitrary optimization goal may be chosen as needed. The user may even set a dummy optimization objective like, e.g.:

$$\min 1 \tag{8.5}$$

This will lead the ILP solver to return one presumably random solution to the ILP which leads to a schedulable system on the benefit of a much faster solving time. In this case, also a SAT-solver like Microsoft's `z3` could be used instead of an ILP solver (cf. Section 2.3). From the approach's point of view, this is simply a matter of the used solver tool and does not deeply inflict the upcoming constraints and formulations. However, it is possible that the returned solution is *barely* schedulable. I.e., if for some reason which was not predicted during system design a task is even one single CPU cycle late, the system might violate its timing constraints. To avoid such situations, the objective from Eq. (8.4) can be used in *addition* to the schedulability-enforcing constraints. As a result, the returned solution will still only return schedulable solutions but due to the additional objective, the solution is more likely to be *robust*.

It is noteworthy to add that the ILP schedulability framework provided in this chapter is *exact*. This means that for given values of c_i, the ILP formulations will return a solution if the "regular" schedulability analyses discussed in Section 4.5

Table 8.1: Properties of an exemplary strictly periodical system. This example is used throughout Section 8.2 to illustrate the schedulability-aware optimization framework.

Task	C^{unopt}	C^{opt}	T
τ_0	9	7	20
τ_1	11	3	30
τ_2	20	8	40

predict schedulability. Also, if these schedulability analyses predict that the system is not schedulable, the ILP-based approaches will lead to an infeasible ILP.

Section 8.2 introduces a rather simple formulation for schedulability-aware optimizations. However, this approach has several limitations, restricting the application to strictly periodical systems with deadlines lower than or equal to the period. Then, based on these findings, Section 8.3 derives an approach to cover systems with arbitrarily triggered tasks and with arbitrary task deadlines. Section 8.4 continues with a general overview of how to model the timing overhead inflicted by the system's scheduler and the context switching costs.

8.2. Strictly Periodical Systems

The general idea for the schedulability-aware optimization of strictly periodical systems using ILP was originally sketched and presented at the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC) which is part of the International Conference on Real-Time Networks and Systems (RTNS) [LF14]. The matured realization was then presented at the 18th International Symposium on Real-Time Distributed Computing (ISORC) [LF15a]. A slightly extended version was also discussed at the 18th International Workshop on Software & Compilers for Embedded Systems (SCOPES) [LF15b]. Based on these publications, the upcoming subsection will introduce the general ideas of the optimization framework for simple strictly periodical task sets. The following synthetic example will be used in this section to illustrate the approach:

Example 8.2 (Synthetic periodical task set)
Consider a task set Γ *consisting of 3 strictly periodically triggered tasks. To keep the example as simple as possible, all tasks have implicit deadlines, i.e., the tasks' periods equal their respective deadline:* $D_i \equiv T_i$

Table 8.1 shows the timing parameters of all tasks. Each task τ_i has a WCET C_i^{unopt} in the unoptimized case and a minimal WCET C_i^{opt} in case that this task can be fully optimized. The system load u of the unoptimized task set can be calculated using Eq. (4.24) *from Definition 4.16:*

$$u^{unopt} = \sum_{\tau_i \in \Gamma} \frac{C_i^{unopt}}{T_i} = 1.31 \tag{8.6}$$

The unoptimized system is therefore definitely not schedulable, as the system load is beyond 100 %. To introduce the optimization framework, the maximum timing gain for each τ_i is calculated:

$$G_i = C_i^{unopt} - C_i^{opt} \tag{8.7}$$

The WCET ILP variable c_i for each task is modeled by:

$$c_i \geqslant C_i^{unopt} - x_i \cdot G_i \tag{8.8}$$

$$x \in \mathbb{R}, 0 \leqslant x \leqslant 1 \tag{8.9}$$

c_i is an integer variable describing the WCET of each task τ_i after being optimized. In this example, the x_i are synthetic numbers between 0 and 1 which describe to which extent each task is being optimized. This turns the ILP into a so-called Mixed Integer-Linear Program (MILP), as it contains both integer and real-value variables.

To prevent the ILP solver from setting all x_i variables to 1, thus fully optimizing all tasks, we model an artificial resource limit for the exemplary system:

$$100 \geqslant 40 \cdot x_0 + 85 \cdot x_1 + 10 \cdot x_2 \tag{8.10}$$

This way, the ILP solver must choose which tasks to optimize to which extent in order to establish a schedulable system while still honoring the limited resources of the target system. Due to the real-valued x_i variables, this example implies that all tasks' WCETs can be scaled seamlessly. Despite being unrealistic in reality, this allows for a relatively simple illustrating exemplary task set.

8.2.1. Dynamic-Priority Scheduling

The undoubtedly most simple system has a given number of N tasks which are expressed as τ_i, $i \in 0, \ldots, N-1$. Each task has a fixed period T_i and implicit deadline (i.e., $T_i \equiv D_i$). I.e., each task is executed at exactly the defined frequency, and the deadline of each task equals this execution frequency. Additionally, preemption penalties are considered negligible.

If such a system is scheduled using an optimal dynamic scheduler like EDF, the system is schedulable if and only if its load u is lower than or equal to 1. This was previously expressed in Eq. (4.32) as part of Proposition 4.3. This equation can be directly added as a constraint to the ILP:

$$\sum_{\tau_i \in \Gamma} \frac{c_i}{T_i} \leqslant 1 \tag{8.11}$$

c_i denotes the WCET of task τ_i, and T_i is its corresponding period.

When adding Eq. (8.11) to the set of inequations proposed in Example 8.2 and solving it with the `Gurobi` ILP solver [Gur19] using a dummy objective function, the following results are retrieved:

$$x_0 = 0 \tag{8.12}$$
$$x_1 = 0.625 \tag{8.13}$$
$$x_2 = 0.5 \tag{8.14}$$

leading to the optimized WCETs:

$$c_0 = 9 \tag{8.15}$$
$$c_1 = 6 \tag{8.16}$$
$$c_2 = 14 \tag{8.17}$$

The optimized system load resolves to $u^{opt} = 1.0$. While this system is formally schedulable, it is at the brink of being unschedulable. Additionally, the available resources are not fully used:

$$40 \cdot 0 + 85 \cdot 0.625 + 10 \cdot 0.5 = 58.125 \tag{8.18}$$

In practice, it might therefore be sensible to add the minimization of the overall system load as an objective function to the ILP:

$$\min \sum_{\tau_i \in \Gamma} \frac{c_i}{T_i} \tag{8.19}$$

Using this objective, `Gurobi` returns the following solution:

$$x_0 = 0 \tag{8.20}$$
$$x_1 = 1.0 \tag{8.21}$$
$$x_2 = 1.0 \tag{8.22}$$
$$c_0 = 9 \tag{8.23}$$
$$c_1 = 3 \tag{8.24}$$
$$c_2 = 8 \tag{8.25}$$

This results in a load u^{opt} of the optimized system:

$$u^{opt} = 0.75 \tag{8.26}$$

And a resource usage of:

$$40 \cdot 0 + 85 \cdot 1 + 10 \cdot 1 = 95 \tag{8.27}$$

A full resource usage of 100 is not achieved, since due to the real-valued x_i variables and integer-valued WCET variables c_i, higher resource usage would not result in lower WCETs.

Although this optimization approach might look very simple and tempting at first, it has several drawbacks: Most importantly, it can only be used for the most simplistic real-time systems (i.e., strict periodically triggered tasks, implicit deadlines and EDF scheduling). Additionally, any eviction penalties are neglected. The result may not be safe in a real-world setup where eviction penalties do exist.

8.2.2. Fixed-Priority Scheduling

This section presents an initial approach for the schedulability-aware optimization of strictly periodical systems executed under a fixed-priority schedule. The approach was previously published in [LF15a]. This first approach assumes that each task's deadline is smaller than or at most equal to its respective period.

The approach presented in the following does not make any assumptions on *how* priorities were assigned to each task. They may follow a common scheduling algorithm like RMS, but may also be assigned arbitrarily by the system designer. In Example 8.2, all tasks' deadlines are implicit. Therefore, RMS scheduling can be used as an optimal fixed-priority scheduling algorithm for this task set.

While the maximum system load as described in Eq. (8.11) is still a necessary requirement for fixed-priority scheduling, it is no longer sufficient. The necessary equation to calculate the WCRT of a task τ_i was introduced earlier in Eq. (4.27):

$$r_i = c_i + \sum_{j=0}^{i-1} \left(\left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j \right) \tag{4.27}$$

Similar to the previously presented dynamic-priority approach, this initial approach for fixed-priority scheduling again considers preemption penalties to be negligible. While it would be technically feasible to include a precise model of the eviction penalties as introduced in Proposition 4.2, this would increase the complexity of the ILP formulation significantly. Especially handling the additional

multiplication introduced by the min() term in Eq. (4.29) would increase the ILP solving times massively. [LF15a] shows a principle way to add a safe over-approximation for the preemption penalties without the necessary multiplication. However, due to the fact that this introduces additional pessimism into the model, and the complication of the formulas being discussed in the following, this will not be discussed any further.

Eq. (4.27) cannot be directly transcribed as set of integer-linear (in)equations. The reason for this is that the second term, $\left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j$, denotes a quadratic term as both r_i and c_j are integer variables in the ILP. Additionally, the original formulation has to be solved iteratively. Due to the ceil operator, it cannot be simply reformulated. As a first step to circumvent these issues, Eq. (4.27) is rewritten:

$$r_i = c_i + \sum_{j=0}^{i-1} h_{i,j} \tag{8.28}$$

$h_{i,j}$ is a helper variable which expresses the timing penalty added to the WCRT of τ_i by a higher-priority task τ_j. This rewritten equation can now obviously be added to the ILP. For Example 8.2, this leads to:

$$r_0 = c_0 \tag{8.29}$$
$$r_1 = c_1 + h_{1,0} \tag{8.30}$$
$$r_2 = c_2 + h_{2,0} + h_{2,1} \tag{8.31}$$

$h_{i,j}$ may be expressed linearly as follows: In any case, τ_i may be evicted at least once by τ_j. However, as long as $r_i \leqslant T_j$, it can obviously be safely assumed that τ_i is also evicted *at most* once by τ_j. If $\left\lceil \frac{r_i}{T_j} \right\rceil \geqslant 2$, then τ_i may be evicted twice, but as long as $\left\lceil \frac{r_i}{T_j} \right\rceil < 3$, it is safe that it is also evicted *at most* twice. This can be repeated up to $r_i = D_i$. Due to the fact that the actual r_i after optimization is not known at the time the ILP is created, all possible values of $h_{i,j}$ must be modeled. This can be done by using a case structure:

$$h_{i,j} \geqslant \begin{cases} 1 \cdot c_j & \text{if } r_i > 0 \cdot T_j \\ 2 \cdot c_j & \text{if } r_i > 1 \cdot T_j \\ \dots & \\ \left\lceil \frac{D_i}{T_j} \right\rceil \cdot c_j & \text{if } r_i > \left(\left\lceil \frac{D_i}{T_j} \right\rceil - 1 \right) \cdot T_j \end{cases} \tag{8.32}$$

For Example 8.2 and using the conditional expressions from Section 5.2.2.2, $h_{1,0}$ is expressed as:

$$r_1 > 0 \cdot 20 \Rightarrow h_{1,0} \geqslant 1 \cdot c_0 \tag{8.33}$$

$$r_1 > 1 \cdot 20 \Rightarrow h_{1,0} \geqslant 2 \cdot c_0 \tag{8.34}$$

$h_{2,1}$ and $h_{2,0}$ are modeled accordingly. While this formulation looks still very compact, the number of equations rises significantly if D_i is much larger than T_j.

Proposition 8.1
Eq. (8.32) *provides a safe upper bound on the delay inflicted to task* τ_i*'s WCRT* r_i *by task* τ_j *with* $P_j < P_i$*, if* r_i *does not exceed the respective deadline* D_i.

Proof. To prove this proposition, the following three cases are considered one at a time:

$$r_i \leqslant \left(\left\lceil \frac{D_i}{T_j} \right\rceil - 1 \right) \cdot T_j \tag{8.35}$$

$$r_i = \left\lceil \frac{D_i}{T_j} \right\rceil \cdot T_j \tag{8.36}$$

$$r_i > \left\lceil \frac{D_i}{T_j} \right\rceil \cdot T_j \tag{8.37}$$

As long as Eq. (8.35) holds, the preemption penalty is given correctly by Eq. (8.32), as it is simply a rewritten form of the term $\left\lceil \frac{r_i}{T_j} \right\rceil$ from Eq. (4.27).

To prove correctness in case that Eq. (8.36) holds (i.e., we have exactly one preemption more than expressed by Eq. (8.35)), Eq. (8.36) is rewritten as follows:

$$\frac{r_i}{T_j} = \left\lceil \frac{D_i}{T_j} \right\rceil \tag{8.38}$$

Removing the ceil operator leads to the following relationship:

$$\frac{D_i}{T_j} \leqslant \frac{r_i}{T_j} < \frac{D_i + 1}{T_j} \tag{8.39}$$

The WCRT r_i may never be larger than D_i. Therefore, r_i must equal D_i in order for Eq. (8.36) to hold. This results in

$$\left\lceil \frac{r_i}{T_j} \right\rceil \cdot c_j = \left\lceil \frac{D_i}{T_j} \right\rceil \cdot c_j \equiv h_{i,j} \tag{8.40}$$

Therefore, Eq. (8.32) also holds in this scenario.

In the final case of Eq. (8.37), τ_i's WCRT r_i must be greater than D_i. However, this violates the system's schedulability constraints, thus the case may never occur. □

If τ_i's WCRT exceeds its deadline, the system is definitely not schedulable without any further tests needed. This constraint of the maximum allowed value on r_i is finally added to the ILP:

$$r_i \leqslant D_i \tag{8.41}$$

As a consequence, Eq. (8.32) is a valid linear reformulation of the preemption penalties of a higher priority task from Eq. (4.27). An ILP using these constraints will result in a schedulable system if a valid result is obtained. In case that the system violates a deadline, the ILP is provably infeasible. For Example 8.2 with a dummy objective being set, the `Gurobi` ILP solver returns:

$$x_0 = 0 \tag{8.42}$$
$$x_1 = 1.0 \tag{8.43}$$
$$x_2 = 1.0 \tag{8.44}$$
$$c_0 = 9 \tag{8.45}$$
$$c_1 = 3 \tag{8.46}$$
$$c_2 = 8 \tag{8.47}$$

which, for this simple example, accidentally equals the result for EDF scheduling with $\min u$ as objective function. When performing a WCRT analysis using Eq. (4.27), the following WCRTs are obtained:

$$r_0 = 9 \tag{8.48}$$

$r_0 = 9 \leqslant 20$, therefore τ_0 is schedulable.

$$r_1^{(1)} = 3 + 9 \cdot \left\lceil \frac{3}{20} \right\rceil = 12 \tag{8.49}$$
$$r_1^{(2)} = 3 + 9 \cdot \left\lceil \frac{11}{20} \right\rceil = 12 \tag{8.50}$$

The iterative formulation terminates with $r_1 = 12 \leqslant 30$, therefore τ_1 is schedulable.

$$r_2^{(1)} = 8 + 9 \cdot \left\lceil \frac{8}{20} \right\rceil + 3 \cdot \left\lceil \frac{8}{30} \right\rceil = 20 \tag{8.51}$$
$$r_2^{(2)} = 8 + 9 \cdot \left\lceil \frac{20}{20} \right\rceil + 3 \cdot \left\lceil \frac{20}{30} \right\rceil = 20 \tag{8.52}$$

The iterative formulation terminates with a $r_2 = 20 \leqslant 40$, therefore τ_2 is also schedulable. Thus, the system was successfully optimized and schedulability could be established.

The previously described approaches showed how to model an ILP-based optimization for strictly periodical systems. However, they have several drawbacks: First of all, they are limited to strictly periodical systems. Systems underlying jitter, or systems with arbitrary, non-periodic task activation patterns must be over-approximated by using the tightest possibly occurring period. This approach guarantees safe results but obviously also introduces an enormous amount of pessimism.

The initial approach does not offer a way to account for preemption penalties like eviction overheads or cache-related effects. For fixed-priority systems, the possibility of adding eviction penalties was presented at the 18th International Workshop on Software & Compilers for Embedded Systems (SCOPES) [LF15b]. However, dynamic-priority scheduling cannot be modeled that easily due to the ILP's very simplistic approach. Furthermore, the previously discussed approach restricts the valid deadlines. For fixed-priority systems, deadlines may not be larger than the respective task's period, and for dynamic-priority systems, even only implicit deadlines can be modeled.

8.3. Modeling Arbitrary Activation Patterns

The shortcomings of the simple approach discussed in the previous section are overcome by modeling the schedulability of the target system using event-based scheduling analysis as presented in Section 4.5.2. The approach was first presented at the 20th Design, Automation and Test in Europe (DATE) conference [LF17]. To illustrate the upcoming approach, Example 8.2 is modified. Instead of using strictly periodically triggered tasks, each task is modeled as a periodic task with initial burst. Additionally, a deadline greater than the minimum time between two occuring events is chosen.

Figs. 8.2 and 8.3 show the task activation pattern of task τ_0 from Example 8.2: Initially, τ_0 is triggered 4 times with a period of 5 time units. Then, after this initial burst, the task is triggered at a rate of 20 time units. To model such a system, the approaches presented in Section 8.2 would have to assume a fixed period $T_0 = 5$ for a safe WCRT estimation. However, this obviously introduces significant pessimism. Example 8.3 shows how this system can be expressed using event density and interval functions.

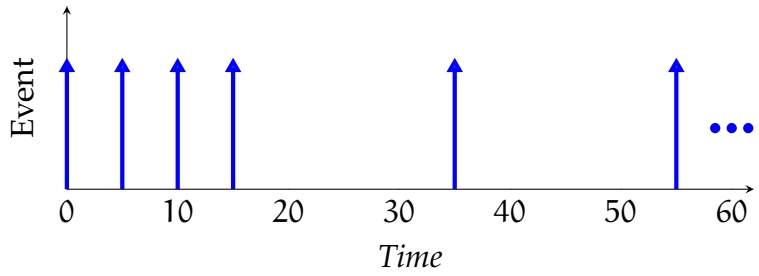


Figure 8.2: Activation pattern for task τ_0. The task is initially triggered 4 times with a period of 5 time units. Then, all subsequent instances are triggered with a period of 20 time units.

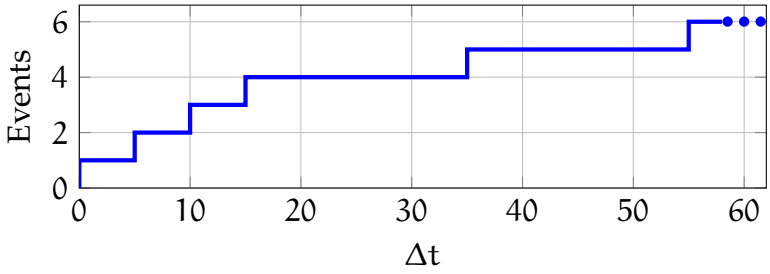


Figure 8.3: Density function for τ_0.

Example 8.3 (Synthetic aperiodic task set)
The WCETs and deadlines of all three tasks from Example 8.2 are left unchanged. They are repeated in Table 8.2. For fixed-priority scheduling, DMS scheduling is assumed.

For the sake of simplicity, all three tasks' bursts consist of 4 consecutive activations with an inter-arrival time of 5 time units each. As a result, the tasks' density functions resolve to:

$$\eta_0(\Delta t) = \begin{cases} 0 & \text{if } \Delta t < 0 \\ \lceil \frac{\Delta t}{5} \rceil & \text{if } 0 \leqslant \Delta t \leqslant 15 \\ 3 + \lceil \frac{\Delta t}{20} \rceil & \text{if } \Delta t > 15 \end{cases} \tag{8.53}$$

$$\eta_1(\Delta t) = \begin{cases} 0 & \text{if } \Delta t < 0 \\ \lceil \frac{\Delta t}{5} \rceil & \text{if } 0 \leqslant \Delta t \leqslant 15 \\ 3 + \lceil \frac{\Delta t}{30} \rceil & \text{if } \Delta t > 15 \end{cases} \tag{8.54}$$

$$\eta_2(\Delta t) = \begin{cases} 0 & \text{if } \Delta t < 0 \\ \lceil \frac{\Delta t}{5} \rceil & \text{if } 0 \leqslant \Delta t \leqslant 15 \\ 3 + \lceil \frac{\Delta t}{40} \rceil & \text{if } \Delta t > 15 \end{cases} \tag{8.55}$$

Table 8.2: Properties of an exemplary system. WCETs, periods and deadlines are taken from Example 8.2.

Task	C^{unopt}	C^{opt}	D
τ_0	9	7	20
τ_1	11	3	30
τ_2	20	8	40

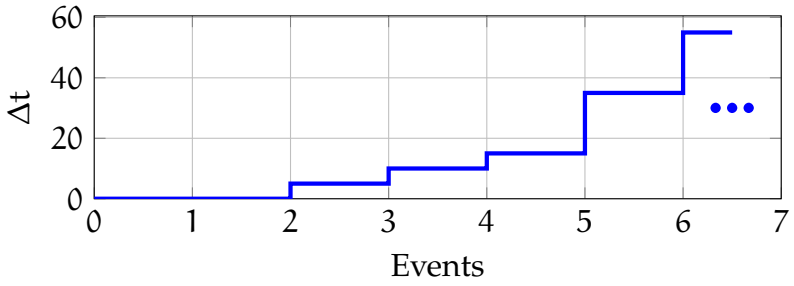


Figure 8.4: Interval function for τ_0 from Example 8.3.

The interval functions may subsequently be expressed as:

$$\delta_0(n) = \begin{cases} 5 \cdot (n-1) & \text{if } n \leqslant 4 \\ 20 \cdot (n-3) - 5 & \text{if } n > 4 \end{cases} \tag{8.56}$$

$$\delta_1(n) = \begin{cases} 5 \cdot (n-1) & \text{if } n \leqslant 4 \\ 30 \cdot (n-3) - 15 & \text{if } n > 4 \end{cases} \tag{8.57}$$

$$\delta_2(n) = \begin{cases} 5 \cdot (n-1) & \text{if } n \leqslant 4 \\ 40 \cdot (n-3) - 25 & \text{if } n > 4 \end{cases} \tag{8.58}$$

Fig. 8.4 shows the interval function for τ_0.

8.3.1. Dynamic-Priority Scheduling

Schedulability tests for arbitrarily stimulated systems scheduled under dynamic-priority scheduling were discussed in Section 4.5.2.2. Eq. (4.43) described the condition under which a system is schedulable:

$$\Delta t \geqslant \sum_{\forall \tau_i \in \Gamma} \left[\eta_i (\Delta t - D_i) \cdot (c_i + e_i) \right] \tag{4.43}$$

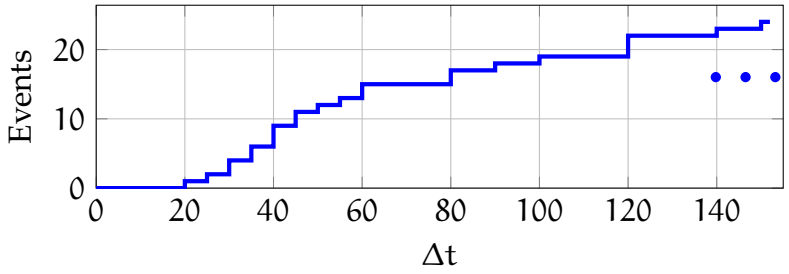


Figure 8.5: Density function for the union over all tasks from Example 8.3.

As discussed in Section 4.5.2.2, this inequation must be tested at each Δt at which the *demand* of processing time increases, in order give a safe estimate on the system's schedulability. For a given task τ_i, these values obviously occur D_i time units after an instance of this task is triggered for execution. A function $\eta_i^D(\Delta t)$ which returns the respective values of Δt can thus easily be derived for a given task τ_i by shifting its density function η_i by its respective deadline D_i:

$$\eta_i^D(\Delta t) = \eta_i(\Delta t - D_i) \tag{8.59}$$

In order to prevent the same values Δt to be tested multiple times when iterating over all tasks in the task set Γ, $\eta_\Gamma^D(\Delta t)$ is defined as the union over all demand density functions of all tasks τ_i in the task set Γ:

$$\eta_\Gamma^D(\Delta t) = \bigcup_{\forall \tau_i \in \Gamma} \eta_i^D(\Delta t) \tag{8.60}$$

For the task set Γ of the exemplary system defined in Example 8.3, this means that in the first 20 time units, no task must finish, then at least one task (τ_0) must have finished, after 30 time units, two tasks in the task set (one instance of τ_0 and one instance of τ_1) must have finished, ... For the task set from Example 8.3, this union can be written as:

$$\eta_\Gamma^D(\Delta t) = \eta_0(\Delta t - D_0) + \eta_1(\Delta t - D_1) + \eta_2(\Delta t - D_2) \tag{8.61}$$

In a nutshell, $\eta_\Gamma^D(\Delta t)$ describes how many instances of tasks in the system must have finished their execution in a given time interval. Fig. 8.5 shows the plot of the resulting density function.

The corresponding inverse, $\delta_\Gamma^D(n)$ can now be used to retrieve all time intervals Δt which must be checked by the scheduling analysis. It can be deduced by rotating $\eta_\Gamma^D(\Delta t)$ by 90° and mirroring it. In practice, also a numerical approach like, e.g., a binary search on the density function may be applied in order to

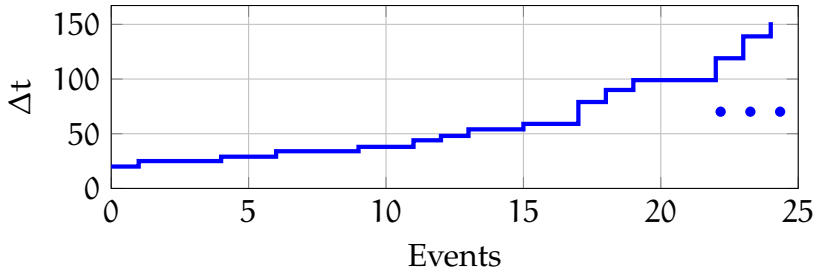


Figure 8.6: Interval function for the union over all tasks from Example 8.3.

retrieve concrete values for the minimal interval in which n events may occur, without having to convert arbitrarily complex density functions into an explicit mathematical formula. Each distinct value Δt of $\delta_\Gamma^D(n)$ with $n > 0$ must be tested, up to the maximum busy window ΔT_B (cf. Section 4.5.2.2).

For the given Example 8.3, the deadline of the first task which has to finish is the first instance of τ_0 after 20 time units. The next interval after which a task has to be finished, is 25 time units (the second instance of τ_0), ... (cf. Figs. 8.5 and 8.6). The schedulability test provided in Eq. (4.43) can now be expressed as the following set of ILP inequations:

$$\delta_\Gamma^D(0) + 1 \geqslant \sum_{\tau_i \in \Gamma} \left\{ \eta_i\left(\delta_\Gamma^D(0) - D_i + 1\right) \cdot (c_i + e_i) \right\} \tag{8.62}$$

$$\delta_\Gamma^D(1) + 1 \geqslant \sum_{\tau_i \in \Gamma} \left\{ \eta_i\left(\delta_\Gamma^D(1) - D_i + 1\right) \cdot (c_i + e_i) \right\} \tag{8.63}$$

$$\dots$$

$$\Delta T_B \geqslant \sum_{\tau_i \in \Gamma} \left\{ \eta_i\left(\Delta T_B - D_i\right) \cdot (c_i + e_i) \right\} \tag{8.64}$$

e_i is an integer variable which denotes the maximum time penalty inflicted to any other tasks in the system if they are preempted by τ_i (e.g., due to CRPD).

The addition of the $+1$ in the interval to check stems from the fact that a new instance of a job is triggered an infinitesimally small time instant *after* the point of discontinuity. E.g., at $\Delta t = 0$, $\tau_0 = 0$. Accordingly, at $\Delta t = D_i$, the resource demand of τ_0 is still $\eta_i(\Delta t - D_i) \cdot (c_i + e_i) = 0$. Due to the fact that time is expressed as integer values within the optimization framework, each time interval must be increased by one time unit to account for this. I.e., the time intervals to be analyzed in the ILP are $\Delta t = 21, 26, 31, 36, \dots$.

A safe upper bound on the maximum allowed busy window can be obtained by calculating the least common multiple of all fundamental periods which occur

in the periodical part of the system, plus the maximum time-interval in which non-periodic events may occur (cf. Section 4.5.2.1):

$$\Delta T_B = 15 + \text{lcm}\,(20, 30, 40) = 135 \tag{8.65}$$

Calculating exact values for the preemption penalty inflicted at each preemption is a tedious task in dynamic-priority scheduling. Due to the nature of a dynamic-priority scheduling algorithm, it cannot be safely predicted which task will have which priority at which point in time at runtime. Therefore, if the preemption penalties vary depending on which task evicts which other task, the maximum has to be taken into consideration:

$$\forall \tau_j \in \Gamma, j \neq i : e_i = \max\left(\{e_{i,j}\}\right) \tag{8.66}$$

The maximum can be modeled within the ILP as proposed in Section 5.2.3. How the actual ILP variable $e_{i,j}$, which models the overhead for one preemption of task τ_i by task τ_j is determined, depends on the actual system and is not relevant for the schedulability analysis framework itself. Modeling the preemption penalties is therefore discussed separately in Section 8.4. The schedulability constraints above can therefore be written as:

$$\begin{aligned} 21 &\geqslant \eta_0\,(21-20)\cdot c_0 + \eta_1\,(21-30)\cdot c_1 + \eta_2\,(21-40)\cdot c_2 && (8.67)\\ &= 1\cdot c_0 \\ 26 &\geqslant \eta_0\,(26-20)\cdot c_0 + \eta_1\,(26-30)\cdot c_1 + \eta_2\,(26-40)\cdot c_2 && (8.68)\\ &= 2\cdot c_0 \\ &\dots \\ 136 &\geqslant \eta_0\,(136-20)\cdot c_0 + \eta_1\,(136-30)\cdot c_1 + \eta_2\,(136-40)\cdot c_2 && (8.69)\\ &\geqslant 10\cdot c_0 + 8\cdot c_1 + 8\cdot c_2 \end{aligned}$$

The requested processing time is greater than the available time for at least one time interval if a modeled system cannot be repaired. As a result, the ILP will be infeasible.

As discussed above, the upper bound on the interval function $\delta_\Gamma(n)$ is given by the maximum valid busy window, after which the periodical activation patterns of the system are repeated. To ensure a safe framework, the system load u must be restricted to be smaller than or equal to 1. To recall the rationale behind this, consider a very simple system consisting of only one single task τ_0 with a WCET $c_0 = 5$, a fixed period $T_0 = 4$ and a deadline $D_0 = 10$. Calculating the system's hyper-period is trivial: $\text{lcm}\,(4) = 4$. When performing the schedulability analysis up to the hyper-period, no deadline will be violated, since the deadline of τ_0, $D_0 = 10$. However, due to the system load being beyond 100 %, one of the subsequently triggered instances of τ_0 will inevitably miss its deadline.

The reason for this is rather simple: As, by definition, the system's worst-case behavior will repeat after the hyper-period, there is no chance for the system to "catch up" if the system cannot finish all tasks' executions within the hyper-period. Instead, this overload will inevitably accumulate over multiples of the hyper-periods until a deadline violation occurs.

Eq. (4.33) in Definition 4.22 gives the system load for arbitrarily triggered systems as follows:

$$u = \lim_{\Delta t \to \infty} \sum_{i=0}^{N-1} \frac{c_i \cdot \eta_i(\Delta t)}{\Delta t} \tag{4.33}$$

As long as $u \leqslant 1$, the resource demand is smaller or equal to the available computational resources, and the system is not in overload.

Eq. (4.33) cannot directly be added to an ILP as there exists no notion of infinity. For systems with periodically recurring activation patterns, the hyper-period can be used as Δt, since, by definition, the system behavior repeats itself after that time interval.

When the activation pattern features an initial burst, only the fundamental periods of the periodical behavior need to be considered when calculating u. As a result, the least common multiple of the fundamental periods of the periodical activation patterns can be used as a Δt and an interval function $\delta_i^{(\text{periodic})}$, which only contains the periodically repeated activation patterns, can be used in Eq. (4.33). The reason for this is that when building the lim for $\Delta t \to \infty$, any events which do *not* repeat periodically but only occur once will diminish and will not influence the limit calculation. With ΔT_{HP} being the hyper-period of periodically recurring activation patterns of the system, this can then be directly written as an ILP inequation:

$$1 \leqslant \sum_{\tau_i \in \Gamma} \left\{ (c_i + e_i) \cdot \frac{\eta_i^{(\text{periodic})}(\Delta T_{HP})}{T_{HP}} \right\} \tag{8.70}$$

It should be noted that this equation dominates Eq. (8.64). Therefore, the constraint introduced by Eq. (8.64) is redundant and could be removed from the ILP model. In practice, any modern ILP solver will remove such redundant constraints in its presolve phase, thus it will not add any notable increase to the solving time. For Example 8.3, this leads to:

$$\begin{aligned} 135 &\geqslant \eta_0(135) \cdot c_0 + \eta_1(120) \cdot c_1 + \eta_2(120) \cdot c_2 \\ &\geqslant 9 \cdot c_0 + 7 \cdot c_1 + 6 \cdot c_2 \end{aligned} \tag{8.71}$$

8.3.2. Fixed-Priority Scheduling

This section presents a basic approach of a schedulability-aware ILP model for arbitrarily triggered systems with arbitrary deadlines if a fixed-priority scheduling algorithm is being used. The approach was originally presented in [LF17]. For the sake of simplicity, eviction penalties and task switching costs are neglected in this section. Extensions for such penalties and scalability improvements will be presented in Section 8.3.3.

WCRT analysis of systems executed under a fixed-priority schedule has been discussed in Section 4.5.2.1. The response time r_i of a task τ_i was expressed in Eqs. (4.34) and (4.35) as:

$$r_i = \max_{\forall K \in [1,\ldots,\eta_i(\Delta T_B))]} \{r_{i,K} - \delta_i(K)\} \tag{4.34}$$

$$r_{i,K} = \min \left\{ \Delta t \,|\Delta t = K \cdot c_i + \sum_{j=0}^{i-1} [\eta_j(\Delta t) \cdot c_j] \right\} \tag{4.35}$$

As long as the WCRT of each task is below its respective deadline when applying these iterative formulas, the system is provably schedulable. In order to express these equations as part of the ILP, the property of the interval function $\delta_\Gamma(k)$ being a piecewise constant step function can be exploited. This property can be directly derived from Definition 4.12.

When looking at Fig. 8.6 which denotes the interval function for the union over all tasks in the system, it can easily be observed that for certain numbers of events, the minimal interval does not change. E.g., the minimal interval for 9 events in the system is 15 time units. The minimal time interval for 10 and 11 events is also 15 time units. Only for 12 events, the minimal interval is increased, i.e., a minimal interval of 20 time units is needed.

In a worst-case scenario, the maximum densitiy of occuring events is considered, as this triggers the maximum interference between tasks. As a result, in WCRT analysis, it is assumed that 9 events are triggered in the interval of 15 time units, for the given example. However, this also implies that no further events will be triggered up to an interval of 20 time units. Therefore, it is safe to calculate the response time only at the points of discontinuity instead of having to analyze any arbitrary $\Delta t \in \mathbb{R}^+$.

The WCRT r_i for each task $\tau_i \in \Gamma$, as well as the individual $r_{K,i}$ will be expressed in the ILP as follows: Modeling the maximum function from Eq. (4.34)

is straightforward. For each argument, an inequation is added modeling a lower bound on r_i. Eq. (4.34) therefore results in the following inequation system:

$$r_i \geqslant r_{i,1} - \delta_i(1) \tag{8.72}$$
$$r_i \geqslant r_{i,K} - \delta_i(K) \tag{8.73}$$
$$\dots$$
$$r_i \geqslant r_{i,\hat{K}_i} - \delta_i\left(\hat{K}_i\right) \tag{8.74}$$

$\delta_i(K)$ is constant for any given task and $K \in \left[1, \dots, \hat{K}_i\right]$, as it only depends on the event function describing the task stimulation but not on any task's WCET. Therefore, it can be calculated prior to solving the ILP.

Due to relying on the event model, the approach is obviously not limited to periodical systems. As long as the user can provide both interval and event density function, Eqs. (8.72) to (8.74) can be modeled by the presented set of integer-linear (in)equations.

To ensure that the ILP solver cannot return a solution leading to an unschedulable system, an additional constraint ensures that the response time has to be lower than or equal to the task's deadline:

$$r_i \leqslant D_i \tag{8.75}$$

The K indicator in $r_{K,i}$ denotes the K'th instance of task i in its WCRT estimation. For tasks with a deadline lower than or equal to their minimal activation distance, the maximum value on K, $\hat{K}_i$ is 1, as a task must not be triggered another time before its previous instance has finished its execution. Otherwise, the task's deadline constraint would be violated. Therefore, no (in)equations for higher numbers of K have to be modeled in the ILP in this special case.

As discussed in Section 4.5.2.1, Eq. (4.35) uses a fixed-point iteration to calculate each $r_{K,i}$. For the ILP formulation, this iterative formula must be rewritten to a set of linear inequations. This will be shown in the following.

A safe lower bound on each task τ_i's WCRT is given by its respective WCET c_i. Because all execution times are positive, it is obvious that $r(K, \tau_i)$ is monotonically increasing. This means that larger values of Δt will never result in a smaller value of $r_{i,K}$. Therefore, a safe lower bound and initial value of Δt which has to be considered is given by c_i.

To express $r_{i,K}$ by a set of linear inequations, the timing costs added to $r_{K,i}$ by a higher priority task j for each K is denoted by the ILP variable $t_{K,i,j}$. This leads to the following ILP equations:

$$r_{i,1} = 1 \cdot c_i + \sum_{j=0}^{i-1} t_{1,i,j} \tag{8.76}$$

$$r_{i,K} = K \cdot c_i + \sum_{j=0}^{i-1} t_{K,i,j} \tag{8.77}$$

$$\dots$$

$$r_{i,\hat{K}_i} = \hat{K}_i \cdot c_i + \sum_{j=0}^{i-1} t_{\hat{K}_i,i,j} \tag{8.78}$$

For each $K \in [1, \dots, \hat{K}_i]$, $t_{K,i,j}$ can be expressed as a set of conditional inequations:

$$t_{K,i,j} \geqslant \begin{cases} 1 \cdot c_j & \text{if } r_{i,K} > \delta_j(1) \\ N \cdot c_j & \text{if } r_{i,K} > \delta_j(N) \\ & \dots \\ \hat{K}_i \cdot c_j + \hat{K}_i \cdot e_j & \text{if } r_{i,K} > \delta_j\left(\hat{K}_i\right) \end{cases} \tag{8.79}$$

The underlying idea is analogous to that given in Section 8.2.2 for strictly periodical systems: If the response time r_i of task τ_i is greater than the minimal distance between N subsequent activations of a higher priority task τ_j, then τ_i may be preempted N times by this task τ_j. Due to the fact that the worst-case number of evictions of task τ_i by τ_j is known, eviction penalties can easily be added to the inequation system. As explained above, in this section, however, it is assumed that eviction penalties can be neglected.

Eq. (8.79) can be expressed in the ILP by adding a set of inequations for each $N \in [1, \dots, \hat{K}_i]$:

$$t_{K,i,j} + b_{N,i,j} \cdot M_{j,N} \geqslant N \cdot c_j \tag{8.80}$$

$$r_{i,K} > \delta_j(N) + L_{i,K} \cdot (1 - b_{N,i,j}) \tag{8.81}$$

$b_{N,i,j}$ is a binary decision variable. Eq. (8.81) enforces the ILP solver to set $b_{N,i,j}$ to 0 if $r_{K,i}$ is greater than the minimal time interval in which task τ_j may be triggered N times. This forces the penalty due to higher-priority tasks $t_{K,i,j}$ in Eq. (8.80) to be greater or equal to $N \cdot c_j$.

$M_{j,N}$ and $L_{i,K}$ are sufficiently large constants. $M_{j,N}$ ensures that Eq. (8.80) will always hold if $b_{N,i,j}$ is chosen to be 1 by the ILP solver. In a schedulable system,

the WCET c_j must not exceed the respective deadline D_j, therefore a safe lower bound on $M_{j,N}$ is given by:

$$M_{j,N} = N \cdot D_j \tag{8.82}$$

$L_{i,K}$ ensures that Eq. (8.81) is always fulfilled if $b_{N,i,j}$ is 0. In analogy to $M_{j,N}$, $r_{i,K} \leqslant K \cdot D_i$ holds for a schedulable system. Therefore, a sufficiently large $L_{K,i}$ can be chosen as:

$$L_{i,K} = K \cdot D_i \tag{8.83}$$

As a result of this constraint-based approach, any valid solution of the ILP is a provably schedulable system. Therefore, the ILP objective function may be chosen arbitrarily, identically to the simple approach presented in Section 8.2.2.

The approach presented in this section allows for the schedulability-aware optimization of arbitrarily triggered multi-tasking systems with fixed-priority scheduling. Although keeping it very close to the original iterative WCRT calculation makes the approach relatively easy to understand, the complexity of the underlying ILP can grow massively. The cases structure in Eq. (8.79) has to be modeled for each task instance K of each task τ_i *and* for each task τ_j with $j < i$ (i.e., for each task with a higher priority). The maximum value for K depends on whether the tasks' deadlines exceed their meinimum inter-arrival time. For the earlier defined Example 8.3, this applies. Therefore, all K up to the maximum busy window must be checked.

By Definition 4.23, the hyper-period is the least common multiple of all activation periods within the task set Γ. When optimizing a system with recurring activation patterns, the maximum busy window ΔT_B to be covered is the hyper-period plus the time window in which the aperiodic behavior may occur:

$$\Delta T_B = 15 + \operatorname{lcm}(20, 30, 40) = 135 \tag{8.84}$$

For systems with initial bursts, the same considerations apply as previously discussed for dynamic-priority scheduling in Section 8.3.1. For the given example, this leads to the following $\hat{K}_i$:

$$\hat{K}_0 = \eta_0(135) = 10 \tag{8.85}$$

$$\hat{K}_1 = \eta_1(120) = 8 \tag{8.86}$$

$$\hat{K}_2 = \eta_2(120) = 7 \tag{8.87}$$

This results in $10 + 8 \cdot 2 + 7 \cdot 3 = 47$ case structures, even for this small example. Each case structure has $\hat{K}_i$ cases, thus the maximum number of events within

the hyper-period quadratically increases the number of inequations. This finally leads to $10^2 + 8^2 \cdot 2 + 7^2 \cdot 3 = 375$ conditional ILP expressions and a subsequently high number of binary helper variables just for this very small task set with an accordingly small hyper-period. Despite this huge complexity, the formulation is still computationally feasible for small task sets using modern commercial ILP solvers like `Gurobi`.

8.3.3. Improved Model for Fixed-Priority Scheduling

This section shows an improved approach for fixed-priority scheduling. It allows for a much leaner ILP formulation without loosing any flexibility or precision. The size of the ILP of this approach grows only linear with the hyper-period. This approach has been published as an article in ACM Transactions on Embedded Systems (TECS) [LOF20].

The underlying idea of this improved approach is to adapt the event stream-based schedulability analysis of dynamic-priority systems (cf. Section 8.3.1), instead of using WCRT analysis. Similar to dynamic-priority scheduling, this approach checks the resource demand for each task at each point of discontinuity of a given task set's Γ unified event density function $\eta_\Gamma(\Delta t)$. However, there are differences due to the nature of the fixed-priority schedule. In contrast to dynamic-priority scheduling, the set of tasks which are allowed to preempt a given task τ_i is predetermined. The idea of the subsequent approach is to analyze the schedulability of each task τ_i separately. For this schedulability test, only τ_i and tasks with higher priority need to be considered. If the schedulability constraints hold for all these tasks, then the system is schedulable.

The ILP integer variable $\nu_{i,K,\Delta t}$ models the resource demand of the K'th instance of task τ_i in the time interval Δt. Each instance K then has to be checked for schedulability. If, for a given K and for at least one analyzed time interval Δt, the resource demand is lower than the interval Δt itself, the K'th instance of τ_i provably finishes within the hyper-period (i.e., the instance does not starve). This is a necessary condition for the task to be schedulable. If this task instance also finishes before missing its deadline, the task is schedulable. The second test has to be added in order to be able to model arbitrary deadlines (which may not only be longer but also smaller than the inter-arrival time).

Informally spoken, the "traditional" schedulability analysis as presented in Section 8.3.1 can be stopped if the busy window is smaller than the time interval Δt and the task is not missing its deadline. However, due to the ILP approach, the analysis cannot be "stopped" at some point when a fixed point has been reached. Instead, the inequations have to cover the maximum possible number of calculations (i.e., analyze each Δt up to the maximum busy window, as discussed

in Section 8.3.1). The stopping criterion is "emulated" by stating that only *at least one* of the calculations must hold. This drastically reduces the ILP's complexity compared to the approach presented in the previous section.

In analogy to the schedulability test for dynamic priorities, $K = 1, \ldots, \hat{K}$ denotes the number of instances of the currently analyzed task τ_i. If the task's deadline is smaller than or equal to its minimal inter-arrival time, then $\hat{K} \equiv 1$ as the task's timing constraints are violated (and the system is thus not schedulable) prior to another instance of τ_i being triggered. Otherwise, if $D_i > \eta_i(2)$, then more than one instance of τ_i may be ready for execution at the same time. In this case, in complete analogy to the scheduling test for dynamic-priority scheduling, $\hat{K}$ equals the hyper-period of the system, plus the maximum time-interval in which non-periodic events may occur (cf. Section 4.5.2.1).

In contrast to the previously presented approach, using the event-based models has the advantage of not containing any quadratic terms anymore. For each Δt, all parameters except the WCET variables c_i and the preemption penalty variables $e_{i,j}$ are constant with regard to the ILP formulation. Therefore, the ILP only grows linear with respect to the number of different time intervals Δt which have to be analyzed. Furthermore, the ILP size depends on the number of tasks in a linear fashion.

$\nu_{i,k,\Delta t}$ is defined as an integer variable modeling the resource demand of the K'th instance of task τ_i in the time interval Δt. Then, Eqs. (4.37) and (4.38) can easily be transcribed as ILP constraint as follows:

$$\nu_{i,K,\Delta t} = K \cdot c_i + \sum_{j=0}^{i-1} \Bigg\{ \eta_j(\Delta t) \cdot c_j + \sum_{n=j+1}^{i} \min\left[\eta_n(\Delta t) \cdot \eta_j(D_n), \eta_j(\Delta t)\right] \cdot e_{n,j} \Bigg\} \tag{8.88}$$

Two additional constraints are added to perform the actual scheduling test:

$$o_{i,K,\Delta t} \equiv 1 \Rightarrow \nu_{i,K,\Delta t} \leqslant \Delta t \tag{8.89}$$

$$o_{i,K,\Delta t} \equiv 1 \Rightarrow \nu_{i,K,\Delta t} \leqslant \delta_i(K) + D_i \tag{8.90}$$

$$o_{i,K,\Delta t} \in [0, 1] \tag{8.91}$$

If Eq. (8.89) holds (with $o_{i,K,\Delta t} \equiv 1$), then the fixed-point iteration converges for this given Δt. The constraint is violated if the resource demand for instance K of task τ_i is greater than Δt within the time interval Δt, meaning that another eviction may take place in the worst case prior to τ_i finishing its execution. The

conditional notation (cf. Section 5.2.2.2) forces the binary variable $o_{i,K,\Delta t} \equiv 0$ in case that the constraint is violated. Otherwise, $o_{i,K,\Delta t}$ may be 0 or 1.

Eq. (8.90) asserts that task τ_i also finishes its execution within its deadline if a fixed-point was reached. Due to the same conditional formulation, the ILP solver is forced to set $o_{i,K,\Delta t} \equiv 0$ if this constraint does not hold.

If Eq. (8.89) is violated for all time intervals $\Delta t \leqslant \delta_i(K) + D_i$, then that instance K of τ_i is not schedulable. Testing time intervals beyond the task's deadline will obviously never lead to a valid schedule as, by definition, the task must finish *prior* to its deadline being missed. Therefore, for each $K = 1, \ldots, \hat{K}$, the maximum Δt which has to be tested equals $\delta_i(K) + D_i$.

As described above, these two constraints have to hold at least for one Δt for each instance K for the task to be schedulable. This is achieved by adding an additional constraint given in Eq. (8.92):

$$\sum_{\forall \Delta t} o_{i,K,\Delta t} \geqslant 1 \tag{8.92}$$

If not at least one of the $o_{i,K,\Delta t}$ variables holds, the system is not schedulable and the ILP is subsequently infeasible. Or, in other words, it must be ensured that for at least one Δt for each instance K, the resource demand is lower than the available resources. This is the analogon to the min term in the classical WCRT analysis (cf. Eq. (4.37)).

To model Eq. (4.36) within the ILP, the (in)equations presented above are modeled for each instance K individually. As Eq. (4.36) describes, the WCRT of the task is the maximum response time over all instances K which have to be analyzed. In the "normal" schedulability analysis, the WCRT can then be compared with the task's respective deadline. This is realized in the proposed ILP formulation by Eq. (8.90). Here, the maximum response time of each instance K of each task τ_i is compared with its respective deadline. If the deadline holds for each K, it obviously also holds for the maximum value over each $r(K, \tau_i)$. Therefore, the approach is *safe*. Accordingly, due to the fact that the WCRT is calculated using the response time of that instance K which has the maximum $r(K, \tau_i)$, performing the check for all other values does not introduce any additional pessimism.

Identically to the approach for dynamic-priority scheduling (cf. Eq. (8.70)), it must be ensured that the system load u does not exceed 1.

For the Example 8.3 from page 119, the formulations presented in this section can be applied as follows: All tasks' deadlines are larger than their minimal inter-arrival time. Therefore, a valid upper bound for the busy window is the hyper-period of the recurring activation patterns plus the maximum time interval for

the burst. By Definition 4.23, the hyper-period is the least common multiple of all activation periods within the task set Γ. This leads to:

$$\Delta T_B = 15 + \operatorname{lcm}(20, 30, 40) = 135 \tag{8.93}$$

From the combined density and interval functions η_Γ and δ_Γ which are depicted in Figs. 8.5 and 8.6, it can be seen that the minimal time intervals at which an event may occur in the system within the maximum busy window are:

$$\Delta t \in \{0, 5, 10, 15, 20, 30, 40, 60, 80, 90, 100, 120\} \tag{8.94}$$

$\Delta t \equiv 0$ may be skipped, as the resource demand of one instance of any task with non-zero WCET will always be greater than 0.

For task τ_0 from Example 8.3, the formulations presented in this section will result in:

$$v_{0,1,5} = 1 \cdot c_0 \tag{8.95}$$
$$o_{0,1,5} \equiv 1 \Rightarrow v_{0,1,5} \leqslant 5 \tag{8.96}$$
$$o_{0,1,5} \equiv 1 \Rightarrow v_{0,1,5} \leqslant 5 + 20 \tag{8.97}$$

Looking at Table 8.2 defining the timing properties of τ_0, the valid ranges of its WCET is $7 \leqslant c_0 \leqslant 9$. Obviously, the constraints above do not hold.

For $\Delta t \equiv 10$, this leads to:

$$v_{0,1,10} = 1 \cdot c_0 \tag{8.98}$$
$$o_{0,1,10} \equiv 1 \Rightarrow v_{0,1,10} \leqslant 10 \tag{8.99}$$
$$o_{0,1,10} \equiv 1 \Rightarrow v_{0,1,10} \leqslant 10 + 20 \tag{8.100}$$

In the ILP, this would be repeated for all remaining time intervals. However, it can be seen that the first instance of τ_0 already holds its deadline constraints. The constraints for the second instance would subsequently look like:

$$v_{0,2,5} = 2 \cdot c_0 \tag{8.101}$$
$$o_{0,2,5} \equiv 1 \Rightarrow v_{0,2,5} \leqslant 5 \tag{8.102}$$
$$o_{0,2,5} \equiv 1 \Rightarrow v_{0,2,5} \leqslant 10 + 20 \tag{8.103}$$
$$v_{0,2,10} = 2 \cdot c_0 \tag{8.104}$$
$$o_{0,2,10} \equiv 1 \Rightarrow v_{0,2,10} \leqslant 10 \tag{8.105}$$
$$o_{0,2,10} \equiv 1 \Rightarrow v_{0,2,10} \leqslant 10 + 20 \tag{8.106}$$
$$\dots$$

Similar to $K = 1$, this holds.

This scheme is repeated up to $\Delta t \equiv 120$ and subsequently $\hat{K} = \eta_0(120) = 9$ events of τ_0:

$$v_{0,9,5} = 9 \cdot c_0 \tag{8.107}$$

$$o_{0,9,5} \equiv 1 \Rightarrow v_{0,9,5} \leqslant 120 \tag{8.108}$$

$$o_{0,9,5} \equiv 1 \Rightarrow v_{0,9,5} \leqslant 115 + 20 \tag{8.109}$$

Similar to dynamic-priority scheduling, a final constraint which limits the maximum busy window dominates equation Eq. (8.108):

$$135 \geqslant v_{0,9,120} \tag{8.110}$$

Tasks τ_1 and τ_2 are processed identically. The only difference is that the nested sum term in Eq. (8.88) is additionally used in order to include the computational demand of higher priority tasks as well as eviction penalties. This was not needed for τ_0, as this is the task with the highest priority in the system and can thus not be preempted by any other task.

Calculating the eviction penalties is straightforward, as the density functions in Eq. (8.88) are only dependent on the currently modeled interval Δt and the tasks' deadlines. Thus, they can simply be pre-computed outside of the ILP and added as constant terms. To keep the example somewhat simple, eviction penalties $e_{i,j}$ are assumed to be 0. Modeling this ILP integer variable is described in the next section.

8.4. Accounting for Preemption Penalties

Previous sections in this chapter focused on the ILP model for schedulability-aware optimizations. Preemption penalties inflicted to a task τ_i by another task τ_j were simply modeled as an additional ILP variable $e_{i,j}$. This chapter discusses how these variables can be calculated.

It should be stressed that the preemption penalty $e_{i,j}$ is only used to model the *additional* timing penalty of *exactly one* interruption of τ_i by τ_j. The WCETs of the preempting tasks as well as the number of preemptions are modeled by the schedulability constraints presented in the previous sections. First, there are several sources for preemption delays which have to be considered separately:

- Execution time of the real-time scheduler.
- Cache-related delays due to cache line evictions by the preempting task.
- Constant timing overhead inflicted by hardware interrupts which trigger the preemption.
- Possibly variable costs for saving and restoring the current context (saving/restoring registers to/from stack, pipeline flush, ...)

The penalty inflicted by the execution time of the scheduler itself is explicitly *not* considered in the preemption overhead. Instead, the scheduler has to be modeled as a regular task in the system. Especially in case of dynamic-priority scheduling, the scheduler is executed even if there is (currently) no higher-priority task. The reason for this is that the scheduler itself simply *is* just another task which decides on whether the currently running "regular" task should be preempted or not. The upcoming Section 11.5 shows how a simple scheduler may be created automatically for a multi-task system, how it can be analyzed for its worst-case timing behavior and then be added to a task set for optimization and schedulability analysis.

Cache-related preemption delays play a major role when it comes to preemption penalties. They may vary significantly, depending on which task is preempted by which other task. Therefore, they can be modeled very effectively by the preemption delay variables. Due to the complexity of analyzing caches both for single-task systems and multi-tasking systems as part of the ILP model, describing cache-related preemption delays is discussed in detail in Chapter 9.

The remaining two factors inflicting on the preemption penalties are constant interrupt-driven overheads and possibly variable context saving costs. Interrupt delays can usually be either measured quite accurately for the given target platform, or safe upper bounds can be retrieved from the target's data sheet. In any case, as interrupt penalties can be assumed to be constant, they can simply be added as a constant to the eviction penalty variable:

$$e_{i,j} = P_{int} \tag{8.111}$$

with P_{int} being the constant timing delay inflicted by the system's interrupt handling.

Costs for saving and restoring the preempted task's context can be variable. In any case, a constant timing penalty can be accounted for which occurs due to the microcontroller's pipeline flush on executing a new task. This means that, on resume of the previously preempted task, instructions have to be re-fetched from memory in order to refill the previously flushed processing pipeline. This delay is target-dependent but can be analyzed prior to creating the ILP.

Some architectures may have hardware support for task context switches, leading to relatively small overhead. Other architectures, like, e.g., the very simple ARM7TDMI architecture, do not have such features. In any case, the preempting task (i.e., the scheduler) has to save the context of the running task (i.e., any registers which might be used by the task being preempted) and restore it before the running task is resumed. Irrespective of whether special hardware instructions are used for this or whether the registers are saved "manually", the according assembly instructions have to be explicitly present in the scheduler's code. This means that these costs are already accounted for in the regular WCET analysis and do not have to be accounted for separately.

The target hardware-specific capabilities for context savings are usually restricted. Care must be taken by the system developer that these capabilities are sufficient in a worst-case scenario. E.g., the Infineon TriCore TC1796 features implicit context savings. A dedicated `call` assembly instruction takes a function label as an argument and handles context saving (at least for half of the available registers) implicitly. Additionally, the architecture features so-called shadow register sets. This means that, as long as a free shadow register set exists, there is no timing delay due to saving registers to the stack. Instead, simply one of the free register sets is used by the newly called function. While this is very efficient in the general case, nested preemptions combined with nested function calls within each task make it hard if not impossible to predict whether the number of available shadow registers is sufficient or not.

We subsequently assume that a worst-case scenario has to be assumed for each preemption. Therefore, the analysis of the worst-case timing overhead due to context saving can also be analyzed outside of the ILP and be expressed as another constant value P_{con} within the ILP model. The eviction penalty is therefore extended to:

$$e_{i,j} = P_{int} + P_{con} \tag{8.112}$$

8.5. Limitations

The ILP-based schedulability-aware optimization framework presented in this chapter comes with the following limitations:

- Modeling the WCET variables c_i may be imprecise. This has been discussed in Section 7.3.3 for the single-tasking static SPM allocation. If the used WCET model only provides a very pessimistic estimate of the WCET of each task in the ILP, then the schedulability constraints will subsequently also provide a pessimistic estimate for the system's schedulability.
- As discussed in Section 4.5, a safe upper bound on the testing interval in a schedulability analysis of an arbitrarily triggered system is the hyper-period. As the hyper-period is the least common multiple of all periods present in the system, it may easily become very large. As a result, periods may have to be changed in the analysis, resulting in a smaller hyper-period but also in an increased pessimism. However, this issue is not specific to the proposed framework. Schedulability analyses themselves suffer from the same issue when analyzing systems with a high computational load.
- Finally, inter-task penalties may have to be over-approximated due to complexity reasons. This issue is very similar to having to over-approximate

the WCET in an ILP-based analysis framework. In the long run, the trade-off between ILP solving time and model complexity must be made by the user, depending on the concrete usage scenario.

Generally speaking, the constraint-based approaches presented in this chapter allow for a precise and exact modeling and inherent optimization of the schedulability of a multi-tasking system. In practice, the designer of a compiler optimization who wants to use the framework will have to make a choice between additional pessimism and increased runtime of the ILP solver.

9. Cache-Aware ILP Optimization

An embedded hard real-time system may feature caches in order to close the gap between the fast processing speed of the microcontroller and the relatively slow access times of traditional Flash memory. Usually, a cache hit leads to access times similar to an SPM access, whereas a cache miss may easily take 5 to 10 times longer. The transparent load and replacement strategy aims at providing a trade-off between fast access times and the limited size of fast memories. Due to the fact that loads and replacements into and out of the cache are transparent to the running system, using caches is simple from a programmer's point of view.

However, it significantly complicates WCET analysis and subsequently also WCET-focused optimizations. As discussed in Section 4.2.2.2, a WCET analysis must carefully analyze which memory accesses might result in a cache miss or hit and which accesses are safe hits or misses. An optimization which modifies the system's memory layout must try to model its impact on the cache hits and misses as closely as possible, in order not to mitigate its impact due to an unwillingly degraded caching behavior.

This chapter uses the previously presented example of an SPM allocation at basic block-level in order to show the applicability of the ILP optimization framework from Chapter 8 to systems using caches. The main issue which is to be shown is that the ILP is still capable of handling the additional complexity which is inflicted by modeling caching behavior within the ILP model.

The general ideas of modeling caches within the ILP framework are not restricted to any specific replacement policy or cache architecture. However, as discussed in Section 3.2.1, set associative caches with an LRU replacement policy are commonly used in embedded systems. This is also the cache architecture and replacement policy used by the Infineon TriCore microcontroller familiy which is used for evaluation in this thesis. This chapter therefore focuses on such set associative caches with LRU policy. Since the Infineon TriCore is a Harvard architecture, it is also not necessary to model data caches in order to show the integration of caches into the instruction SPM optimization and the ILP schedulability framework. The chapter will also therefore solely focus on *instruction* caches.

Section 9.1 gives a brief motivation on the subject. Section 9.2 shows how the actual cache parameters of a basic block (tag, index, offset) can be calculated within an ILP. Section 9.3 subsequently introduces how intra-task cache conflicts

can be modeled within the ILP formulation. The approach is then integrated into the single tasking SPM optimization from Chapter 7. This has previously been presented at the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES) [LKF16]. The main ideas and concepts have been derived in the course of this thesis. An approach on the implementation was later done as part of the master thesis by the co-autor of the aforementioned publication, Christina Kittsteiner.

The implementation was later substantially rewritten for publication and extended towards also supporting CRPD in case of multi-tasking systems. These extensions are presented in Section 9.4. Modeling CRPD effects within the ILP framework has not yet been published.

9.1. Motivation

When a WCET or schedulability-aware optimization inflicts any changes on the system's memory layout, cache behavior is likely to be changed and may both improve or worsen the system's timing behavior. When looking at the instruction SPM allocation previously discussed in Chapter 7, moving a basic block from Flash memory to SPM will leave a "gap" in the Flash. In practice, leaving such gaps is possible, but wastes the often very limited memory capacities of the system. Therefore, when moving a basic block to the SPM, all subsequent basic blocks will move up – thus, their addresses in memory change. Additionally, the jump correction which was discussed in Section 7.3.1 can increase the size of a basic block, thus again changing the position in memory of all subsequent basic blocks.

As discussed in Section 3.2.1, the cache position where a basic block is moved to in a set associative cache depends on its index. The index is directly retrieved from the basic block's address in memory. Thus, if the basic block's position in memory changes, its address changes, and therefore its location in the cache might change. This may completely change cache analysis, as the block will now evict different blocks from memory and may subsequently also be evicted by different blocks itself.

Apart from this, when moving a basic block to the SPM, the basic block is no longer loaded into the cache on access. Therefore, even when neglecting any changes in the address layout, moving a basic block to the SPM will change the caching behavior as fewer evictions may take place.

The following sections focus on how to model a system's caching behavior as part of the ILP optimization framework proposed in Chapter 8. This can then be used to evaluate the impact of this explicit cache modeling on the optimization results in the evaluation in Chapter 12 and compare it to the cache-unaware optimization.

9.2. Modeling Cache Properties

To be able to account for caching effects within the ILP, it is necessary to express to which cache line(s) each basic block maps to. This section bases on the following prerequisites:

1. It is assumed that there exists an ILP integer variable for each basic block i, which precisely expresses the start address of a basic block in memory. This variable is called $m_{i,0}$.
2. It is assumed that the size of each basic block i is also modeled by another ILP integer variable s_i.
3. A safe upper bound on the maximum size of each basic block i is known. This is denoted by S_i^{max}.

Using the start address and the size of the basic block, a new ILP integer variable can be introduced which models the end address $m_{i,E}$ of basic block i:

$$m_{i,E} = m_{i,0} + s_i \tag{9.1}$$

Depending on the start address and its size, a basic block might span over multiple cache lines. The ILP variables $m_{i,0}$ and $m_{i,E}$ can be used in order to model which cache lines a basic block occupies. Finally, this information is used in combination with the program's CFG to add inequations which model possible cache evictions.

9.2.1. Modeling the First and Last Cache Lines

To avoid ambiguities, it is assumed that a basic block will always be smaller than the net cache size. The net cache size is determined by the overall cache size divided by the cache's associativity. If a basic block is too large, it may be split into smaller parts for the optimization to comply with this requirement.

Fig. 9.1 shows how a given memory address is associated with a corresponding cache line, as discussed in Section 3.2.1. The bits α through $\beta - 1$ determine the range of the index which determines the cache line a memory block will be mapped to. For any given architecture, α, β and γ are known constants. The net size is determined by the sum of bits which are used for the index and the offset. As a result, the size of a basic block is therefore limited to $2^\beta - 1$ bytes. As basic blocks are usually not aligned with the cache boundaries, all memory addresses which belong to a given basic block will map to at most two different tags and $2^{\beta-\alpha}$ cache lines.

In normal programming, one can use shift instructions and Boolean or modulo operations to calculate the index for a given memory address. However, those

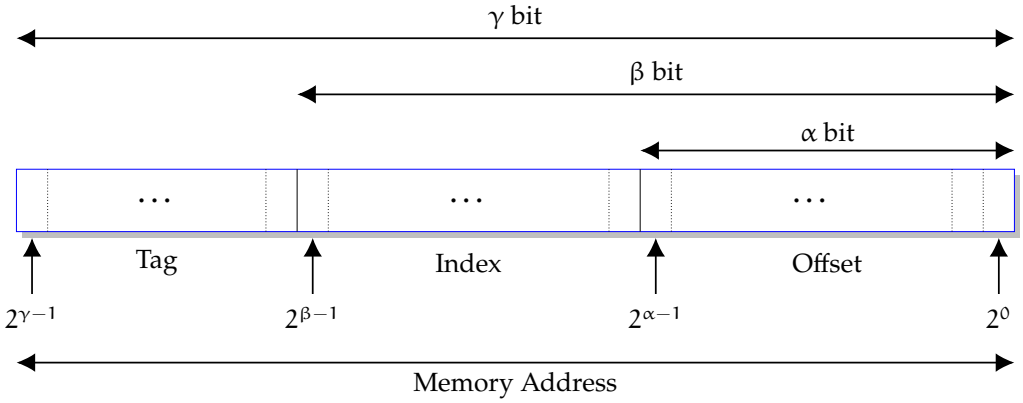


Figure 9.1: Mapping of a memory address with γ bits to its cache parameters [HP12, App. B] (cf. Section 3.2.1).

operations cannot be directly used in an ILP. Instead, they have to be transformed into linear inequations.

To solve this problem, an address $m_{i,\nu}$ which maps to the ν'th cache line of basic block i, is decomposed using its base number representation as shown in Section 5.2.4.

$$m_{i,\nu} = 2^0 \cdot o_{i,\nu} + 2^\alpha \cdot n_{i,\nu} + 2^\beta \cdot t_{i,\nu} \tag{9.2}$$
$$0 \leqslant o_{i,\nu} \leqslant 2^\alpha - 1 \tag{9.3}$$
$$0 \leqslant n_{i,\nu} \leqslant 2^{\beta-\alpha} - 1 \tag{9.4}$$
$$0 \leqslant t_{i,\nu} \leqslant 2^{\gamma-\beta} - 1 \tag{9.5}$$

$o_{i,\nu}$ is an ILP variable which will hold the cache offset of basic block i's address $m_{i,\nu}$. Accordingly, $n_{i,\nu}$ signifies the index and $t_{i,\nu}$ the cache tag.

By applying the equations to both start and end addresses $m_{i,0}$ and $m_{i,E}$, (in)equations are created which model the first and the last cache line occupied by each basic block i of the system.

$$m_{i,0} = 2^0 \cdot o_{i,0} + 2^\alpha \cdot n_{i,0} + 2^\beta \cdot t_{i,0} \tag{9.6}$$
$$m_{i,E_i} = 2^0 \cdot o_{i,E_i} + 2^\alpha \cdot n_{i,E_i} + 2^\beta \cdot t_{i,E_i} \tag{9.7}$$

Example 9.1 (Cache Index Modeling)
Consider a 32 bit architecture with Flash memory starting at address 0x80100000 *with a size of* 1 024 kB. *The architecture features a* 8 192 B *large cache. The cache is 2-way set*

associative with 5 *bits for the offset, and* 8 *bits for the index. As a consequence, there are* 19 *bit used for the tag.*

Consider an arbitrary basic block A *with start address* $m_{A,0}$ *and a maximum size of* $S_A^{max} = 36\,B$. *To avoid numerical issues in the ILP solver, basic block addresses are modeled relatively to the start of the memory section instead of using absolute memory addresses. As a result, the first basic block in Flash memory starts at address* 0. *The cache parameters stay unchanged.*

The cache parameters lead to:

$$\alpha = 5 \tag{9.8}$$
$$\beta = 5 + 8 = 13 \tag{9.9}$$
$$\gamma = 32 \tag{9.10}$$

The base number decomposition to determine the cache index $n_{A,0}$ *can be written as follows:*

$$m_{A,0} = 2^0 \cdot o_{A,0} + 2^5 \cdot n_{A,0} + 2^{13} \cdot t_{A,0} \tag{9.11}$$
$$0 \leqslant o_{A,0} \leqslant 2^5 - 1 = 31 \tag{9.12}$$
$$0 \leqslant n_{A,0} \leqslant 2^{13-5} - 1 = 255 \tag{9.13}$$
$$0 \leqslant t_{A,0} \leqslant 2^{31-13} - 1 = 262\,143 \tag{9.14}$$

For example, consider the start of basic block A *at* 0x8010100a. *Due to addressing the basic blocks relative to the section start, the address will result in* 0x000100a *in the ILP formulation. As a decimal number, this leads to* $m_{A,0} = 4\,106$ *and the end address* $m_{A,E} = 4\,142$.

This leads to a base number decomposition of:

$$4106 = 2^0 \cdot 10 + 2^5 \cdot 128 + 2^{13} \cdot 0 \tag{9.15}$$
$$\tag{9.16}$$

Therefore, the start of basic block A *is located in the cache line with index* 128. *Analogously, the end address* $m_{A,E}$ *of basic block* A *is:*

$$4142 = 2^0 \cdot 14 + 2^5 \cdot 129 + 2^{13} \cdot 0 \tag{9.17}$$

Thus, $n_{A,E} = 129$.

9.2.2. Calculating Used Cache Lines

With the maximum size S_i^{max}, the basic block i will occupy at most $\left\lceil \frac{S_i^{max}}{2^{\alpha-1}} \right\rceil + 1$ cache lines. The additional cache line stems from the fact that the start of a basic

block might not be aligned with the start of a cache line. Therefore, even a tiny basic block might reside in two different cache lines.

For Example 9.1, the basic block has a size of 36 B. Therefore, it may occupy at most 3 cache lines, depending on its start address. $E_i^{max} + 1$ denotes the maximum number of cache lines which may be occupied by a given basic block i. Then, E_i^{max} can be written as:

$$E_i^{max} = \left\lceil \frac{S_i^{max}}{2^{\alpha-1}} \right\rceil \tag{9.18}$$

The general idea of modeling all used cache lines is as follows: An upper bound of the maximum number of cache lines occupied by a basic block is known prior to creating the ILP. Subsequently, the ILP contains $E_i^{max} + 1$ integer variables $n_{i,\nu}, \nu = 0, \ldots, E_i^{max}$. Each of these variables will hold the index of the ν'th occupied cache lines of basic block i. In case that basic block i actually uses fewer cache lines, the last cache line variable will hold the same value.

Anagolously to $m_{i,0}$ modeling basic block i's start address and $m_{i,E}$ modeling its end address, ILP integer variables modeling the corresponding cache indices can be defined:

$n_{i,0}$ describes basic block i's first and $n_{i,E}$ its last cache line. Due to the fact that E_i^{max} is known prior to creating the ILP, additional ILP integer variables can be created for each cache line which may be occupied by each basic block i: $n_{i,k}, k = 1, \ldots, E_i^{max}$ denotes the index of the k'th occupied cache line of basic block i. Block offsets are not needed for the cache hit/miss analysis. Therefore, they can be skipped and modeling an address in the ILP can start at index boundaries. Subsequently, the lowest bit describing the index (2^α, cf. Fig. 9.1) may be modeled by 2^0 instead of 2^α in the ILP model. The start of the cache's tag field is subsequently moved to $2^{\beta-\alpha}$. This significantly lowers the range of the resulting ILP integer variables, reducing the risk of numerical issues in the ILP solver.

The basic block's start index $n_{i,0}$ and tag $t_{i,0}$ can be calculated as shown in the previous section. To simplify the ILP calculation, all subsequent indices $k, k > 0$ and tags are calculated by incrementing the preceding index by one:

$$2^{\beta-\alpha} \cdot t_{i,k-1} + n_{i,k-1} + 1 = 2^{\beta-\alpha} \cdot t_{i,k} + n_{i,k} \tag{9.19}$$

$$k \in [1, \ldots, E_i^{max} - 1] \tag{9.20}$$

To ensure that $n_{i,k}$ and $t_{i,k}$ are not larger than the physical cache properties allow, the variable bounds from Eqs. (9.4) and (9.5) are applied accordingly. Due to the fact that a basic block's size and its memory address are integer variables within the ILP, the actual number of occupied cache lines is not known when creating the ILP.

Thus, it may be that $n_{i,E^{max}}$ and $t_{i,E^{max}}$ denote cache lines which are not actually used by the basic block. This can be tackled, as the last occupied cache line parameters $n_{i,E}$ and $t_{i,E}$ can be calculated as shown in the previous section. Whether the k'th cache line is actually occupied by a basic block can therefore be expressed as:

$$(t_{i,E} > t_{i,k}) \Rightarrow a_{i,k} = 1 \tag{9.21}$$
$$(t_{i,E} = t_{i,k}) \wedge (n_{i,E} > n_{i,k}) \Rightarrow a_{i,k} = 1 \tag{9.22}$$
$$(t_{i,E} < t_{i,k}) \Rightarrow a_{i,k} = 0 \tag{9.23}$$
$$(t_{i,E} = t_{i,k}) \wedge (n_{i,E} \leqslant n_{i,k}) \Rightarrow a_{i,k} = 0 \tag{9.24}$$

Eq. (9.21) forces the binary variable $a_{i,k}$ to one if the k'th tag is lower than the tag of the basic block's end. This is the case if a wrap around occurs in the occupied cache lines. Therefore, the current line must always be used by the basic block.

If the tags are equal (Eq. (9.22)), the line is only actually being used, if its index is lower than the index in the last possibly occupied line. This is expressed by the second part of the logical AND conjunction.

Finally, in case that the index values are equal, that line has already been considered. In this case, $a_{i,k}$ is forced to 0 (Eq. (9.24)). To cover all possible cases, Eq. (9.23) forces $a_{i,k}$ to 0 in case that the tag exceeds the tag basic block's end. As a consequence, the binary variable $a_{i,k}$ expresses whether the cache line modeled by the ILP variables $n_{i,k}$ for the index and $t_{i,k}$ for the tag is actually being used by the basic block i.

Because the basic block's minimum and maximum size is known, the conditional constraints to determine whether an ILP variable models an actually used cache line or not has not to be modeled for all k, as the maximum variance of occupied cache lines can easily be calculated. This was considered in the implementation of this approach. However, for the sake of simplicity, this is not reflected in the upcoming sections. Therefore, if the ILP is built strictly following these descriptions, it is slightly more complex than it could be, yet obviously still safe (as the conditionals in Eqs. (9.21) to (9.24) will always result in an a variable value of 1).

Example 9.2 (Cache Line Calculations)
Following up on Example 9.1, basic block A can occupy at most 3 cache lines, depending on the value of the start address variable $m_{A,0}$. As a consequence, $E_A^{max} = 2$. Therefore, three ILP integer variables are defined for modeling the cache index:

$$n_{A,\nu}, \nu \in \{0, 1, 2\} \tag{9.25}$$
$$\tag{9.26}$$

$n_{A,0} = 128$ *and* $n_{A,E} = 129$ *have already been calculated in Example 9.1. Using Eq.* (9.19)*,* $n_{A,1}$ *can be calculated as follows:*

$$2^{13-5} \cdot t_{A,0} + n_{A,0} + 1 = 2^{13-5} \cdot t_{A,1} + n_{A,1} \tag{9.27}$$

$$0 \leqslant n_{A,1} \leqslant 255 \tag{9.28}$$

This leads to:

$$2^8 \cdot 0 + 128 + 1 = 2^8 \cdot 0 + 129 \tag{9.29}$$

Therefore, the second occupied cache line $n_{A,1}$ *is* 129*. If proceeded accordingly for* $\nu = 2$*, this leads to* $n_{A,2} = 130$ *and* $t_{A,2} = 0$*. As it can be seen,* $n_{A,1} \equiv n_{A,E}$ *and* $n_{A,2}$ *actually describes a cache line which is not occupied by the basic block.*

When using Eqs. (9.21) *to* (9.24)*, the binary variables* $a_{A,1}$ *and* $a_{A,2}$ *are subsequently both forced to* 0*:*

$$(t_{A,E} > t_{A,1}) \Rightarrow a_{A,1} = 1 \tag{9.30}$$

$$(t_{A,E} = t_{A,1}) \wedge (n_{i,E} > n_{i,k}) \Rightarrow a_{A,1} = 1 \tag{9.31}$$

$$(t_{A,E} < t_{A,1}) \Rightarrow a_{A,1} = 0 \tag{9.32}$$

$$(t_{A,E} = t_{A,1}) \wedge (n_{i,E} \leqslant n_{i,k}) \Rightarrow a_{A,1} = 0 \tag{9.33}$$

As $t_{A,E} \equiv t_{A,1}$*, the first and third constraints do not match. Because* $n_{A,E} \equiv n_{A,1}$*, the last equation matches and subsequently* $a_{A,1} \equiv 0$*. For* $a_{A,2}$ *the same conditionals match.*

Therefore, in conclusion, only the cache lines described by $n_{A,0}$ *and* $n_{A,E}$ *have to be accounted for when modeling cache conflicts within the ILP.*

9.3. Identifying Intra-Task Cache Conflicts

Determining intra-task cache conflicts is done in two stages: First, prior to building the ILP, a control flow analysis is performed in order to identify the basic blocks which have to be considered in the cache conflict analysis. Second, for each of these basic blocks, ILP constraints are created to model possible cache evictions. The result will be a Boolean ILP variable $h_{i,k}$ which is forced to 1 if the k'th cache line occupied by basic block i may be evicted between two consecutive executions of basic block i. This variable may then be used by any ILP-based optimization in order to model additional penalties in case of a cache miss.

9.3.1. Identifying Interfering Basic Blocks

Section 4.5.3 introduced the terms Useful Cache Block (UCB) and Evicting Cache Blocks (ECB) which will be reused in the following: A UCB is a basic block, which,

if evicted from cache, can lead to a timing penalty. An ECB is the basic block which may evict a UCB from cache.

In a program's control flow, a cache conflict between two blocks can only occur if the first block is useful, i.e., it is executed more than once and the second block might be fetched from cached memory between two consecutive executions of the first block. If the first condition is not fulfilled, evicting the first block from cache will not lead to any timing penalties, as the block is never needed again. If the second block is not fetched between two consecutive executions of the first block, it can obviously not evict it from memory. Therefore, cache conflicts are typically created either by instructions within a (possibly nested) loop or by code of a function which is called within a loop, therefore possibly evicting code from the loop. Additionally, the instructions of a given function may be evicted by instructions which are executed between two subsequent calls of this function.

A basic block i is said to be in conflict with another block j, if j may be executed between two subsequent executions of i. In this case, mapping both blocks to the same cache line could lead to a cache miss for i. The set of all blocks which are in conflict with i is denoted by $\mathcal{C}_i$. Due to the significant computational complexity, this approach only considers blocks in loops and nested loops within each function separately in the ILP's conflict analysis.

Context and infeasible path analysis on the CFG as well as global conflict analysis across different functions may provide even better solutions. However, they also come at the cost of further increasing the ILP's complexity and solving time. Therefore, this approach does not consider them so far.

9.3.2. Modeling Cache Conflicts

Once the set of conflicting basic blocks $\mathcal{C}_i$ for each useful basic block i has been determined, binary ILP variables are introduced comparing each cache line k which might be occupied by basic block i with each cache line k' of basic block $j \in \mathcal{C}_i$:

$$(n_{j,k'} = n_{i,k}) \wedge (t_{j,k'} \neq t_{i,k}) \Rightarrow q_{i,j,k,k'} = 1 \tag{9.34}$$

This enforces the ILP variable $q_{i,j,k,k'} \equiv 1$ in case that both basic blocks share the same cache index but have different cache tags. The comparison of the tags is necessary to avoid cache conflicts of subsequent basic blocks which actually share the *identical* memory chunk. In this case, basic block i even benefits, as the memory block needed by basic block i may be preloaded into the cache when basic block j is accessed.

Because the approach targets caches with LRU replacement policy, a basic block may be evicted from the cache in a worst-case scenario if the number of conflicting

blocks reaches the cache's associativity N. For example, a single basic block j can never evict parts of i from a 2-way set associative cache (due to the limitation of the basic block's size to not exceed the size of the cache). However, if two blocks j and k are both in conflict with i and all blocks map to the same cache lines, i may be evicted between two consecutive executions. Therefore, all cache conflicts are summed up for each cache line k of each basic block i and are compared with the cache's associativity N. The binary variable $h_{i,k}$ is subsequently forced to 1 if the number of conflicting blocks is equal to or bigger than the cache associativity N:

$$\left(\sum_{j \in \mathcal{C}} \sum_{k'=0}^{E_j^{max}} q_{i,j,k,k'} \geqslant N \right) \Rightarrow h_{i,k} = 1 \tag{9.35}$$

where E_j^{max} is the maximum number of cache lines which may be occupied by j.

9.3.3. Integration into a Static Instruction SPM Allocation

For an integration of the ILP formulations presented above, the following steps have to be performed:

- The maximum size of a basic block i, S_i^{max} must be known. This can be trivially retrieved from the underlying compiler framework.
- Each basic block i's start address $m_{i,0}$ and its size s_i must be calculated in order to determine the occupied cache lines. This has been shown in the previous sections.
- The constant WCET C_i of each basic block i must be calculated in a sensible, cache-aware, way. As long as the target architecture does not suffer from memory-related timing anomalies (cf. Section 4.2.2.2), it can be assumed that C_i expresses block i's execution time in case of a cache miss. Then, if the ILP constraints predict a safe cache hit, (i.e., $h_{i,k} \equiv 0$), a timing gain G^{Hit} is subtracted from a basic block's net WCET variable c_i.
- A safe estimate on the gain G^{Hit} achieved by one cache hit must be given. This is platform-specific and has to be retrieved from the target platform's technical specification.
- It must be ensured that the gain by a cache hit is only subtracted from i's WCET, if the basic block is actually allocated to cached memory. Especially, this means that the basic block must not be allocated to the SPM ($x_i \equiv 0$).

In order to keep the complexity of the resulting ILP manageable, conflicts are modeled at function-level only. I.e., cache hits due to repetitive calls to the same function are not modeled. However, of course, if a function `foo()` may be called

between two subsequent executions of a basic block i which belongs to function `func()`, then these conflicts will be handled.

The maximum size S_i^{max} of a basic block is known before formulating the ILP by summing its net size and the maximum size of additional jump correction instructions. This was earlier discussed in Section 7.3.1. When covering the jump correction for the SPM allocation optimization, the actual size of a basic block was also already modeled in the ILP. This was previously needed in order to prevent the ILP from assigning more basic blocks to the SPM than would fit into it. Therefore, s_i can also be considered as given.

If a basic block i is assigned to the SPM (i.e., $x_i = 1$), the block's memory addresses are not relevant for this optimization, as the block will obviously neither cause nor suffer from a cache eviction. If $x_i = 1$, then the memory address will not be used for modeling the block's actual execution time, therefore the ILP solver may set $m_{i,0}$ and m_{i,E_i} to an arbitrary value and thus circumvent cache conflicts. This can be achieved by conditionally enabling the constraints from Section 9.2.1 which restrict $m_{i,0}$ and m_{i,E_i}.

We define the rank of a basic block by its position in memory relative to other basic blocks, if all blocks are assigned to Flash memory. A basic block j has a lower rank than i if its address in memory is lower. With $m_{i,0}$ being the start address of a basic block i, $\mathcal{L}_i$ is the set of basic blocks which have a lower address in memory if they are not moved to the SPM:

$$\mathcal{L}_i = \{j | m_{j,0} < m_{i,0}\} \tag{9.36}$$

When a basic block is moved from Flash to SPM, the relative order of all basic blocks which remain in the Flash memory is not changed. As a result, for each given basic block i which resides in Flash, i's start address is determined by all those blocks which also stay in Flash memory and have a lower rank. With this information, the start address $m_{i,0}$ of each block i as well as its end address m_{i,E_i} can be modeled using integer-linear programming:

$$(x_i = 0) \Rightarrow m_{i,0} = \sum_{j \in \mathcal{L}_i} s_j \cdot (1 - x_j) \tag{9.37}$$

$$(x_i = 0) \Rightarrow m_{i,E_i} = m_{i,0} + s_i \tag{9.38}$$

Eq. (9.37) is not linear, as a binary decision variable is multiplied with an integer variable. However, this can be tackled by substituting the term by another integer variable which is conditionally enabled by the binary variable x_j. This way, both start and end addresses of each basic block i can precisely be modeled.

Calculating safe WCETs C_i for each basic block i is straightforward. To ensure a safe over-estimation despite the basic blocks' changing memory addresses, for

all basic blocks' WCET, the WCET in case of a cache miss has to be used. This can usually be obtained by configuring the WCET analysis tool which is used for the initial WCET analysis accordingly.

Then, the gain for a cache hit G^{Hit} has to be subtracted from each basic block i's WCET variable c_i for each set which is a safe cache hit. Therefore, Eq. (7.35) which bounds the WCET of a basic block i is extended as follows:

$$v_i = 1 - x_i \tag{9.39}$$

$$y_{i,k} = 1 - h_{i,k} \tag{9.40}$$

$$c_i = C_i^{Flash} - x_i \cdot G_i - \sum_{k=0}^{E_i^{max}} \left((y_{i,k} \wedge v_i) \cdot G^{Hit} \right) \tag{9.41}$$

A cache hit can only be accounted for, if basic block i is *not* allocated to the SPM and it does definitely *not* conflict with another basic block. Eq. (9.39) and Eq. (9.40) therefore negate the binary decision variable x_i which is 1 if i is allocated to the SPM, and $h_{i,k}$ which is 1 if a conflict may occur. The logical AND in Eq. (9.41) then ensures that the gain is only subtracted if both conditions apply.

For a coarse WCET optimization, this may already be sufficient. However, this approach may underestimate the WCET of a basic block i, since the initial cache miss on the very first execution of the basic block is no longer considered. This can be tackled by adding these costs to a safe predecessor of basic block i which is only executed once. E.g., this can be added to the overall cost of the optimized function c_{func} given that i is part of `func()`.

It should be mentioned that – in any case – the presented approach does not model the LRU caching behavior precisely. Modern microcontrollers like the Infineon TriCore fetch memory blocks in larger chunks, thus a cache miss may be transparent for the processor, as the fetch unit implicitly prefetched a certain memory block. Additionally, the cache hit/miss analysis in the ILP is not precise. Both all possibly conflicting basic blocks, and all occupied cache lines by each basic block are compaired pair-wise. As a result, the number of constraints grows quadratically both with the number of possibly conflicting basic blocks and with the number of lines that a basic block may occupy. The complexity of solving an ILP grows exponentially with the number of constraints. Therefore, any additional precision is bought by immense complexity of the resulting ILP. Therefore, in practice, a suitable ILP model may under- or over-approximate cache miss behavior. Additionally, due to this massive complexity, the conflict analysis is quite basic compared to the current state of the art when it comes to WCET analysis.

However, the model provides a good approximation of the caching behavior which basically points the ILP solver into the right direction to determine which basic blocks should be assigned to the SPM. For safe and tight WCET estimates, a

dedicated WCET analyzing tool like AbsInt `aiT` [FH04] should be used on the optimized code to get safe WCET estimates of the optimized task.

9.4. Inter-Task Cache Optimization

Sophisticated CRPD analysis techniques as previously described in Section 4.5.3 cannot be directly applied within the ILP-based optimization. The key issue is that for the analysis of UCBs and ECBs, the cache mapping and thus the addresses of the basic blocks in memory must be known. However, as seen in the previous section, memory addresses change in the course of a compiler optimization. Yet, the effects of CRPD can be modeled in an ILP formulation as shown in the following.

9.4.1. Identifying Useful and Evicting Basic Blocks

To identify UCBs of a preempted task τ_i, a standalone WCET analysis is performed for each task in isolation by a regular WCET analyzer like, e.g., `aiT`. The results of this analysis are used to identify which basic blocks of τ_i may have gains due a cache hit if executed without any preemptions by another task j. These basic blocks form the set of UCBs of τ_i.

In a worst-case scenario, the current task τ_i is preempted right after fetching a new memory block. Thus, the basic block which is executed at the time of the preemption by another task τ_j must always be considered as useful. Unfortunately, that basic block is not known as it is not known at which position task τ_i is preempted.

In order to restrict the number of ECBs of a preempting task τ_j prior to formulating the ILP, both the addresses of all UCBs of the preempted task τ_i and the addresses of the basic blocks of τ_j would have to be known. However, as already mentioned above, these addresses are subject to change in the course of the code optimization. Therefore, *any* basic block of a preempting task τ_j must be considered to be a potential ECB. For fixed-priority scheduling, the basic blocks of lower-priority tasks can obviously be excluded, as they will not be able to preempt τ_i. For dynamic-priority systems, all other tasks must be taken into consideration as any task may preempt any other task at some point during the system's runtime.

9.4.2. Modeling Cache Conflicts

In order to model whether a cache conflict occurs, Eq. (9.34) has to be applied for each UCB and potential ECB. As a result, the variable $q_{m,n,k,k'}$ models whether a potential ECB n of τ_j is actually conflicting with the useful block m of task τ_i due

to a mapping to the same cache line. For intra-task cache conflicts, Eq. (9.35) was subsequently applied in order to account for a cache conflict only if the number of ECBs between two subsequent accesses to the given UCB m may equal or exceed the cache's associativity.

However, for inter-task conflicts, this is not that simple. An associativity-aware cache model would not only have to account for the intra-task age of a given basic block in the cache, but also be aware of any possible nested preemptions. Modeling this as part of the ILP would introduce a massive complexity, as the memory addresses of all basic blocks are considered variable. Therefore, for CRPD-aware optimizations it is assumed that any conflict might always result in a cache miss. In analogy to Eq. (9.35), this leads to:

$$q_{m,n,k,k'} = h^{\text{crpd}}_{i,j,k} \tag{9.42}$$

In summary, if the useful cache line k of task τ_i may be evicted from memory if τ_i is preempted by task τ_j, then $h^{\text{crpd}}_{i,j,k} \equiv 1$. Otherwise, it is 0. Using this indicator variable, the eviction penalty $e_{i,j}$ can be extended accounting for CRPD analogously to Eqs. (9.39) and (9.41):

$$\nu_i = 1 - x_i \tag{9.43}$$

$$e_{i,j} = P_{\text{int}} + P_{\text{con}} + \sum_{k=0}^{E_i^{\max}} \left(\left(h^{\text{crpd}}_{i,j,k} \wedge \nu_i \right) \cdot P_{\text{miss}} \right) \tag{9.44}$$

As in Eq. (9.39), ν_i simply negates the SPM decision variable x_i. In Eq. (9.44), the eviction penalty variable $e_{i,j}$ is extended by the CRPD which may be inflicted to τ_i by τ_j. This is done by iterating over all useful cache lines k of τ_i. If a conflict exists for this cache line, then an additional penalty is added to $e_{i,j}$. P_{miss} models the timing penalty of one additional cache miss. The value of P_{miss} is a target platform-specific constant which must be retrieved from the corresponding technical documentation.

10. Genetic Algorithm for Schedulability-Aware Optimizations

Due to the exponential worst-case solving complexity of ILP, previous chapters introduced several simplifications in order to cope with the immense complexity of the underlying optimization problem. Therefore, despite the fact that ILP per se is able to provide an optimal solution, this solution is only as good as the underlying mathematical model.

Genetic algorithms cannot *guarantee* to return an optimal solution or even to find a valid solution at all. However, they have proven to be suitable for many types of large combinatorial problems in practice. This chapter therefore introduces a genetic algorithm as an alternative to the ILP-based approach. This approach can then be used to compare the ILP-based approach – both with regard to its capability of finding a solution which leads to a schedulable system and with regard to the time needed to optimize the system.

Since being introduced by Goldberg in 1989 [Gol89], genetic algorithms have become an important tool when it comes to solving complex combinatorial problems. The main idea behind a genetic optimization is to randomly generate an initial population of possible solutions. Then, the best individuals of this population are transferred on to the next so-called generation. The remaining slots in the new generation are filled by recombining individuals from the preceding generation. Additionally, some random permutations are performed on the newly created individuals. This is meant to represent genetic recombination and mutation. If the resulting individual is considered dysfunctional or invalid with regard to some user-defined design constraints, it may either be discarded or repaired by a custom repair function. Finally, the resulting individual is analyzed and its quality, or "fitness" is evaluated. Inspired by Darwin's "survival of the fittest", the individual is then either kept as a new parent or dropped. This is repeated until either no improvement can be made over a pre-defined number of generations, or a user-defined timeout is reached.

If the algorithm's parameters for recombination and mutation probabilities are chosen carefully, many problems tend to converge quickly towards their global

optimum using genetic optimization approaches. To evaluate the quality of genetic algorithms on schedulability-aware code optimizations, and to allow for putting the evaluation results of the previously presented ILP approach into context, this chapter is going to introduce a schedulability-aware SPM optimization based on a genetic algorithm. The optimization was previously presented for single-tasking multi-core optimizations for the ARM7TDMI architecture in [OLF17]. It is adapted to cover single-core multi-tasking systems in the following sections.

Section 10.1 will elaborate on how the initial population is created. Section 10.2 proceeds by discussing the approach on recombining two individuals. Afterwards, Section 10.3 shows how an individual is mutated. For a combinatorial optimization problem under resource constraints like, e.g., a limited SPM size, the resulting individuals may easily be invalid. Section 10.4 therefore tackles the issue of repairing an invalid individual. Finally, Section 10.5 shows how the fitness of an individual may be estimated.

10.1. Initial Population

Finding a good initial population is crucial for a fast convergence of the genetic algorithm. As in traditional biological genetics, combining multiple very similar individuals will not tend to give major changes to the resulting individual. In this case, any real progress with regard to finding a close to optimal solution would rely almost solely on the mutation process.

To generate as diverse but still reasonable initial individuals as possible, this work uses the following approach:

- The first individual is not using any SPM at all initially, i.e., it is identical to the unoptimized task set.
- The second individual is completely assigned to the SPM. Of course, this individual will probably not work, as the SPM is not large enough to hold the whole task set (if it is, finding the optimum assignment would be trivial). As a result, the repair function which is elaborated on in the upcoming in Section 10.4 is used to repair this individual.
- For all other individuals, each block is assigned to the SPM with a probability of 0.5. As with the second individual, the repair function is used to make sure that each individual is valid.

The algorithmic representation of the instruction SPM assignment on a basic block-level is straightforward: An arbitrary but fixed list of all basic blocks of the task set is inherently created by the compiler framework. Then, a genome of same length as this list is created, consisting of binary variables. Each entry in

this genome denotes whether the basic block at the corresponding position will be assigned to the SPM or not.

10.2. Recombination

Recombination of two individuals is performed using one point-crossover [Gol89, p. 12]. Given two individuals A and B, a crossover position in the list of basic block assignments is selected randomly.

Then, all decision variables of individual A which are behind this randomly selected position are swapped with the decision variables of B from these positions. To ease descriptions, when talking about any "random" selection within this chapter, it is referred to a uniformly distributed random selection unless explicitly stated otherwise. The random source is considered to not have a memory.

10.3. Mutation

After the recombination, one or multiple entries in the decision vector may be mutated randomly. To achieve this, two kinds of mutation were implemented which can be selected by the user: one-bit and multi-bit mutation.

10.3.1. One-Bit Mutation

For one-bit mutation, one entry in the decision vector is chosen randomly. This entry is then flipped with a probability of p which can be defined by the user.

10.3.2. Multi-Bit Mutation

Multi-bit mutation is used to mutate several entries of the solution vector of an individual. This process is two-staged:

Given a solution vector of length N, firstly, the maximum number M, $M \in [0, \ldots, N-1]$ of entries which are mutated is selected randomly.

Secondly, a random entry E, $E \in [0, \ldots, N-1]$ is selected M consecutive times, and each time, this entry E is flipped with a user-specified probability of p.

As a result, this means that in a solution vector of length N, a solution may be flipped at most N times. Additionally, it is theoretically possible that by chance the same entry is chosen twice and also toggled twice, thus although mutations took place, the mutated individual is unchanged afterwards.

In order to find good settings for the mutation algorithm, both one-bit and multi-bit mutations were evaluated for single-tasking systems. The results are shown in Appendix B.1. Based on this evaluation, a one-bit mutation with a mutation probability of $p = 0.1$ was chosen.

10.4. Repair Function

For the instruction SPM allocation, an individual is invalid if more elements are assigned to the SPM than there is space available. However, an invalid solution may in fact be near-optimum. As a result, a repair function is used to remove blocks from the SPM until the basic blocks assigned to the SPM fit into the physically available memory.

To achieve this, in a first step, the solution vector is applied to the actual tasks and all blocks with an according value in the solution vector are assigned to SPM without accounting for the needed space. Then, the functional correctness of the task set is restored by applying the control flow correction techniques described in Section 7.3. Finally, the theoretically needed size of the SPM memory region is calculated.

If the task set fits into the physically available memory, no corrections are performed and the repair function terminates.

Otherwise, the repair algorithm is called: Blocks in SPM are randomly selected to be removed from SPM without re-applying control flow corrections, until at most a pre-defined percentage of the SPM are still occupied. E.g., for 50 %, half the SPM is emptied if an individual causes an overfull SPM. For 100 %, the SPM is only emptied such that it is hardly not overfull anymore. In any case, the algorithm will always remove at least one block from SPM. Obviously, this may degrade the achieved results. However, for large benchmarks, running the control flow correction routines is quite time-consuming and thus heavily decreasing the genetic algorithm's performance – especially for larger task sets with a relatively small SPM memory.

Once the repair algorithm finishes, the control flow is corrected once again, and it is tested whether the program adheres to memory size constraints. If not, e.g., due to a very unfavorable SPM allocation leading to extremely high jump correction costs, the repair algorithm is called again iteratively until the task set actually fits into the SPM. Due to the fact that each run of the algorithm removes at least one block from SPM, this approach is guaranteed to terminate with a valid individual.

Different removal ratios from 50 % to 100 % were evaluated. The results are shown in Appendix B.2. The results show that a ratio of 80 % leads to notably decreases in runtime without any notable decrease in the resulting WCETs. For

different architectures or optimization scenarios, this parameter might have to be tuned by the user.

Note, that a "valid" individual will not necessarily fulfill all timing constraints of the system. Instead, the adherence to timing requirements is evaluated using the fitness function which is discussed in the upcoming section.

10.5. Fitness Function

In a multi-tasking environment, the goal is to assert that all response times are smaller than the tasks' respective deadlines. From a timing analysis view, a real-time system is either broken if at least one task might miss its deadline. Or, it is considered working if all tasks will provably meet their timing constraints. In this second case, the optimization can be stopped as soon as a sufficiently well solution has been found. As a result, a naturally chosen fitness function for the genetic algorithm is binary: 0 if the system is schedulable, or 1 if it is considered broken.

Obviously, this measure is not suitable for a genetic algorithm, as it cannot distinguish which of two broken solutions is "better" or "worse". As a result, only a purely random selection between two individuals can be performed and the genetic algorithm degrades to a purely randomized search of the valid design space. To counter this issue, two sensitivity analyses were developed as part of this thesis to provide a good measure on "how invalid" a given non-schedulable system is: One task removal heuristic based on the ILP model presented in Chapter 8 and one based on a constant scaling factor.

10.5.1. Task Removal Heuristic

The basic idea is to calculate the number of tasks which have to be removed from the system in order to achieve schedulability. Therefore, in a system with N tasks, the sensitivity analysis would return 0 in case that the system is schedulable and all tasks always safely meet their respective deadlines. On the other hand, the analysis will return N in a worst-case scenario where even one single task of the task set cannot be scheduled without missing its deadline.

The optimal analysis is based on the ILP model provided in Chapter 8. The schedulability constraints presented in Section 8.3 are subsequently used to analyze arbitrarily triggered task sets. Since, however, the sensitivity measure does not perform an actual optimization, the CFG model of each task as presented in Chapter 7 is not needed.

As discussed in Chapter 8, each task τ_i's runtime behavior is denoted by one single ILP integer variable c_i. In order to turn the schedulability-aware optimization framework into a sensitivity analysis, c_i is modeled as follows:

$$c_i = b_i \cdot C_i \tag{10.1}$$

C_i is the constant WCET of the whole task τ_i of the individual whose fitness is currently being analyzed. This WCET can be obtained by any WCET analyzer tool like, e.g., AbsInt `aiT`. b_i is a binary decision variable. The value of b_i is not further restricted and may thus be chosen freely by the ILP solver. Therefore, by setting $b_i \equiv 0$, the ILP solver can basically choose to remove a task from the task set.

Context switching penalties $e_{i,j}$ can be modeled accordingly, leading to a context switching penalty of 0 if $b_j \equiv 0$, i.e., the preempting task τ_j is being removed by the ILP solver. Again, since the fitness analysis does not perform any actual optimization, the actual values for any context switching penalties can be calculated outside of the ILP.

For a task set Γ, the ILP's objective function is subsequently set to

$$\max \sum_{i \in \Gamma} b_i \tag{10.2}$$

This way, the ILP solver will try and remove as few tasks as possible while the schedulability constraints from Section 8.3 will ensure that the result will be a schedulable system. This approach works for both fixed-priority and EDF constraints which were previously described.

As an additional benefit, if needed, the system designer may check the ILP solver's solution in order to find out which tasks have to be removed in order to achieve a schedulable system with the current WCETs.

The approach presented in this section allows for a non-binary fitness function: An individual i is considered fitter than an individual j if fewer tasks have to be removed in order to achieve schedulability. Unfortunately, this measure is still very coarse-grained. In a system with I tasks, each task may consist of many basic blocks. However, only I values are available for the fitness function. Therefore, in case that the same number of tasks is to be removed from two task sets, that task set for which the remaining system load is lower is considered to be fitter. This allows for a fine-grained fitness function. Formally, the relative fitness of two individuals can be given as follows:

Definition 10.1 (Relative fitness of two individuals)
Given two individuals with the task sets Γ_1 and Γ_2 respectively, the individual with the task set Γ_1 is considered to be fitter than the one with Γ_2, if fewer tasks must be removed from Γ_1 than from Γ_2 in order to achieve schedulability.

If the number of tasks to be removed is identical, Γ_1 *is fitter, iff*

$$\sum_{i \in \Gamma_1} \lim_{\Delta t \to \infty} \left[\frac{C_i \cdot \eta_i(\Delta t)}{\Delta t} \right] \cdot b_i < \sum_{j \in \Gamma_2} \lim_{\Delta t \to \infty} \left[\frac{C_j \cdot \eta_j(\Delta t)}{\Delta t} \right] \cdot b_j \tag{10.3}$$

C_i *and* C_j *are the WCETs of each task in each corresponding task set.* b_i *and* b_j *are* 1 *if the respective task may stay in the system and* 0 *if the task must be removed from the system in order to achieve schedulability of the remaining tasks. The fraction calculates the load that a task causes in the system, as defined in Definition 4.22 on page 61.*

10.5.2. Scaling Factor Based Approach

As an alternative to the ILP measure, another sensitivity measure is based on a simple real-valued constant scaling factor A, $A \in [0, 1]$. All WCETs and context-switching costs in the system are multiplied by this factor A:

$$\forall i \in \Gamma : c_i^{\text{scaled}} = A \cdot C_i \tag{10.4}$$

$$\forall i, j \in \Gamma : e_{i,j}^{\text{scaled}} = A \cdot e_{i,j} \tag{10.5}$$

Within its range $A \in [0, 1]$, the largest value of A is determined for which the system is still schedulable. I.e., in a system which is already schedulable without any modifications necessary, A equals 1. In a worst-case scenario, A equals 0, meaning that all WCETs would have to be 0 in order for the system to be schedulable.

The maximum value of A can easily be determined by using a binary search and iteratively running the schedulability analysis with the scaled WCETs. In order to limit the number of steps needed to find the real-valued augmentation factor, a user-definable parameter ϵ, $0 < \epsilon < 1$ can be used to define a minimum step width. For the scope of this thesis, $\epsilon = 0.000\,1$ has proven to be a good step width which allows for both differentiating the fitness of two individuals but also leads to practically unnoticable runtimes of the binary search. The algorithm is shown in pseudo code in Listing 10.1.

Listing 10.1: Algorithm used to calculate the augmentation factor of a task set using binary search.

```
1
2  bool testAugmentationFac( taskSet, augFac )
3  {
4    tempTaskSet = taskSet;
5    for( task : tempTaskSet ) {
6      task.wcet = augFac * task.wcet;
7    }
8    // Return true, if tempTaskSet set is schedulable
9    return schedTest( tempTaskSet );
10 }
11 double getAugmentationFactor( taskSet, epsilon )
12 {
13   // Test if initial system is schedulable
14   if( testAugmentationFac( taskSet, 1.0 ) )
15     return 1.0;
16
17   // Perform binary search
18   double upperBound = 1.0;
19   double lowerBound = 0.0;
20
21   while( true ) {
22     // epsilon precision reached. Return result
23     if( lowerBound + epsilon > upperBound )
24       return lowerBound;
25     else {
26       double newBound = (upperBound + lowerBound ) / 2.0;
27       if( testAugmentationFac( taskSet, newBound ) )
28         // newBound leads to schedulability.
29         // This means it is the new lower bound
30         lowerBound = newBound;
31       else
32         // newBound is the new upper bound
33         upperBound = newBound;
34     }
35   }
36 }
```

11. Evaluation Methodology and Experimental Setup

In order to get an impression of the quality of the previously proposed optimizations, it is necessary to automatically create multi-tasking task sets in order to investigate as many different systems as possible. Additionally, multiple assumptions and definitions with regard to the underlying hardware platforms need to be made.

Section 11.1 discusses existing benchmark suites and their limitations and shows how multiple single-tasking benchmarks can be combined to multi-tasking benchmarks. Then, Sections 11.2 and 11.3 proceed by showing how timing parameters like, e.g., activation periods and deadlines can be created randomly for the previously generated task sets. Section 11.4 briefly discusses the relevant properties of the evaluated hardware platforms like, e.g., memory access times and the like. To evaluate the impact of a dedicated real-time scheduler on the optimization framework, Section 11.5 shows how such a scheduler can be created automatically as part of the optimization process. Section 11.6 gives an overview of the tools used for WCET analysis and ILP solvers in this thesis. Finally, Section 11.7 summarizes the concrete parameters used for the evaluations in Chapter 12.

11.1. Existing Benchmark Suites

Multiple benchmark suites exist which target the evaluation of embedded system design methods and tools. All suites have their distinct advantages and limitations. One of the key limitations is that, in order to be analyzable by a WCET analyzer like `aiT`, all cyclic control-flow structures in the benchmarks must be carefully annotated in order to restrict maximum loop bounds and recursion depths. Additionally, while some benchmark suites are simply a collection of multiple stand-alone programs, some feature their own wrappers which are meant to simplify evaluation and configuration. Unfortunately, this makes an automated and comparable evaluation of the benchmarks from different benchmark suites quite hard.

Over the years, many different benchmark suites were included in the `WCC` framework. The benchmarks were carefully annotated with flow facts (cf. Sec-

tion 4.2.1) in order to provide safe bounds on loop bounds and recursions. The rest of this section gives an overview of commonly used benchmark suites, which have all been included into the `WCC` framework in order to be able to perform consistent evaluations.

Most of these benchmark suites target single-tasking systems. A benchmark from these suites consists of one single program with a `main()` routine which is executed once before the benchmark is exited. It does not feature any notion of multi-tasking, activation patterns, deadlines or alike. The best-known single-tasking benchmark suites targeting the evaluation of hard real-time systems are:

DSPstone The DSPstone benchmark suite [Ziv+94] contains benchmarks which specifically target the evaluation of the performance of DSPs. Examples are benchmarks for convolution and matrix multiplication operations, as well as an FIR filter and an FFT. There exist both benchmarks tailored towards fixed point and floating point arithmetic.

MediaBench In contrast, the MediaBench suite [LPM97] targets multimedia applications. The included benchmarks comprise image transformations, GSM enconding and decoding as well as exemplary implementations of h264 and mpeg2 multimedia codecs.

MiBench The MiBench benchmarks [Gut+01] consist of implementations of different algorithms which are commonly found in the domain of embedded systems. Most notably, implementations of the Rijndael encryption algorithm, the SHA hash algorithm, but also a simple qsort algorithm are provided.

MRTC The Mälardalen WCET Benchmarks [Gus+10] are one of the most frequently used benchmark suites for the evaluation of hard real-time systems. It consists of over 30 benchmarks, each resembling a typically found scenario. The benchmarks range from a prime test over sorting algorithms like bsort and quick sort up to ADPCM signal encoders and decoders. The MRTC benchmark suite also features some benchmarks with constructs which are meant to be challenging for WCET analyses and optimizations, e.g., an implementation of Duff's Device [Duf88].

SarBench SarBench [And98] implements the image processing of a Synthetic Array Radar. The benchmark provides both C code and a behavioral description of a processor which can be used to simulate the image processing. However, the C code may also be compiled for other targets without making use of the VHDL descriptions.

StreamIt The StreamIt programming language aims at simplifying the creation of stream oriented programs. The StreamIt community [Str19] provides a benchmark suite for evaluation purposes. Although most of the benchmarks are written in StreamIt, some of them are also provided in C. Most notably,

it provides benchmarks for audio beamforming, an implementation of the sorting network bitonic sort and an FM radio decoder.

TACLeBench The TACLeBench [Fal+16] benchmark suite is a collection of numerous openly available benchmarks. These benchmarks were aggregated from different benchmark suites like, e.g., MRTC or DSPstone. The benchmarks were adapted in order to provide a common coding style and external dependencies were removed. Additionally, cyclic control-flow structures like loops were annotated using flow facts (cf. Section 4.2.1).

UTDSP Similar to DSPstone, the UTDSP benchmark suite [LCS19] provides multiple benchmarks targeting the evaluation of DSPs. It comprises benchmarks for jpeg processing, filters like, e.g., FFT and FIR, compression algorithms and G721 audio decoding and encoding

Few benchmark suites exist which specifically target multi-tasking benchmark suites in the domain of hard real-time embedded systems. Some of the most commonly used multi-tasking benchmark suites are:

DEBIE The DEBIE benchmark [HLS00] consists of 8 tasks. Its origins stem from a European Space Agency project, where the DEBIE software was used as part of a satellite to detect micro-meteroids. It can be compiled for arbitrary platforms for real-time evaluation purposes.

JetBench The JetBench benchmark [QMM10] performs calculations of the thermodynamic behavior of a jet engine. It uses the OpenMP framework in order to allow for multi-threaded execution. In this respect, it does not model any tasks or execution patterns but rather aims at, e.g., distributing the calculations done inside of loops on multiple processing units.

PapaBench The PapaBench benchmark [Nem+06] aims at controlling a small unmanned aerial vehicle. It follows the same task model as DEBIE or `WCC`, using distinct functions as task entries. The PapaBench benchmark consists of two parts, an autopilot and a fly by wire system, each of them consisting of 5 tasks. It was originally created for an ATmega micro-processor. Although it can technically be compiled for any target, it features several platform-dependent features like writing to distinct memory mapped registers.

PolyBench The PolyBench benchmark suite [PY19] consists of multiple single-tasking benchmarks from the domain of data mining and linear algebra. It features a framework using so-called "kernels" which allow for the combination of these benchmarks into multi-tasking task sets.

From the aforementioned multi-tasking benchmarks, only DEBIE and PapaBench could be directly used as a basis for the evaluation of multi-tasking compiler optimizations. JetBench has a different focus on intra-task parallelization, and the PolyBench suite does not offer any timing constraints. For the sake of this

thesis' evaluation, it can basically be reduced to another suite containing multiple single-tasking benchmarks.

Unfortunately, both DEBIE and PapaBench have proven to be of very limited use in practice. Their fixed numbers of tasks makes it impossible to scale the benchmarks. Thus, it is not possible to show the impact of the size of a benchmark on the optimization framework. Additionally, both systems are meant to be schedulable. The provided functionality is so small that the pre-defined deadlines are easily kept without any optimization at all.

As a resort to the limitations of the existing benchmark suites, the evaluation of this thesis will be performed by randomly clustering different single-tasking benchmarks from different benchmarking suites into task sets of pre-defined sizes.

Due to their popular and wide-spread use, the upcoming evaluation chapter uses the benchmarks from the MRTC benchmark suite for task set creation. The `duff` benchmark was removed due to its use of irregular loops. Additionally, the `recursion` benchmark was excluded, because it models indirect recursions.

For reference, the results using task sets created out of all available benchmarks over all benchmark suites listed above are presented in Appendix C. Due to the fact that the TACLeBench benchmark collection mostly comprises benchmarks from the other benchmark suites, it was not considered explicitly.

From the set of selectable benchmarks, a user-defined number of unique benchmarks is selected randomly using a uniform distribution. These benchmarks are then directly passed as a task set to `WCC`. For evaluation purposes, `WCC` has been extended in order to automatically suffix any global symbols of each task. This prevents duplicate definitions of function names or global symbols in a task set. Each individual benchmark's `main()` function is also prefixed and marked as an entrypoint using a `_Pragma` annotation.

At this point, there is scheduling code existing which actually manages the execution of each task if the task set was run on real hardware. For a pure analysis of the performance of an optimization framework, this may be neglected. This corresponds to the assumption that, in a fully synthetic task set, without loss of generality the highest priority task may simply be assumed to be the task set's scheduler.

As an alternative, `WCC` was extended within the course of this thesis to automatically generate a fixed-priority scheduler specifically tailored towards a supplied task set. This is done by adding a specialized scheduler function as part of the regular compilation. Despite being useful for evaluation purposes, it may also be used in regular use cases if a minimal scheduler is sufficient for the system designer. Due to its complexity, the automatically generated scheduler is described in more detail in the upcoming Section 11.5.

11.2. Creating Scheduling Parameters

Once a synthetic task set has been created, timing parameters like deadlines and activation periods must also be created. In case that the system to be evaluated is already schedulable without applying a schedulability-oriented optimization, applying the optimization is meaningless. Therefore, the evaluation framework has to generate systems which are *not* schedulable. To examine the potential of the optimization framework, the generated systems must range from "almost schedulable" to systems with CPU loads significantly beyond 100 %.

The general idea is to first compile a given task set using standard ACET compiler optimizations. Then, the WCETs c_i of each task τ_i in the task sets are analyzed using a standard WCET analyzer like `aiT` or `WCC`'s internal WCET analyzer.

For the upcoming evaluation, a periodical task model with jitter will be used. Due to the event-based modeling of the schedulability in Section 8.3, the ILP optimization framework does not differ between differently activated task models. The genetic optimization framework proposed in Chapter 10 neither does distinguish between different activation patterns.

Therefore, a relatively simple periodic system with jitter provides a practical way for presenting and discussing evaluation results, as each task's execution pattern depends on only two parameters. Using more complex or even randomly assembled activation patterns for each task would render any evaluation results practically incomprehensible and irreproducible.

The density function for a periodical task with jitter τ_i is modeled by a fixed period T_i and a maximum jitter J_i. The corresponding density function $\eta_i(\Delta t)$ and the interval function $\delta_i(n)$ can be written as follows [Ric05, p. 39-43]:

$$\eta_i(\Delta t) = \left\lceil \frac{\Delta t + J_i}{T_i} \right\rceil \tag{11.1}$$

$$\delta_i(n) = \max((n-1)T_i - J_i, 0) \tag{11.2}$$

In combination with the WCETs of the unoptimized tasks, each task's timing properties like its deadline, period and jitter can be calculated in order to achieve a user-defined overall system load u. If this load is beyond 100 %, the system is clearly unschedulable. Otherwise, an initial schedulability test may be performed in order to determine whether the generated task set is schedulable without applying schedulability-oriented optimization.

To achieve this, partial loads u_i for each task τ_i are generated using the UUniFast algorithm by Bini and Butazzo [BB05] for a user-defined target system load of u. UUniFast creates uniformly distributed partial loads for each task such that the sum over all partial loads u_i equal u.

Given the randomly generated load u_i and a previously analyzed WCET c_i of the unoptimized task τ_i, the period T_i can easily be calculated:

$$T_i = \frac{c_i}{u_i} \tag{11.3}$$

For the Jitter J_i, a maximum derivation x from the period is defined by the user. Then, the jitter is also determined randomly in the range $[0, T_i \cdot x]$ with a uniform distribution.

The same way, deadlines can be generated with a uniform random distribution being either tighter or larger than the task's period. For another given maximum derivation y, the deadline D_i is chosen from the interval $[T_i - T_i \cdot y, T_i + T_i \cdot y]$. This way, systems with $D_i > T_i$ can be evaluated if needed.

11.3. Adjusting Periods

The hyper-period is the least common multiple over all fundamental periods in a task set with tasks which are triggered in a periodically recurring pattern (cf. Definition 4.23 in Section 4.5.2).

The WCETs of tasks in a task set may easily range between only a couple of hundred CPU cycles for a small task to multiple million CPU cycles for larger tasks. As a result, CPU clock cycles are the only common time base which can be chosen to randomly determine the period for a task as described in the previous section. The CPU frequency for the systems analyzed in this thesis are in the order of some MHz. A more coarse-grained time base (like, e.g., micro-second or even milli-second) might therefore be feasible for tasks with a WCET in the range of some million CPU cycles, but leads to a rounded WCET and period of 0 for the small tasks.

Example 11.1 (Hyper-Period of a small task set)
The benchmarks `crc`, `fibcall`, `lms` *and* `sqrt` *were selected randomly from the MRTC benchmark suite and assembled into one task set. The task set was compiled for the Infineon TriCore TC1796 architecture using* `WCC` *and the standard optimizations provided by* `-O2`*. Each task's WCET was then analyzed using AbsInt* `aiT`*. Based on these WCETs, the partial loads u_i were randomly generated using the UUnifast approach as discussed in the previous section.*

Following Definition 4.16, the load of a periodical system is calculated as

$$u = \sum_{\tau_i \in \Gamma} u_i = \sum_{\tau_i \in \Gamma} \frac{c_i}{T_i} \tag{11.4}$$

Table 11.1: Generated Periods of an exemplary task set consisting of 4 tasks. The WCETs have been retrieved using `aiT` for the TriCore TC1796 target architecture.

Task	c	Partial Load u_i	Resulting Period T_i
τ_0 (`crc`)	196 353	0.111 058	1 768 022
τ_1 (`fibcall`)	698	0.271 647	2 569
τ_2 (`lms`)	3 618 471	0.410 23	8 820 590
τ_3 (`sqrt`)	39 062	0.007 065 34	5 528 679

If the WCET c_i and partial load u_i are given, the resulting period T_i can subsequently be calculated by:

$$T_i = \frac{c_i}{u_i} \tag{11.5}$$

For the exemplary system, the resulting periods are depicted in Table 11.1. They were rounded towards the next smaller integer value. It can be seen that the resulting period T_1 of τ_1 is quite small compared to T_2 of τ_2. For this task set, the resulting hyper-period is approximately $2.215\,86 \times 10^{22}$.

As Example 11.1 shows, even for a relatively small system consisting of only 4 tasks, the hyper-period can already become huge. Such large hyper-periods lead to two problems: First – depending on the also randomly chosen deadlines for each task – a schedulability test might have to be performed up to the hyper-period. Second, the schedulability constraints encoded into the ILP-based optimization framework presented in Chapter 8 also have to be performed up to this bound. For the smallest task, τ_1, this means that approximately 8.6×10^{18} instances of τ_1 must be checked for schedulability.

To both decrease solving time and reduce the risk of numerical issues within the used ILP solver tools due to the large integer numbers, this work uses the approach proposed by Xu [Xu10] in order to slightly reduce the periods of each task such that the hyper-period of the task set is reduced significantly.

This does not render the proposed approaches unsafe, but solely adds some pessimism. I.e., a task set could be repairable using the original periods, but might fail using the reduced ones in some cases.

The basic idea of Xu is to generate a list of valid periods which were all derived from a polynomial with given prime base factors. Then, the next tighter period is chosen from this list for each task. Depending on the polynomial's parameters,

a trade-off between the maximum deviation from the original period and the maximum hyper-period can be made.

In this thesis, the set of valid periods $\mathcal{T}$ is calculated as follows by using the first 7 prime numbers:

$$\mathcal{T} = \left\{ T^{red} | T^{red} = 2^{e_2} \cdot 3^{e_3} \cdot 5^{e_5} \cdot 7^{e_7} \right\} \tag{11.6}$$
$$e_2 = 0, \ldots, E_2 \tag{11.7}$$
$$e_3 = 0, \ldots, E_3 \tag{11.8}$$
$$e_5 = 0, \ldots, E_5 \tag{11.9}$$
$$e_7 = 0, \ldots, E_7 \tag{11.10}$$

The e_x variables denote the exponents for the respective base prime number. The set of valid reduced periods $\mathcal{T}$ is generated by iterating each and every e_x from 0 to a user-defined maximum value E_x.

Obviously, the maximum reduced period which can be expressed by this approach is $2^{E_2} \cdot 3^{E_3} \cdot 5^{E_5} \cdot 7^{E_7}$. Therefore, the maximum exponents E_x are crucial in order to reduce large periods without a large deviation from the original period. As a trade-off between small deviations from the original periods and feasible hyper-periods even for very large original periods, different sets of maximum exponents are chosen in this thesis, depending on the largest original period within a given task set.

Using exhaustive search, different exponents have been determined which are used for task sets with certain maximum periods. Each of these guarantee a relative error of the reduced periods of maximum 20 %, although in practice the error is significantly lower in most cases.

For a maximum period which is lower than 1×10^7, the following maximum exponents are used:

$$E_2 = 6 \tag{11.11}$$
$$E_3 = 6 \tag{11.12}$$
$$E_5 = 3 \tag{11.13}$$
$$E_7 = 1 \tag{11.14}$$

For this set of exponents, the largest deviation is 18.35 % for the maximum period of 1×10^7 and the largest possible hyper-period of the task set using the reduced periods is 40 824 000.

If the maximum period in the task set exceeds 1×10^7 but is below 1×10^{11}, the following maximum exponents are used:

$$E_2 = 7 \tag{11.15}$$
$$E_3 = 6 \tag{11.16}$$
$$E_5 = 3 \tag{11.17}$$
$$E_7 = 3 \tag{11.18}$$

For these exponents, the largest relative error is 19.99 % and the largest hyper-period using reduced periods is 4 000 752 000.

Finally, for task sets in which the largest occuring period exceeds 1×10^{11} but is below 1×10^{13}, the following exponents are applied:

$$E_2 = 9 \tag{11.19}$$
$$E_3 = 6 \tag{11.20}$$
$$E_5 = 5 \tag{11.21}$$
$$E_7 = 3 \tag{11.22}$$

For these values, the largest relative error is 20.00 % and the largest possible hyper-period is 400 075 200 000. The maximum period in a task set is not formally limited, as both the maximum values of the maximum exponents as well as the largest prime number used in Eq. (11.6) can be chosen aribitrarily by the system designer. For this thesis, the task set is treated as not repairable if a larger period is observed in the task set.

Example 11.2 (Reduced Periods)
For the task set given in Example 11.1, all original periods are below 1×10^7*. Thus, the first set of exponents is used.*

The reduced period T_i^{red} *for each task* τ_i *is obtained by finding the largest period in the set of reduced periods* $\mathcal{T}$ *which is still smaller than or equal to the original period* T_i*. The reduced periods* T_i^{red} *are depicted in Table 11.2.*

It can be seen that τ_3 *is suffering from the highest deviation from the original period with* 7.699 %*. At the same time, the reduced period of* τ_1 *only differs by* 1.91 % *from the original period.*

The least common multiple and thus the hyper-period of the task set using the reduced periods results in 40.824×10^6*. I.e., the original hyper-period is reduced by a factor of about* 5×10^{14}*.*

Table 11.2: Reduced Periods of an exemplary task set consisting of 4 tasks. For comparison reasons, the table also contains the original periods and the (rounded) deviation in percent.

Task	Original Period T_i	Reduced Period T_i^{red}	Deviation $\left(1 - \frac{T_i^{red}}{T_i}\right) \cdot 100\,\%$
τ_0 (`crc`)	1 768 022	1 701 000	3.791 %
τ_1 (`fibcall`)	2 569	2 520	1.907 %
τ_2 (`lms`)	8 820 590	8 164 800	7.435 %
τ_3 (`sqrt`)	5 528 679	5 103 000	7.699 %

11.4. Evaluated Hardware Architectures

`WCC` offers support for the ARM7TDMI and the Infineon TriCore architectures. This section gives a brief overview of the key parameters of both architectures, as they are used for the evaluation in the upcoming Chapter 12.

11.4.1. ARM7TDMI

As basis for the evaluated ARM7TDMI microcontroller architecture, the NXP LPC2880 running at a maximum of 60 MHz is used throughout the upcoming evaluations [LPC08]. It follows the von Neumann architecture which was described in Section 3.1. The microcontroller features multiple on-chip memories. A Flash memory is provided as regular instruction memory. An access to this memory needs 6 CPU cycles. Additionally, a fast SRAM memory is available. An access to the SRAM can be achieved within 1 CPU cycle.

The SRAM is divided into two regions: One for data, and one for instructions. The part reserved for data is considered to be large enough to hold all data objects needed by the task set. The other part of the SRAM is used as instruction SPM. Analogously to the TriCore setup, the SPM memory will be scaled in the evaluation chapter to provide better impressions on the capabilities of the optimization framework.

The ARM7TDMI features a very simple 3 stage pipeline consisting of instruction fetch, instruction decode, and an execute stage. It does not feature any instruction prefetching or block fetches.

The ARM7TDMI is also lacking dedicated floating-point and branch prediction units. Instead, all floating-point operations must be performed in software. When executing a branch instruction, its implicit physically succeeding instructions are

always fetched. If the branch is then taken, a pipeline flush is issued in order to clear the incorrectly fetched instructions.

For the ARM7TDMI architecture, it is assumed that all caches have been disabled. The impact of instruction caches will instead be evaluated on the much more complex TC1796 architecture which is introduced in the next section. Evaluating the impact of caches on the distinctly simpler ARM7TDMI architecture is not expected to provide any additional scientific insights.

11.4.2. Infineon TriCore

As an example of one of the more complex microcontrollers which are used in timing critical setups, the Infineon TriCore TC1796 running at 150 MHz is evaluated within this thesis. It follows the Harvard architectural model (cf. Section 3.1) using different memories for data and instructions.

It features an on-chip Flash memory with an access time of 6 CPU cycles which is used as regular memory for instructions. For the evaluations in this thesis, all data is stored in an on-chip SRAM acting as data SPM with 1 cycle access time. A separate SRAM which acts as an instruction SPM can also be accessed within 1 CPU cycle.

To be able to see the impact of the SPM optimization according to the available SPM, the size of the SPM is increased and decreased for evaluation purposes. This is explained in more detail where needed in the evaluation (cf. Chapter 12).

Finally, the TC1796 features an instruction cache which may be activated or deactivated by the user. This is used in order to evaluate the impact of caches to the schedulability-aware optimization framework (cf. Chapter 9). Analogously to the SPM size, the cache size will be scaled in the evaluation to evaluate the optimization framework's performance for different cache sizes.

On an architectural level, TriCore features 3 different pipelines: One regular 4 stage pipeline, which offers the well-known instruction fetch, decode, execute and writeback stages. The execute stage features an Arithmetic Logical Unit (ALU) with a dedicated floating-point unit for single precision floating point arithmetic. Additionally, a co-processor offers basic DSP functionalities like, e.g., support for multiply-accumulate assembly instructions.

Apart from this "main" pipeline, it offers a so-called "Load Store" pipeline which is dedicated to handling instructions which load data from or write data to memory, perform address arithmetic or context switching. It allows for the execution of `load` instructions without stalling the execution of arithmetic instructions, as long as these instructions do not interfere with the registers used by the `load` instructions. Finally, a loop pipeline enables the usage of special

`loop` assembly instructions which allows for the realization of loops with zero runtime overhead (i.e., the `loop` instruction does only need computational time in the CPU's pipeline for the first and last iteration of the loop).

Additionally, TC1796 also supports block fetches due to a 64 bit wide bus width of its program memory unit. This means that, depending on the alignment of a basic block, fetches of multiple consecutive instructions may be significantly faster than fetching each instruction separately.

11.5. Accounting for the Scheduler

Section 11.1 discussed the combination of multiple single-tasking benchmarks into a multi-tasking task set. In such a task set, each individual task is not yet triggered, as no scheduler exists. For a purely synthetical schedulability analysis, this usually suffices. It can either be assumed that each task is triggered by hardware interrupts or that a scheduler runs on a dedicated co-processing unit without interferences on the regular tasks' runtime behavior. Such a co-processor exists, e.g., on the Infineon TriCore platform. Alternatively, scheduling costs can simply be considered to be negligible. Finally, the highest priority task in the randomly generated task set can be assumed to be the scheduler. For a general overview of the applicability and quality of the optimization framework, these simplifications are feasible. Yet, in the course of the evaluation of a low-level compiler based evaluation framework, it is interesting to see the actual impact of a scheduler in a real-world scenario.

In order to investigate these issues, this section proposes the auto-generation of a small fixed-priority scheduler which can then be used for an automated evaluation. This task is realized as a dedicated function which performs the dispatching of the actual tasks. It is generated automatically by the `WCC` compiler framework. The general outline of the scheduler was done as part of this thesis. The implementation of the scheduler code itself was conducted as part of a student's bachelor thesis [Fis18].

11.5.1. Scheduler Structure

In order to allow for a tight WCET estimation, the automatically generated scheduler consists of one single function. Each task is depicted as a C **`struct`** consisting of a pointer to the task's entry function as well as its period and internal management data (like, e.g., an integer value holding the address of each task's stack pointer offset).

A global 32 bit integer variable acts as a bit field where each bit of the integer signifies if the task with the respective index is ready for execution or not. E.g.,

if the least significant bit in this so-called ready-list is 1, then τ_0 is ready for execution. The restriction to a maximum of 32 tasks is sufficient for the evaluation within this thesis.

The scheduler function is considered to be executed by an interrupt service routine. This may happen either periodically or each time an incoming event occurs.

On execution, the scheduler checks the ready-list and dispatches the task with the highest priority which is ready for execution. It is therefore also assumed that the ready-list can be updated by an interrupt service routine or any comparable means. From a technical point of view, this can be seen as a regular call of the respective task's entry function.

In order to allow for clean preemptions, the scheduler saves any registers prior to executing a task and restores the previously saved registers once the task finishes its execution (i.e., the call to the task's entry function returns). The scheduler also ensures that the stack pointer is set such that tasks' stack memories do not overlap.

Despite the fact that the general structure of the scheduler is machine-independent, the implementation of the dispatching code is highly platform dependent due to the necessary stack management and register saving. Additionally, on some architectures (e.g., on the Infineon TriCore TC1796), optimized assembly instructions exist which allows for the evaluation of the ready-list in constant time.

In the course of the bachelor thesis [Fis18], the scheduler was implemented for the TC1796 architecture. Functional correctness was shown as part of the bachelor thesis by applying unit tests and simulating the execution of the scheduler using the platform simulator Synopsys CoMET [Com19]. The concrete implemention of the scheduler is merely technical and is discussed in detail in [Fis18].

11.5.2. Activation Pattern

The event density function $\eta_i(\Delta t)$ of a task τ_i bounds the maximum number of events which may happen within Δt. At the hardware level, an event will most likely be signaled to the microcontroller by an interrupt which is triggered. If only a few tasks exist, the microcontroller's hardware interrupt capabilities may already be used in order to manage priorities for each task in the system. In this case, no real scheduler will be needed.

In the general case, a microcontroller's hardware capabilities will not suffice to manage all tasks using individual interrupt routines. In this case, the microcontroller's interrupt routine will trigger the dedicated scheduling task which, in turn, determines which task should be executed. There are two fundamentally different ways how this may be done: *time-triggered* or *event-triggered*.

A *time-triggered* scheduler is executed with a given execution period T_s, similar to a regular periodic task. In such a setup, no task can be executed without the scheduling task being executed first. In a worst-case scenario, a task has to wait up to one full period of the scheduling task T_s until it can be executed – independent from its priority. This so-called *blocking time* must be added as a constant summand to the WCRT of every task. Subsequently, the WCRT of each task in the system is highly dependent on the activation period of the scheduler. If the activation period is chosen too low, the blocking time is so high that the task might miss its deadline. If the activation period is chosen too small, the scheduler preempts the actual tasks so frequently that this overhead may also lead to deadline violations. Additionally, for arbitrarily triggered tasks, a time-triggered scheduler is inflexible, as it cannot appropriately handle activation bursts. Either the scheduling period must be chosen so small that activation bursts are handled quickly, then the runtime overhead by the scheduler will be high, or the scheduler period is chosen larger, then several instances of one task may pile up. For the course of this thesis, finding a good period for a time-triggered scheduler is even more difficult: As the underlying assumption of this thesis is that the unoptimized system is *not* schedulable, there is no "better" or "worse" scheduler period, as *any* period will lead to deadline violations.

In contrary to the time-triggered system, any *event* which triggers the execution of a task (like, e.g., incoming sensor data) instantaneously triggers the execution of the scheduler task in an *event-based* system. The scheduling task can thus be seen as a regular task in the system with an event density function η_s. This event density function can simply be expressed by the sum over all event density functions of all tasks within the system:

$$\eta_s(\Delta t) = \sum_{\tau_i} \eta_i(\Delta t) \tag{11.23}$$

In such a design, no blocking time by the scheduler has to be considered and the system can be considered to be fully preemptive. The scheduler can be considered as a regular task with highest priority in the schedulability analysis.

As the optimization framework in this thesis focuses on event-based systems, the evaluated scheduling task is also supposed to be triggered in an event-based fashion.

11.6. Used Tools

Apart from the `WCC` compiler framework, external tools were used for both WCET analysis and in order to solve the ILPs needed for the schedulability-aware

optimizations. This section briefly introduces the concrete tools which were used in the upcoming evaluations.

11.6.1. WCET Analyzers

AbsInt `aiT` [FH04] is one of the leading tools used to analyze the WCET of a given task. `aiT` contains support for a number of target architectures, including the ARM7TDMI and Infineon TriCore architectures which are used for evaluation in the course of this thesis. For the evaluation of the Infineon TriCore TC1796 architecture, `aiT` was used in version 18.10. At the time at which this thesis' evaluation was performed, this was the latest version available.

For the ARM7TDMI architecture, `WCC` also features its own precise internal WCET analyzer, as proposed by Kelter [Kel15]. This framework is able to analyze both single-core and multi-core architectures based on ARM7TDMI microcontrollers. While `aiT` 18.10 is able to analyze the ARM7TDMI, it is not able to account for multi-core effects. Despite the fact that this thesis focuses on single-core architectures, both the ILP-based and the genetic approach are also applicable to multi-core systems with shared memories. This has been realized as an outlook to possible future use cases of the proposed framework [LOF20].

In order to provide comparability with future works on multi-core systems, `WCC`'s internal WCET analyzer will be used for the ARM7TDMI target.

11.6.2. ILP Solvers

There exist multiple tools which may be used in order to solve ILPs. Some of the most commonly found tools comprise:

`CPLEX` `CPLEX` [Cpl19] is a commercial ILP solver by IBM. It is known for its high performance and is commonly used by both industry and research.

`Gurobi` `Gurobi` [Gur19] is a commercial ILP solver by Gurobi Optimization, Inc. Compared to `CPLEX`, it is relatively new but comes with the promise of significant performance improvements over any other ILP solver.

`lpSolve` As an alternative to commercial solvers, `lpSolve` [Lps19] is a non-commercial open source tool which is publicly available under the GNU lesser general public license. As a result of its public availability, it is commonly used as an ILP solver in academia.

The ILP framework proposed in this thesis can lead to large inequation systems with several thousand variables and constraints. Many of these variables are *binary*, due to logical formulations (cf. Chapter 5).

In the course of this thesis, the ILP solvers were compared with regard to their capability of solving such large combinatorial problems with many binary decision variables. The results of these evaluations can be found in [LOF18]. All evaluation results show that `lpSolve` is outperformed significantly by both `CPLEX` and `Gurobi`. In fact, solving realistic problems with `lpSolve` is basically infeasible as its runtime easily reaches multiple hours or even days. For `Gurobi` and `CPLEX`, `Gurobi` outperformed `CPLEX` in all cases. For logical operations like AND or XOR, `CPLEX` is almost as fast as `Gurobi`. However, for ILPs containing large amounts of conditionally enabled constraints, `CPLEX` easily needs several hours to find an optimal solution while `Gurobi` finishes within a couple of seconds. In the evaluation presented in [LOF18], an ILP containing 10 000 conditional operations could not be solved by `CPLEX` realiably within a 2 h time limit, while `Gurobi` finished within 80 s on average. Furthermore, it turned out that native support of conditionally enabled constraints in `Gurobi` decreases the solving performance significantly.

As a result, the `Gurobi` solver in its current version 8.1 is used for all evaluations. Additionally, `Gurobi`'s native implementation for logical operations like AND, OR, min and max are used when feasible. However, due to a much faster solving time, conditionally enabled constraints (cf. Section 5.2.2.2) are modeled explicitly instead of using `Gurobi`'s native support. In order to speed up the solving times of the ILP, `Gurobi` was configured to spawn up to 4 computational threads in parallel.

11.6.3. Invocation of WCC

All evaluations are performed using the WCET-Aware C Compiler (`WCC`) [FL10]. An introduction into `WCC` was given in Chapter 6. For the realization and evaluation of the genetic approach proposed in Chapter 10, `WCC` is tightly interfacing with `PISALib` [Ble+03].

Each task set is initially compiled using `WCC` and the `-O2` optimization level. This enables several standard ACET compiler optimizations like, e.g., redundant and dead code elimination, value propagation and constant folding, loop transformations, removal of unused return values and function arguments. This is approximately comparable to running `gcc` with the `-O2` optimization flag. The maximum standard optimization level `-O3` was not used, because – similar to `gcc` – this would enable function inlining and loop unrolling which may heavily increase a task set's code size, thus possibly even reducing the optimization potential of any subsequent memory allocation based WCET-aware optimization technique. For the Infineon TriCore TC1796 architecture, additionally the use of floating-point assembly instructions was activated. For the ARM7TDMI architecture, this does not apply as ARM7TDMI does not feature a floating-point unit.

Compared to a fully unoptimized program (i.e., issuing the `-O0` optimization level), this allows for a more realistic evaluation, as common ACET optimizations like, e.g., constant propagation also improve on the worst-case timing behavior. The task set is then fully assembled and linked and each task's WCET is initially analyzed. Based on these WCETs, timing parameters are then determined for a user-specified target load as discussed in Section 11.2.

Then, the optimization which is to be evaluated is performed from within `WCC`. If a suitable solution is found within a time limit of two hours, the optimization returns this solution. The optimized program is then compiled, assembled and linked, and once again each task's WCET is analyzed. Finally, a schedulability test is performed using these timing parameters in order to verify that the optimization did return a correct solution.

In case that no suitable solution to the optimization problem could be found within a user-selectable time limit, the genetic approach selects the best individual available. The ILP-based algorithms returns the original task set without any modifications. In both cases, the resulting task set will not yield a schedulable system.

11.7. Evaluation Setup

This section gives an overview of the concrete parameters used for the evaluation setup in the upcoming Chapter 12. The evaluation is performed for task sets consisting of 2, 4, 6 and 8 tasks, respectively. For each task set size, 20 different task sets were generated randomly (cf. Section 11.1). These task sets are then used for all optimizations for both ARM7TDMI and Infineon TC1796.

For each task set, different unoptimized loads are randomly generated as described in Section 11.2. All evaluations are evaluated for initial loads ranging from 0.8 to 2.2. This allows for a good impression of the performance of most of the evaluated optimizations. However, for the ILP-aware SPM optimization for ARM7TDMI in Section 12.1.1, the repair rates were so good that the range was extended to an initial load of 3.0 for this evaluation.

Each task is modeled as a periodical task with jitter. The jitter may range between 0 % and 1 % of the task's original period. The tasks' deadlines were chosen between 80 % and 120 % of the original period. Periods are then adjusted as described in Section 11.3 in order to obtain computationally feasible hyper-periods.

The size of the SPM memory plays a crucial role when evaluating the optimization potential of an SPM allocation. If the SPM size is very small, only little to none basic blocks can be allocated to the fast memory. Subsequently, the evaluation will not show any meaningful results, because, no matter how good the optimization

algorithm is, the hardware configuration does simply not allow for any meaningful improvements. If the SPM size is chosen too big, then the optimization problem becomes trivial as all – or at least all timing-critical – basic blocks of the task set can be assigned to the SPM. To counter this issue, the SPM size was chosen relatively to each task set size. E.g., if a task set's code size is 100 kB, an SPM with 20 % relative size would be 20 kB big. Thus, at most 20 % of all instructions of the task set can be allocated to the SPM. In reality, this will be somewhat less, because the allocation is done at basic block and not instruction level. Also, the additionally needed jump correction (cf. Section 7.3.1) may add additional overhead. For this thesis, a relative SPM size of 40 % proved to provide meaningful results for the SPM allocation for both the ARM7TDMI and the TC1796 architectures. Appendix A shows the results for 20 %, 40 %, 60 %, 80 % and 100 % relative SPM size.

When using an instruction cache, the cache size plays an equal role than the SPM size when it comes to evaluating an optimization's performance: If the cache size is too big, there won't be any cache evictions. If the cache size is too small, there won't be any meaningful number of cache hits. In order to maintain comparability, the cache size was chosen identical to the SPM size discussed in the previous paragraph.

To avoid barely schedulable systems, the ILP objective to minimize the sum over all WCETs was also applied to the ILP *with* schedulability constraints.

As a basic preemption penalty, two additional memory accesses are assumed for both Infineon TriCore and ARM7TDMI in order to refill the processor's pipeline when resuming a preempted task (cf. Section 8.4). The concrete position of a preemption in the code is not known at compilation time. Therefore, the preemption variable $e_{i,j}$ has been modeled within the ILP such that the access timings to the slow main memory are assumed if at least one basic block of the preempted task resides in this slow main memory. If the whole task is allocated to SPM, then the SPM timings are used.

Finally, in order to limit evaluation times, a timeout of two hours was specified for each optimization, *excluding* the compiler-based overhead or WCET analyses.

12. Evaluation

In this chapter, both the ILP-based optimization framework and the framework based on a genetic algorithm are evaluated and the results are discussed. For all evaluations, the methods discussed in Chapter 11 are used.

For each evaluated setup, both the repair rates and the needed compilation time are evaluated. The compilation time always comprises the whole compilation process, including standard optimizations, all WCET and schedulability analyses and the optimization itself. If a process is terminated by timeout, the depicted time may exceed the denoted timeout, as the watchdog which monitors all evaluation processes for their runtime may not always be able to terminate the running process at once due to techncial reasons. Evaluation times are depicted as box plots. In these plots, the central red mark of each box denotes the median over all results. The edges of the box depict the 25^{th} and 75^{th} percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75^{th} and 25^{th} percentile. Outliers are indicated by additional red markers.

The repair rates are given as bar plots. The x axis depicts the load of the system prior to applying the optimization under test. The y axis depicts the percentage of systems which is schedulable after applying the respective optimization.

Section 12.1 shows the results for systems with no caches for both ARM7TDMI and Infineon TriCore architectures. Section 12.2 continues by evaluating systems with instruction caches. In both sections, the scheduling task is not modeled explicitly. Section 12.3 shows the impact of a scheduler task on the repair rates for the Infineon TriCore architecture. The results are summarized in Section 12.4.

12.1. Systems Without Instruction Cache

In order to give an impression of the general performance of the schedulability-aware optimization framework, this section evaluates the repair rates and necessary compilation times for systems without caches and without explicit modeling of a scheduler.

12.1.1. ILP Optimization on ARM7TDMI

This section shows the results of the ILP-based optimization approach for the ARM7TDMI architecture. The concrete setup was previously described in Section 11.7.

Fig. 12.1 shows the results with regard to the repairable task sets. The X-axis of each graph denotes the original unoptimized system load. The Y-axis gives the percentage of task sets which are schedulable after applying the respective compiler optimizations. The bars depict the arithmetic mean over the repair rates for DMS and EDF for each evaluated optimization and system load.

The first two bars show the results for DMS scheduling. The first bar (green color) shows the percentage of systems which are schedulable after applying the schedulability-aware ILP optimization framework as proposed in Section 8.3.2. In order to illustrate that the schedulability-aware ILP constraints are actually needed in order to achieve good repair rates, the second, blue, bar shows the results for the schedulability *un*aware SPM optimization from Chapter 7. In the graphs' legend, this is marked by "NoSched". This optimization simply tries to minimize the sum over all WCETs of all tasks without regarding context switching costs or the impact of the scheduling algorithm. The second two bars in orange and yellow show the analogous results for EDF scheduling in the same order.

The results show that the schedulability-aware optimization framework is able to repair a significant amount of task sets up to an original unoptimized system load of 300 %. For loads below 1.2, virtually all task sets were schedulable. For improved readability, these loads are therefore skipped in Fig. 12.1. For the sake of completeness, the numbers can be found in the Appendix in Table A.2. For the depicted loads, it can be seen that the number of repairable task sets stays at a very high level up to a load of 1.6 and then starts to gradually decrease. These generally high repair rates stem from its relatively simple hardware structure. ARM7TDMI does not feature any prefetching mechanisms. With the timings used in this evaluation, each instruction fetch from the slow Flash memory will stall the CPU for 6 cycles, while an access to the SPM can be performed within one cycle. As a result, moving all instructions to the SPM would lead to a speedup of up to a factor of 6. This allows for a good impression of the capabilities of the schedulability-aware optimization framework, as the theoretical improvements due to a more efficient hardware usage (i.e., an optimal SPM allocation) are very high.

The repair rates at low loads do not change significantly with the size of the task set. The reasons for this are both the high optimization potential and the relative SPM size used for the evaluation. However, due to the larger number of evictions, repair rates inevitably decrease for increasing task set sizes at higher loads.

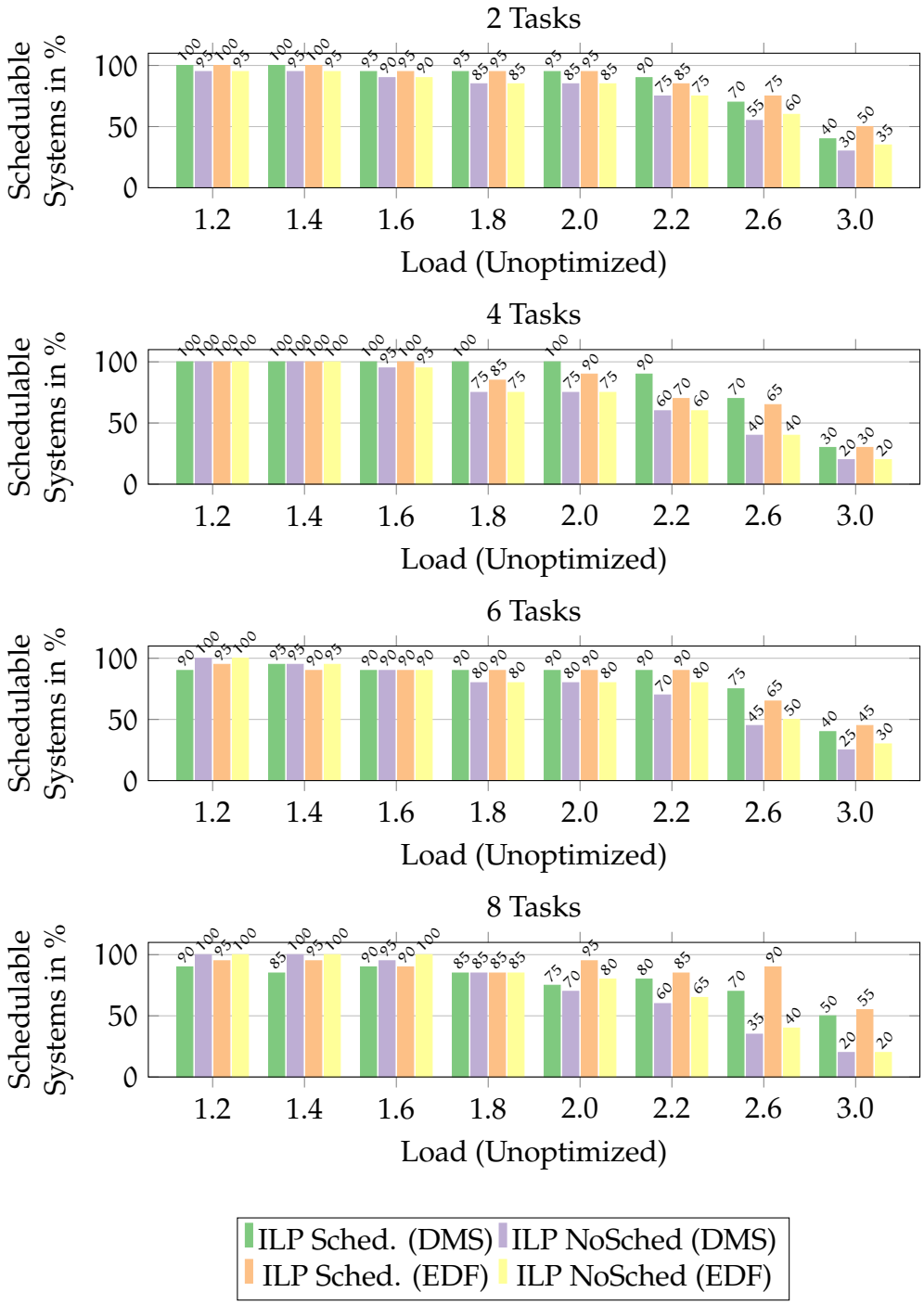
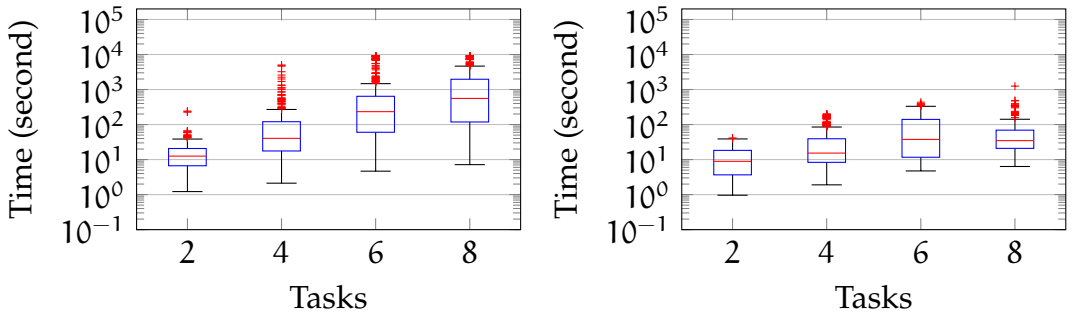


Figure 12.1: ILP-based repair rates for the SPM optimization on the ARM7TDMI architecture. The SPM size is 40 % of the size of each task set. The graphs compare the repair rates for ILPs with and without schedulability constraints for both EDF and DMS scheduling.

For relatively small loads, DMS (green) and EDF (orange) perform comparably good. For larger task sets and loads, however, differences can be seen. For the systems with 4 tasks and an original system load of 200 %, 100 % of the evaluated task sets could be repaired when using DMS, compared to only 90 % for EDF. At 220 % original load, still 90 % of the task sets could be repaired for DMS and 60 % when using EDF. Despite the fact that EDF is supposed to be an optimal scheduling algorithm, it can lead to worse results in practice. This stems from the fact that EDF may lead to a higher number of possible preemptions. While the approach in this section does not explicitly model a scheduling task, it *does* account for preemption penalties (e.g., the flushing of the CPU's pipeline at preemption of a task and having to refill it once the preempted task is resumed). As a result, EDF may lead to worse results than fixed-priority scheduling algorithms like DMS. However, this is no general rule but depends on the actual tasks and their scheduling parameters. This can be seen, e.g., for the system consisting of 8 tasks and an original load of 200 % and 260 %. Here, EDF significantly outperforms DMS.

In comparison to the schedulability-*un*aware optimization, the results in Fig. 12.1 show that for relatively low system loads in large task sets, the schedulability-aware framework may perform worse than an schedulability-unaware optimization. This stems from the fact that both modeling the timing behavior of a benchmark within the ILP-based optimization framework is not as precise as a final WCET analysis. The reason for this lies in the simplifications made by modeling each task's WCET in the underlying ILP framework (cf. Section 7.3.3). Additionally, the calculation of the gain achieved by moving a basic block into SPM can also be conservative within the ILP model, due to the necessary accounting for possible jump correction code. Therefore, the schedulability-aware optimization framework may not return *any* solution, as it does not find a solution within the ILP formulation which does not violate at least some constraints. For the ILP without schedulability constraints, however, over-approximations of the WCETs are not that critical. The optimization simply tries to optimize "into the right direction" by reducing the sum over all WCETs. For relatively small loads, chances are good that mostly *any* optimization will in fact reduce the task sets' WCETs to an amount such that the system is schedulable. Therefore, while the schedulability-aware optimization simply bails out, the schedulability-unaware approach may lead to a feasible solution. For larger loads, however, it can clearly be seen that accounting for the tasks' timing constraints is crucial in order to achieve good repair rates.

The advantages of the schedulability-aware constraints can thus be especially seen for the large task set consisting of 8 tasks and the highest evaluated system load of 300 %. Here, 50 % and 55 % of the systems could be repaired with the schedulability-aware ILP optimization, while the *not* schedulability-aware reference optimization is only able to repair 20 % of the evaluated task sets.



(a) Schedulability-aware (b) Schedulability-unaware

Figure 12.2: Compilation times for ILP-based SPM allocation for the ARM7TDMI architecture (40 % SPM size), including the whole compilation process and all WCET analyses.

Finally, it can be seen that there are some somewhat counter-intuitive evaluation results. E.g., for the task set consisting of 4 tasks and a load of 180 %, 85 % of the task sets could be repaired when using EDF. On the other hand, for a load of 200 %, the number of repairable task sets actually raises to 90 %. This stems from the fact that the timing parameters were generated randomly. It is therefore possible that, for the load of 180 %, deadlines and periods are generated less favorable for the optimization framework as for the higher load. Not all tasks can be optimized to the same extent. The WCET of a task featuring one small loop which is executed very often may be drastically reduced even with a small SPM. On the other hand, benchmarks with many branches and few loops are hard to optimize since moving the code of one branch to the SPM might easily lead to another WCEP in the task's CFG without actually significantly reducing the WCET. If a task which cannot be optimized a lot is assigned a high partial system load and a very tight deadline, it may not be possible to repair it. On the other hand, in a system with a higher overall load, the partial loads might be distributed more fairly between the tasks or a task with a high optimization potential has the largest fraction of the overall load. Such statistical spikes could be eliminated by increasing the number of task sets in the analysis. However, the large time needed to compile, optimize and analysis each individual task set limits such evaluations in practice.

Fig. 12.2 shows the corresponding compilation times as box plots. For both schedulability-aware and schedulability-unaware optimizations, the runtime grows exponentially with the number of tasks. This stems from the fact, that the number of ILP variables and constraints grows linearly with the number of tasks in the task set, and the ILP's solving complexity grows exponentially with the number of variables in the ILP. However, despite the exponential growth, even for a system with 8 tasks, the median of the complete compilation process, including all necessary WCET analyses, is still only within a couple of minutes.

Additionally, it can be observed that the schedulability constraints obviously add on the time needed to optimize a given task set. For all task set sizes, optimization time is higher than for the schedulability-unaware optimization which solely tries to minimize the sum over all WCETs of all tasks. However, the increase is still only moderate. Most benchmarks with only 2 tasks still compile within 10 *s*. For large task sets, the average overall compilation time can rise by a factor of 10.

12.1.2. ILP Optimization on TriCore

The evaluation setup for the Infineon TriCore TC1796 architecture was chosen identical to the ARM7TDMI evaluation and has been described in Section 11.7.

Fig. 12.3 shows the results for the Infineon TriCore TC1796 as target platform. The graphs' axis setup and bar order is identical to ARM7TDMI in the previous section. The general trend of the results for TriCore TC1796 looks comparable to the ones for ARM7TDMI. As expected, the repair rates of the schedulability-aware optimizations for both DMS and EDF decrease approximately linearly with the unoptimized system load.

However, compared to the ARM7TDMI evaluation in the previous section, there are two major differences: First, the overall repair rates are significantly lower and second, the gap between the schedulability-aware optimization and the schedulability-unaware optimization is much smaller.

The reason for the lower repair rates stems from the fact that the Infineon TriCore TC1796 features a memory prefetching unit. This means, that – in contrast to the ARM7TDMI – not each instruction is fetched individually from memory. Instead, multiple instructions are fetched as a block. Only the first of these instructions suffers from the full latency of a fetch from the slow Flash memory. Subsequent instructions are then already buffered and can be accessed within one cycle. Therefore, the theoretical speedup by an SPM allocation is significantly smaller than for the ARM7TDMI.

This smaller optimization potential is also the reason for the smaller gap between the schedulability-aware and unaware optimizations. If the optimization potential is low, both optimizations tend to move the same basic blocks to the faster SPM.

Additionally, due to the fact that the schedulability-aware optimization is *constraint* based, any over-approximation of the WCET within the ILP framework may easily lead to an apparently unrepairable system. Obviously, if the optimization potential is large (like, e.g., for the ARM7TDMI platform), small over-approximations of the WCET can be compensated by the optimization. Potential reasons for such over-approximations have been discussed in Section 7.3.3. If, however, the optimization potential is relatively small, any over-approximation of the WCET can lead to an appparently unrepairable system using a constraint-based optimiza-

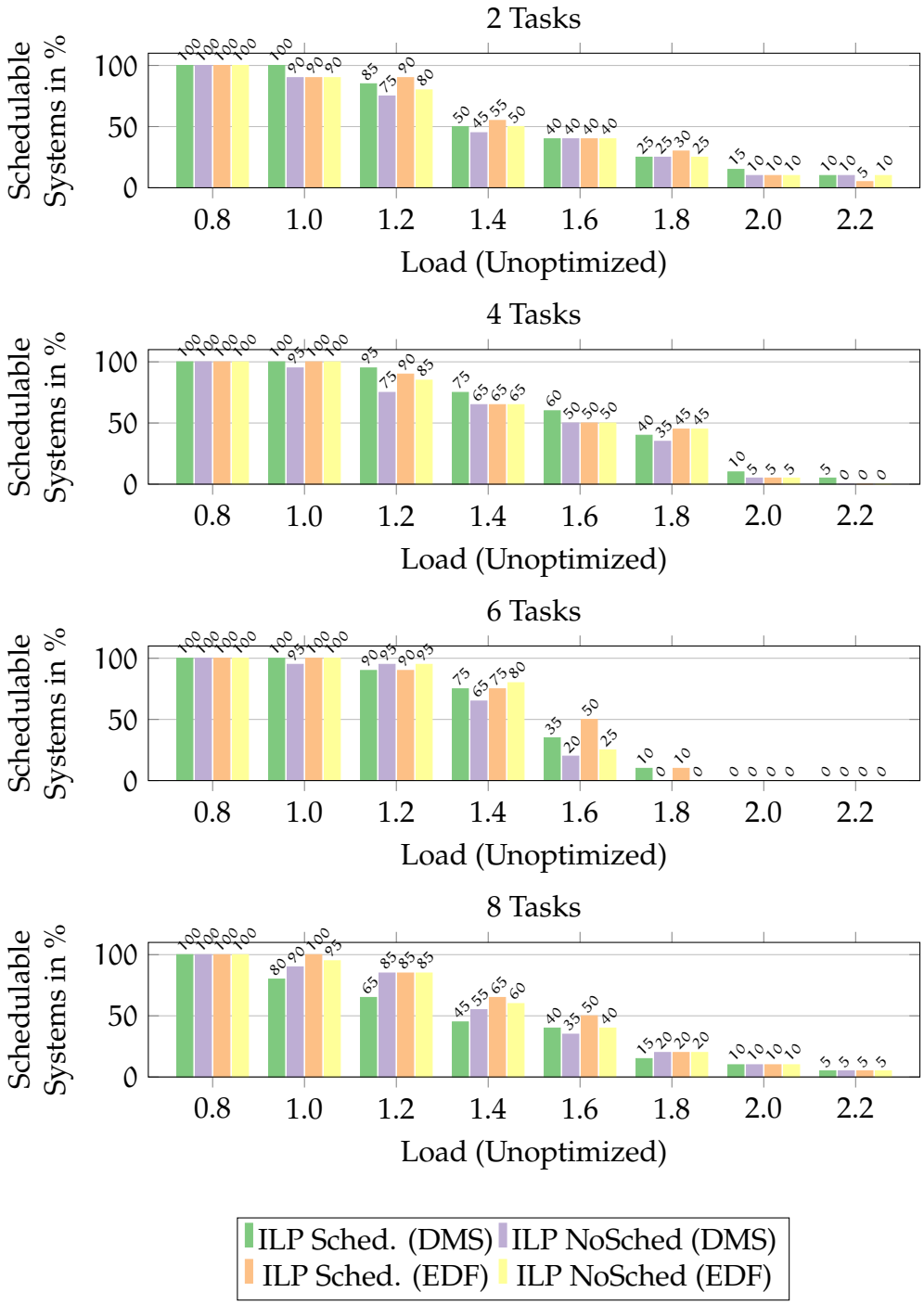
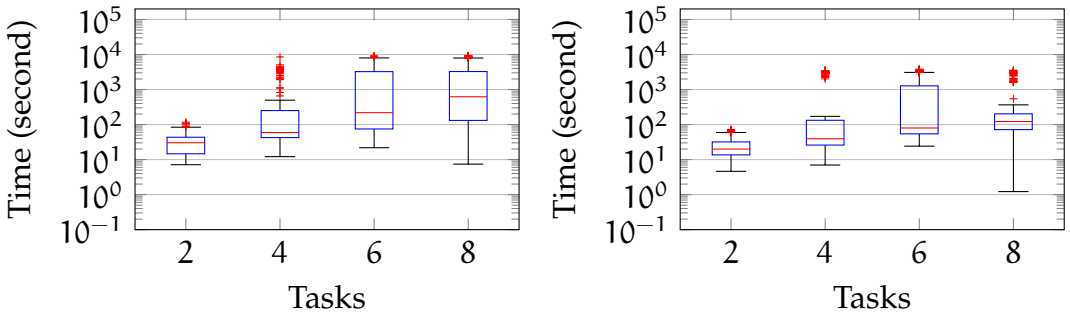


Figure 12.3: ILP-based repair rates for the SPM optimization on the TC1796 architecture. The SPM size is 40 % of the size of each task set. The graphs compare the repair rates for ILPs with and without schedulability constraints for both EDF and DMS scheduling.



(a) Schedulability-aware (b) Schedulability-unaware

Figure 12.4: Compilation times for ILP-based SPM allocation for the TC1796 architecture (40 % SPM size), including the whole compilation process and all WCET analyses.

tion framework. On the other hand, the optimization without schedulability-awareness is simply performing a best-effort optimization. If a system is just barely unschedulable, *any* WCET-aware optimization might lead to a schedulable system. Thus, in these corner-cases the schedulability *un*-aware optimization can be able to repair a system even if the absolute numbers on the WCETs are over-approximated.

However, apart from few outliers (e.g., for 8 tasks with an original load of 1.0 and 1.2 with DMS scheduling), schedulability-aware optimization is either better or at least as good as the schedulability-unaware optimization.

When comparing the scheduling algorithms, EDF is not able to outperform DMS. For larger task sets, EDF achieves better repair rates compared to DMS. E.g., for 8 tasks and a load of 1.6, 50 % of the systems could be repaired using EDF, compared to 40 % when using DMS. However, for smaller task sets, this may differ. While EDF slightly outperforms DMS for the smallest task set with only 2 tasks, DMS outperforms EDF for the 4 task systems. E.g., at an original load of 1.6, 60 % of the task sets could be repaired using DMS but only 50 % when using EDF. The number of evaluated task sets is still too low to state that DMS outperforms EDF, instead it rather shows that both are performing equally well on average.

Fig. 12.4 shows the corresponding compilation times as box plots. Analogous to the ARM7TDMI architecture, this includes the whole compilation process including all WCET analyses. For the schedulability-aware optimization, the complexity of the ILP grows linearly with the number of tasks for which a schedulability test has to be performed. As a result, the compilation times grow exponentially. However, even for a system with 8 tasks, the median of the compilation time is still within the range of a couple of minutes only.

For the schedulability *un*aware optimization, compilation times also grow exponentially. However, due to the massively simpler ILP, this increase is very

small. It can be observed that the randomly assembled task sets lead to an anomaly in the case of task sets with 6 tasks. Here, some ILPs take a significant amount of time to solve, such that the 75 % quartiles heavily outweight the results for the other task set sizes. Furthermore, it can be seen that (apart from the 6 task case), the variance in execution time is very small. This stems from the fact that without schedulability-awareness, the ILP solver will always return the same result, no matter what activation period, deadline or jitter was assigned to a task. Therefore, apart from solver-internal variances or load variances on the server running the evaluations, there are no expected differences between solving one task set for different original loads.

12.1.3. Evolutionary Algorithm on ARM7TDMI

This section evaluates the performance of the genetic algorithm proposed in Chapter 10 for the ARM7TDMI platform. In order to be able to compare the results with the ILP-based optimizations from Section 12.1.1, all system parameters are adopted from the ILP optimization. This especially means that for each given number of tasks in a task set and a given unoptimized load, the identical systems with the identical timing parameters were used for the evaluation. As a result, the results from Section 12.1.2 and this section can be directly compared in order to give an impression of the optimization quality and runtime of both approaches.

Fig. 12.5 shows the repair rates when performing the genetic algorithm. The first two bars for each target load show the results for DMS scheduling when using either a scaling factor or the number of tasks which have to be removed as a fitness function (cf. Section 10.5). Overall, the evaluation shows that the scaling factor based approach leads to significantly better or at least not worse repair rates than the task removal heuristic. Only for very few systems (2 tasks with a load of 1.6, 4 tasks with a load of 1.6 or 6 tasks with an original load of 1.2), the task removal heuristic performs slightly better than the scaling factor for at least one scheduling algorithm. On the other hand, for several setups, the task removal leads to significantly worse repair rates. This especially holds for the large task set with 8 tasks and DMS scheduling. For an original load of just 1.0, the task removal heuristic can only repair 45 % of the task sets with DMS scheduling, while the scaling factor-based approach is able to repair 85 % of the evaluated systems. For a load of 1.4, the scaling factor based approach can even repair four times more systems than the task removal heuristic. For a load of 1.6, the scaling factor-based approach can repair twice as many systems as the task removal heuristic.

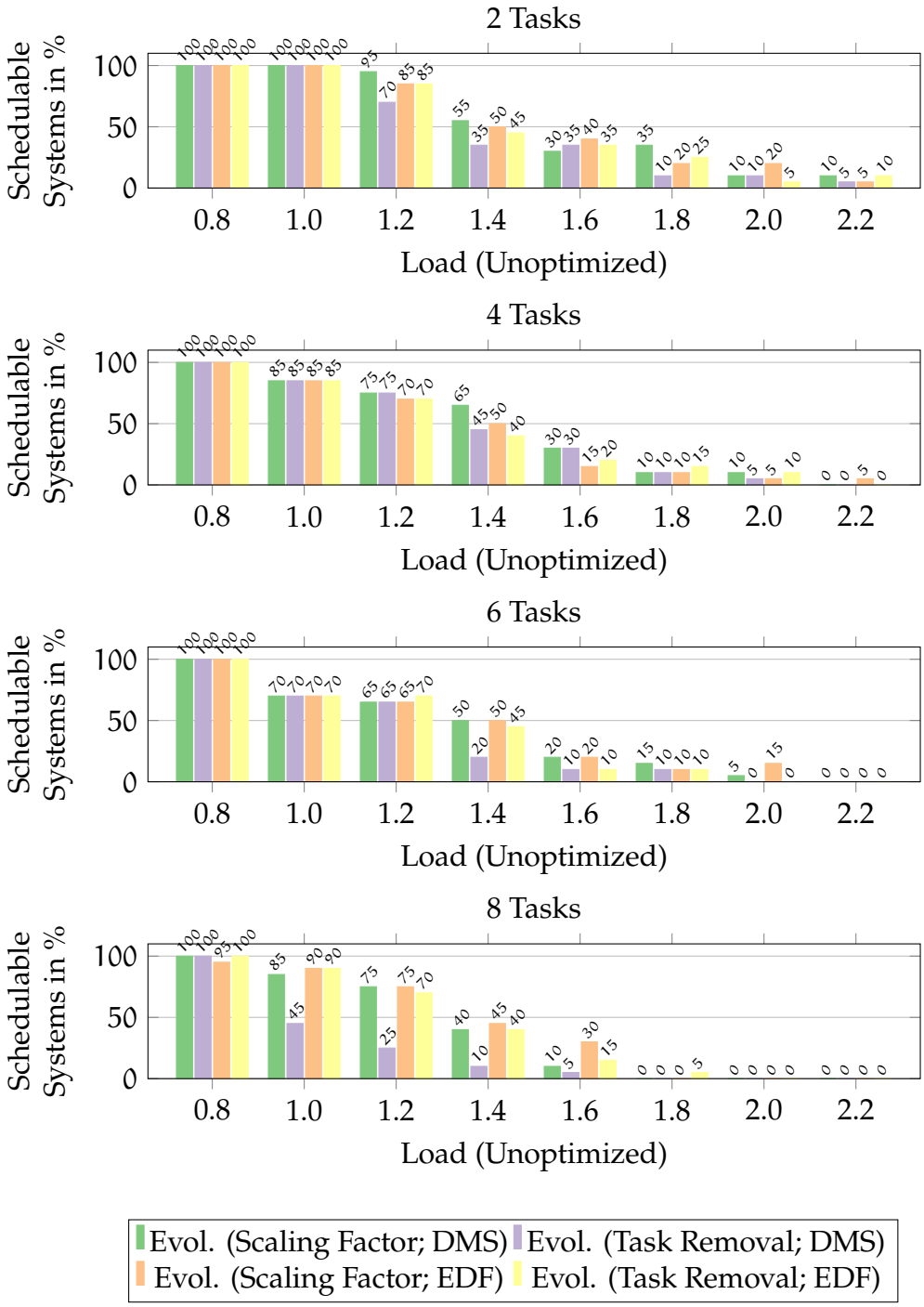
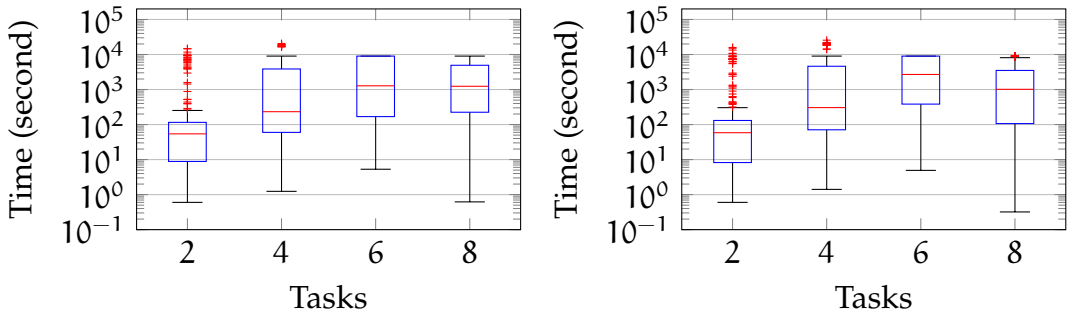


Figure 12.5: Evolutionary algorithm based repair rates for the SPM optimization on the ARM7TDMI architecture. The SPM size is 40 % of the size of each task set. The graphs compare the repair rates for two different fitness functions for both DMS and EDF scheduling.



(a) Scaling Factor Based Fitness Function (b) Task Removal Based Fitness Function

Figure 12.6: Compilation times for the evolutionary algorithm based SPM allocation for the ARM7TDMI target, including all WCET analyses.

In general, it can be observed that the gap is especially high for DMS scheduling. For EDF scheduling, the scaling factor-based approach is still slightly better but usually only by about 5 % to 10 %.

In comparison to the ILP-based optimizations it can be seen that the genetic algorithm performs significantly worse. The genetic algorithm performs even worse than the reference ILP optimization *without* awareness of the system's schedulability parameters. E.g., for the largest task set with 8 tasks and a system load of 2.0, the genetic algorithm is not able to repair *any* task set. In contrast, the ILP based optimization *without* schedulability-awareness was able to repair 70 % of the task sets using DMS scheduling and 80 % using EDF. The schedulability-*aware* ILP could even repair 75 % of the task sets under DMS scheduling and 95 % using EDF. Even for the smallest task sets with only two tasks, the genetic algorithm could only optimize 10 % of the systems for DMS and 20 % for EDF scheduling at an initial load of 2.0. On the other hand, the schedulability-aware ILP optimization framework could repair 95 % of the task sets for each scheduling algorithm.

Fig. 12.6 shows the results on the runtime of the genetic optimization. It can be seen that the runtime does not heavily differ between the task removal heuristic and the scaling factor-based approach. For both approaches, a significant amount of evaluations hits the two hour time cap. Even for the task sets consisting of only two tasks, both approaches have single evaluations which do not finish within the time limit. In contrast, for the ILP-based approaches all compilation times were clearly under 1 000 s for such small systems. The few number of task sets which can be solved within one second are systems which are already schedulable without any optimization (i.e., the systems with an original load of 0.8). For these systems, the initial population of the genetic algorithm already features a schedulable system, thus the optimization terminates instantaneously.

12.1.4. Evolutionary Algorithm on TriCore

In this section, the performance of the genetic algorithm is evaluated for the Infineon TriCore architecture. The results correspond to Section 12.1.2 where the same setup was evaluated using the ILP-based approach.

Fig. 12.7 shows the results on the repair rate for the genetic algorithm. For the different task set sizes, the graphs compare the number of systems which could be repaired with either the scaling factor based fitness function or the ILP based task removal heuristic (cf. Section 10.5). The optimizations were performed for both DMS and EDF scheduling.

In consistence with the ARM7TDMI results, the most obvious observation is that the genetic algorithm performs significantly worse than the ILP-based optimization depicted in Fig. 12.3. The genetic approach is barely able to repair *any* task set with an initial load of 1.4 or higher. At a load of 1.0, the genetic algorithm is still able to repair a significant amount of systems, but even then, the repair rate is clearly outperformed by the ILP approach. This stems from the fact that for each individual, a WCET analysis must be performed in order to determine the fitness of the individual. This analysis is very time consuming, such that the optimization runs into its 2 h timeout without being able to evaluate a significant amount of individuals

The comparison of the ILP-based optimizations for ARM7TDMI and TriCore already showed the lower optimization potential of the SPM optimization for the more sophisticated TC1796 architecture. This can also be seen for the genetic algorithm, where the repair rates are significantly lower than for the genetic algorithm on the ARM7TDMI architecture. For those small loads for which the genetic algorithm does show any significant repair rates, the scaling factor-based approach slightly outperforms the task removal-based heuristic. However, this gap is smaller than observed with the ARM7TDMI-based setup. E.g., for the task set consisting of 8 tasks and DMS scheduling, 60 % of the systems could be repaired using the scaling factor-based approach, compared to 45 % using the task removal heuristic. In contrast for ARM7TDMI, in this scenario 85 % could be repaired with the scaling-factor as fitness function, but only 45 % using the task removal heuristic. For larger loads, the relative difference between scaling factor-based fitness function and task removal heuristic becomes larger. E.g., three times more systems could be repaired with the scaling factor-based approach for DMS scheduling and an original load of 1.2 in the 8-task system with the task removal heuristic. However, the overall repair rates of 15 % and 5 % are so low that these relative numbers do not have much significance.

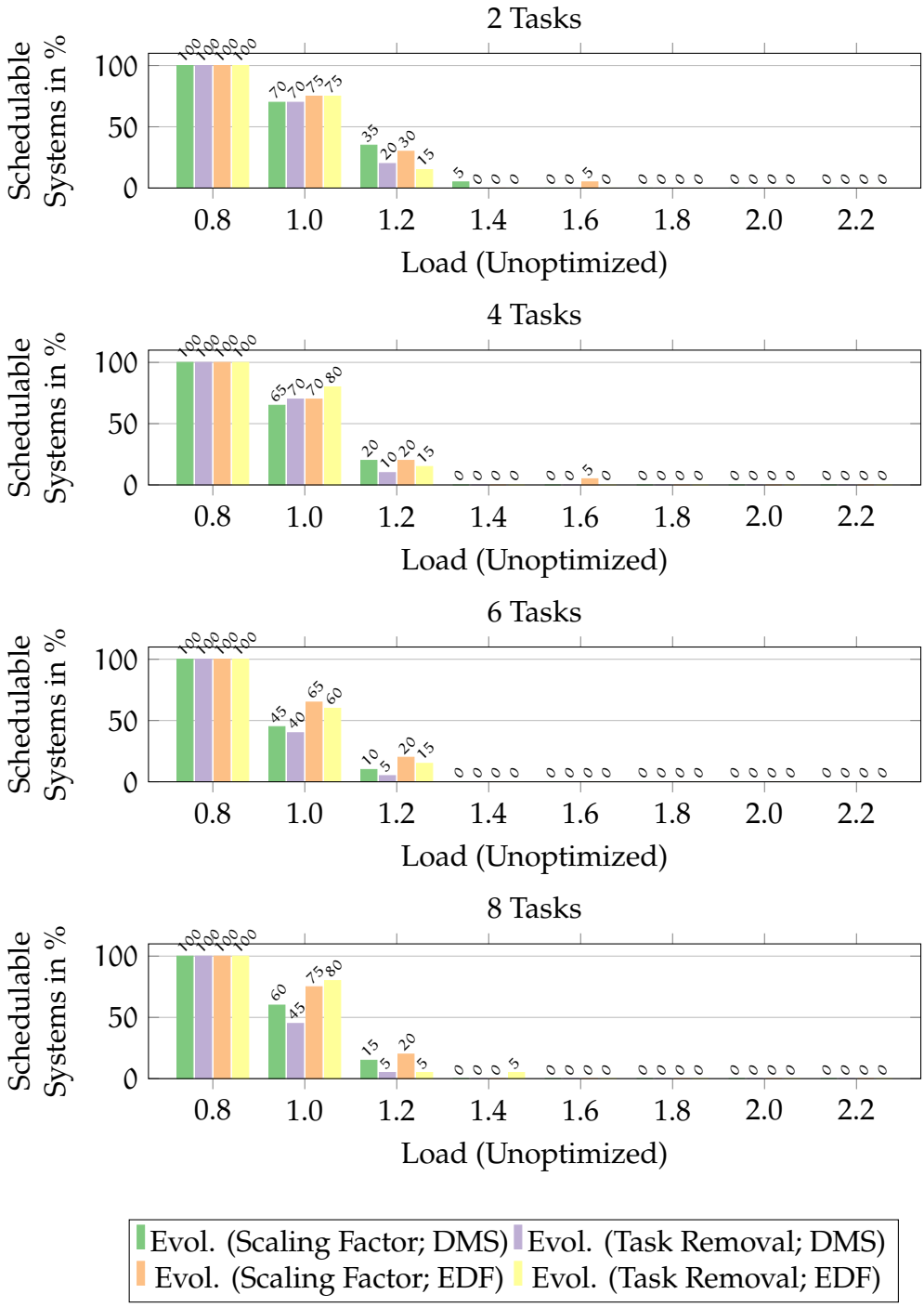
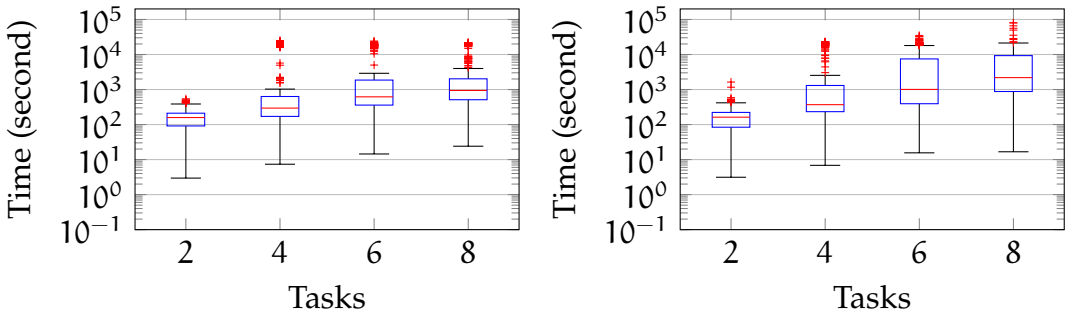


Figure 12.7: Evolutionary algorithm based repair rates for the SPM optimization on the TC1796 architecture. The SPM size is 40 % of the size of each task set. The graphs compare the repair rates for two different fitness functions for both DMS and EDF scheduling.



(a) Scaling Factor Based Fitness Function (b) Task Removal Based Fitness Function

Figure 12.8: Compilation times for the evolutionary algorithm based SPM allocation for TC1796, including the all WCET analyses.

Fig. 12.8 shows the execution times of the genetic algorithm. The evaluation times do not significantly differ from the ARM7TDMI case. For the smallest task sets, no evaluation hits the time limit, yet the repair rates of the genetic algorithm have been shown to be bad. These cases stem from the fact that the genetic algorithm preliminarily aborts if it cannot find any improved solution for two generations. These early terminations suggest that the random selection and recombination approach of the genetic algorithm is generally not ideal in order to find good solutions for a schedulability-aware optimization. The reason for this may reside in the general problem of WCET optimization: Optimizing any part of the program which is *not* part of the WCEP will not change the result in any way. At the same time, optimizing any part of the program which *is* part of the WCEP may lead to an apparently random change of the WCEP, rendering previous SPM allocations useless. The ILP-based optimization approaches tackle this issue by implicitly modeling the WCEP through each task of the task set at any time of the optimization (cf. Chapter 7). The genetic algorithm, on the other hand, has no such means. Each individual is only evaluated based on its WCET. Path information cannot be encoded into the fitness, making it hard to impossible for any selection algorithm to choose between the best individuals to combine.

12.2. Systems With Instruction Cache

This section evaluates the effect of instruction caches on the optimization. The evaluation is performed on the Infineon TriCore TC1796 architecture. The TriCore features a two-way set-associative instruction cache with LRU replacement policy. In order to account for effects of different benchmark sizes, the overall size of the cache is chosen to be as large as the SPM. I.e., in the following evaluation, both SPM and cache will each be 40 % of the size of the evaluated task set.

The goal of this section is twofold: First, the general applicability of the schedulability-aware ILP framework to cached systems is to be evaluated. Second, the impact of an explicit modeling of cache conflicts, as introduced in Chapter 9, is shown.

The evaluated task sets are identical to the ones without caches. Due to the fact that the unoptimized system already uses caches, the unoptimized WCETs of all tasks differ compared to the unoptimized systems in the previous sections. Because the deadlines and periods are generated randomly such that a given target load is met for the unoptimized system, these parameters also differ compared to the previous sections. However, for each optimization in this section, timing parameters are identical such that a direct comparison of the different optimizations is possible.

In order to show the impact of the explicit modeling of the cache, for each DMS and EDF, four optimizations are evaluated: The schedulability-aware ILP optimization *without* any explicit modeling of caches. The schedulability-*unaware* ILP optimization *without* any explicit modeling of caches. These two optimizations are identical to the ones performed in Section 12.1.2. Then, the schedulability-aware ILP optimization is performed with explicit modeling of both intra-task cache conflicts and inter-task cache conflicts (i.e., CRPD), as described in Chapter 9. Finally, the ILP optimization is performed with *only* inter-task CRPD analysis.

For the cache-*unaware* optimizations, the ILP does not guarantee to return *safe* results. By moving basic blocks to the SPM, the caching behavior may change, thus the WCET of a task within the ILP may be under-approximated. In case that the ILP found an apparently schedulable solution, all WCETs of all tasks of this optimized task set were retrieved by using `aiT`. Then, UCB/ECB analysis was performed on this task set and schedulability was re-analyzed with all these inter- and intra-task cache conflicts. Only if this schedulability analysis based on the exact WCET analysis by `aiT` yields a schedulable system, the system was counted as "repairable" within the results.

Fig. 12.9 shows the results for DMS scheduling. The overall repair rates are significantly lower than for the non-cached systems evaluated in Section 12.1.2. This behavior is to be expected, since the cache further compensates for the slow Flash memory access times. As a result, the theoretical improvement of the schedulability behavior due to an SPM assignment is even further decreased. Yet, for the task sets containing 6 tasks, still 30 % of all task sets could be repaired with an unoptimized load of 160 %.

It can be seen that even for a load of 0.8, not all task sets can be repaired. This stems from additional context switching costs due to CRPD. Furthermore, it can be seen that, apart from low original system loads of 0.8 and 1.0, the schedulability *aware* ILP optimization outperforms the schedulability *unaware* reference. E.g.,

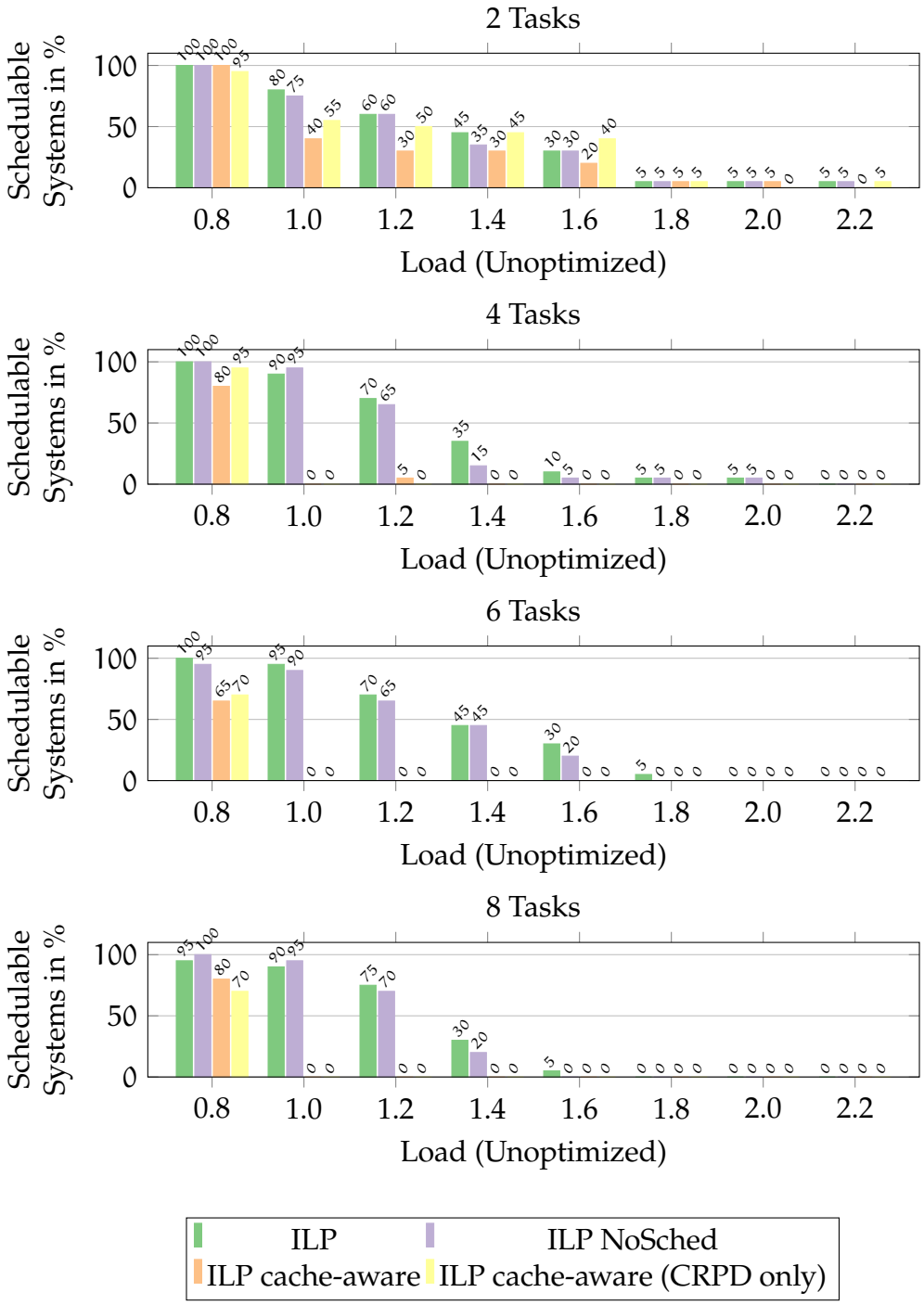


Figure 12.9: ILP-based repair rates for the SPM optimization on the TC1796 architecture with instruction cache using DMS scheduling. The SPM size and the cache size are 40 % of the size of each task set. The graphs compare the repair rates for different levels of explicit cache modeling.

for a task set size of 4 tasks, 35 % of the task sets could be repaired with the schedulability-aware approach for an original load of 1.4, while only 15 % could be repaired without schedulability-awareness.

When looking at the ILP optimizations *with* explicit cache modeling, it can easily be observed that the repair rate is generally very low. Only for the smallest task set size, the schedulability-aware ILP optimization with explicit cache modeling produce notable results. However, only for an unoptimized load of 1.6, the solely CRPD-aware ILP optimization is able to outperform the schedulability-aware optimization without explicit modeling of the cache. This result stems from the high complexity which is added by explicitly modeling the caching behavior within the ILP. As a result, most of the ILPs were terminated by timeout without a result after 2 h.

Fig. 12.10 shows the results for EDF scheduling. For EDF scheduling, the overall repair rates tend to be slightly higher for larger task sets. E.g., for 4 tasks and an original load of 1.2, the schedulability-aware ILP optimization was able to repair 75 % of the task sets, while only 70 % could be repaired under DMS scheduling. This gap further increases with increasing task set size. For 8 tasks and an original load of 1.6, 20 % of the evaluated task sets could be repaired using EDF while only 5 % were schedulable with DMS. Apart from this trend which is consistent with the results of the evaluation of the non-cached system, the results for EDF scheduling show the same trend as when using DMS.

It should be noted that – despite the smaller optimization potential and the potentially unsafe WCET estimation within the ILP – the repair rates of the cache-*unaware* ILPs still outperforms the repair rates of the genetic algorithm of the uncached system by far. E.g., for the largest task set with 8 tasks and an original load of 1.2, the genetic algorithm in Section 12.1.4 was only able to repair up to 15 % of the task sets using DMS and 20 % with EDF scheduling. On the other hand, the proposed ILP optimization framework was able to repair 75 % of the systems under DMS and 90 % of the evaluated task sets when using EDF scheduling.

Due to the fact that the genetic algorithm performs such bad compared to the ILP, with almost every single evaluation hitting the 2 h time limit, no explicit evaluation of the genetic algorithm on cached memory is given.

Fig. 12.11 shows the resulting timings. For the schedulability-aware ILP optimization *without* explicit cache modeling, the evaluation times are slightly larger than when applying the same optimization on an uncached system. This timing increase stems from the smaller optimization potential. Even without explicitly modeling the cache behavior in the ILP, some basic blocks are considered a cache hit by the initial WCET analysis of the unoptimized system. Assigning these basic blocks to the SPM will not lead to any improvement in the schedulability of the system, as a cache hit is equally fast as an SPM access on the TriCore

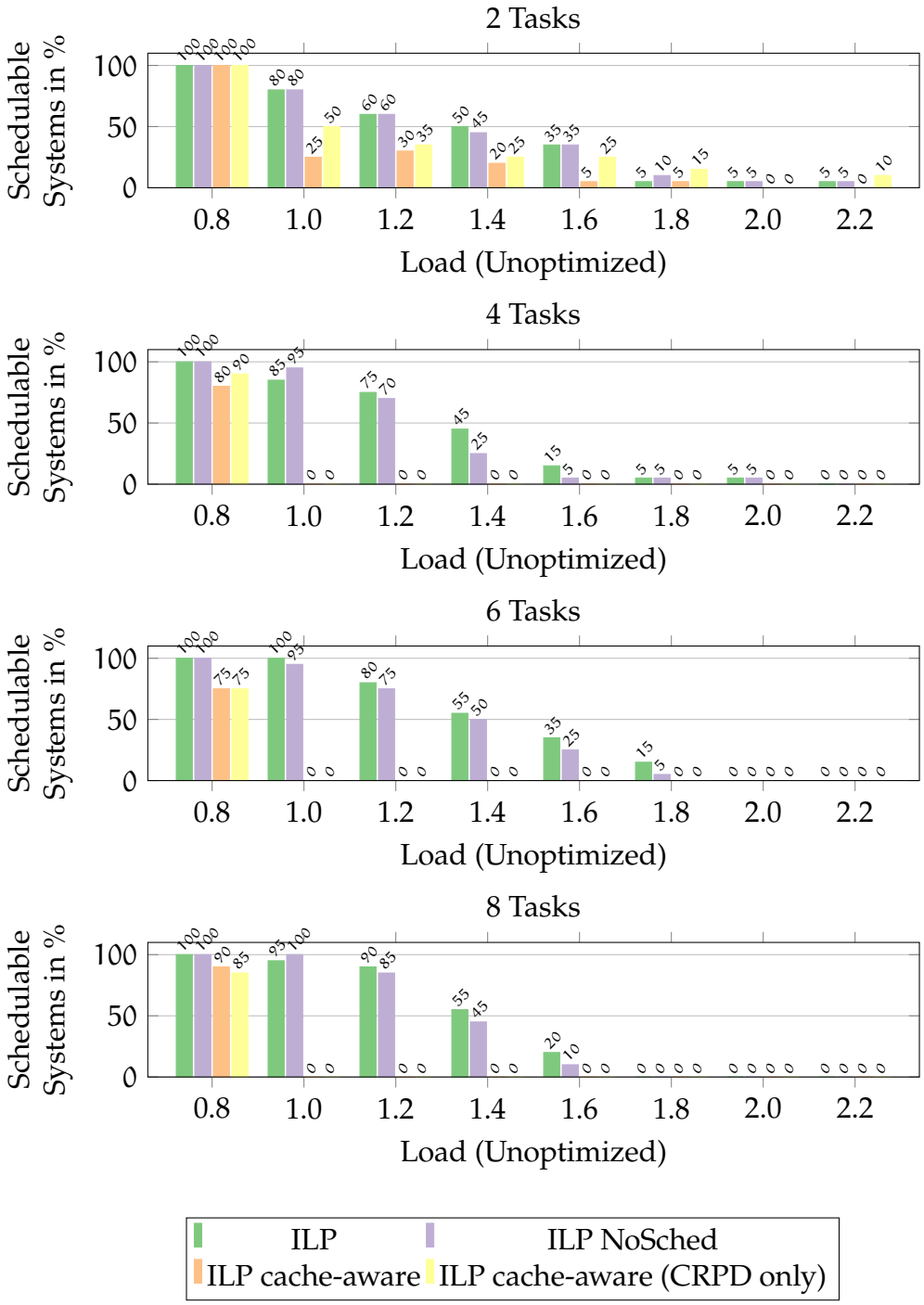
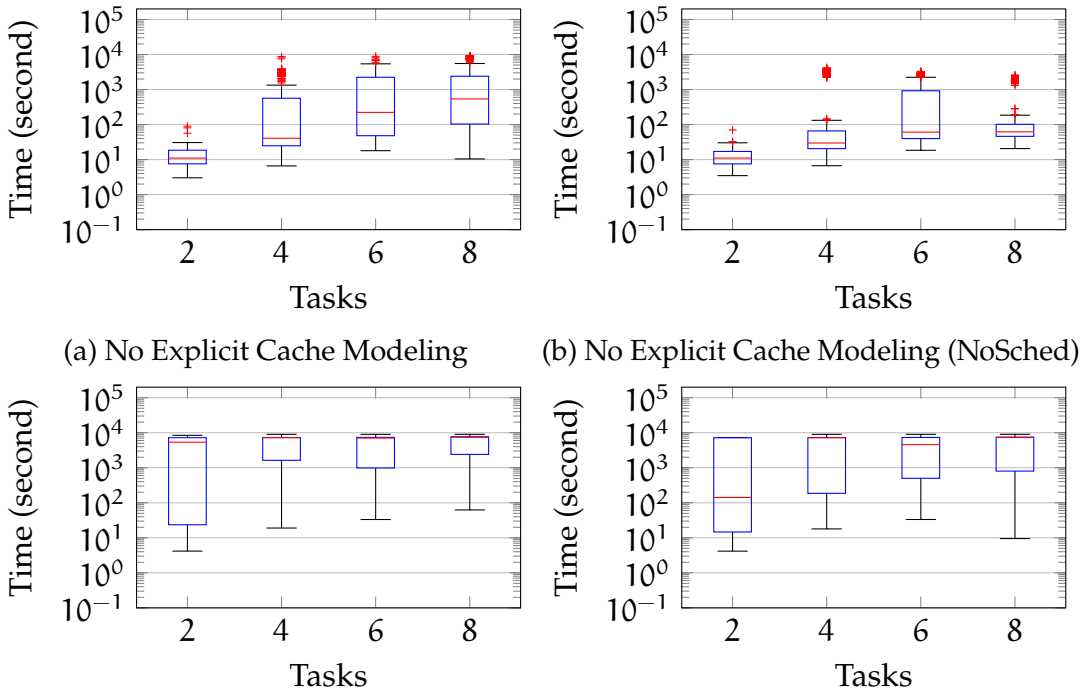


Figure 12.10: ILP-based repair rates for the SPM optimization on the TC1796 architecture with instruction cache using EDF scheduling. The SPM size and the cache size are 40 % of the size of each task set. The graphs compare the repair rates for different levels of explicit cache modeling.



(a) No Explicit Cache Modeling

(b) No Explicit Cache Modeling (NoSched)

(c) Explicit Cache Modeling

(d) Explicit CRPD Modeling

Figure 12.11: Compilation times for ILP-based SPM allocation for the TC1796 target with instruction caches. The depicted times comprises all WCET analyses.

architecture. Therefore, less feasible solutions (if any) exist, which lead to a schedulable system, which has a negative impact on the solver's performance.

The ILP without cache nor schedulability-awareness (cf. Section 12.2) does not have any noticable performance decrease compared to the execution on uncached memory.

As previously discussed for the repair rate, it can be seen that the vast majority of ILPs *with* explicit cache-awareness hit their two hour time limit. Only for the smallest task sets with two tasks, the median of the ILP optimization with only CRPD modeling is at about 100 s. The whiskers of the box plots indicate that for each task set, few systems are always solved within a couple of seconds. These stem from the systems at a low original system load of 0.8 which were actually schedulable without any optimization at all and did therefore not need any optimization.

In conclusion, this section showed that the schedulability-aware ILP optimization framework proposed in this thesis can successfully be applied to cached systems. Due to the nature of the caches which accelerate access to the slow Flash

memory, the optimization potential of performing an SPM allocation is smaller than when applied to a non-cached system. As a result, the overall repair rates are smaller than without caches. Additionally, it could be shown that the additional overhead of an explicit modeling of the cache conflict behavior is not reasonable. While prior publications showed good results in single-tasking systems [LKF16], such modeling proved to result in a combinatorial problem which is computationally infeasible with today's ILP solvers and hardware.

However, even without explicit modeling of a cache, the ILP-based schedulability-aware framework proved to provide good results which even outperform the repair rates of the genetic approach on *un*cached memory.

12.3. Impact of Real-Time Schedulers

This section aims at illustrating the impact of a scheduling task on the system's overall schedulability. For this, the system setup from Section 11.7 is used as a basis. In detail, this means that for each task set size, the same task sets were used as in all the setups above. The Infineon TriCore TC1796 is used as a target architecture.

Then, a small DMS scheduler is created which is specifically tailored towards the number of tasks in each given task set, as discussed in Section 11.5. When calculating the relative SPM size, the size of the scheduler is *not* accounted for.

Due to the fact that this thesis covers event based systems, the scheduling task does not need any specific activation pattern. Instead, it is assumed that each time an event occurs, the scheduler task is executed. The scheduler then decides whether the triggered task is executed and preempts the currently running task (in case that it has a logically higher priority), or whether it must wait. The scheduling overhead is thus added on top of the unoptimized system. This means that the actual unoptimized load may be significantly higher than the depicted load of the net task set without scheduling overhead.

These overall systems are then optimized towards their schedulability using the ILP framework. Within the optimization framework, the scheduler is modeled as another task with highest priority and multiple activation periods which equal all activation periods of the regular tasks (cf. Section 11.5.2). An evaluation using the genetic approach was not performed, due to the low repair rates of the genetic algorithm even without the additional schedulability overhead.

Fig. 12.12 shows the result of the optimization. The repair rates are significantly lower than the ones in Section 12.1.2 which do not feature any scheduler overhead. Without scheduler overhead, all task sets were schedulable at a load of 0.8. Now, with the scheduler, only 80 % of the task sets can be repaired for the smallest task sets using the schedulability-aware ILP optimization framework, and only

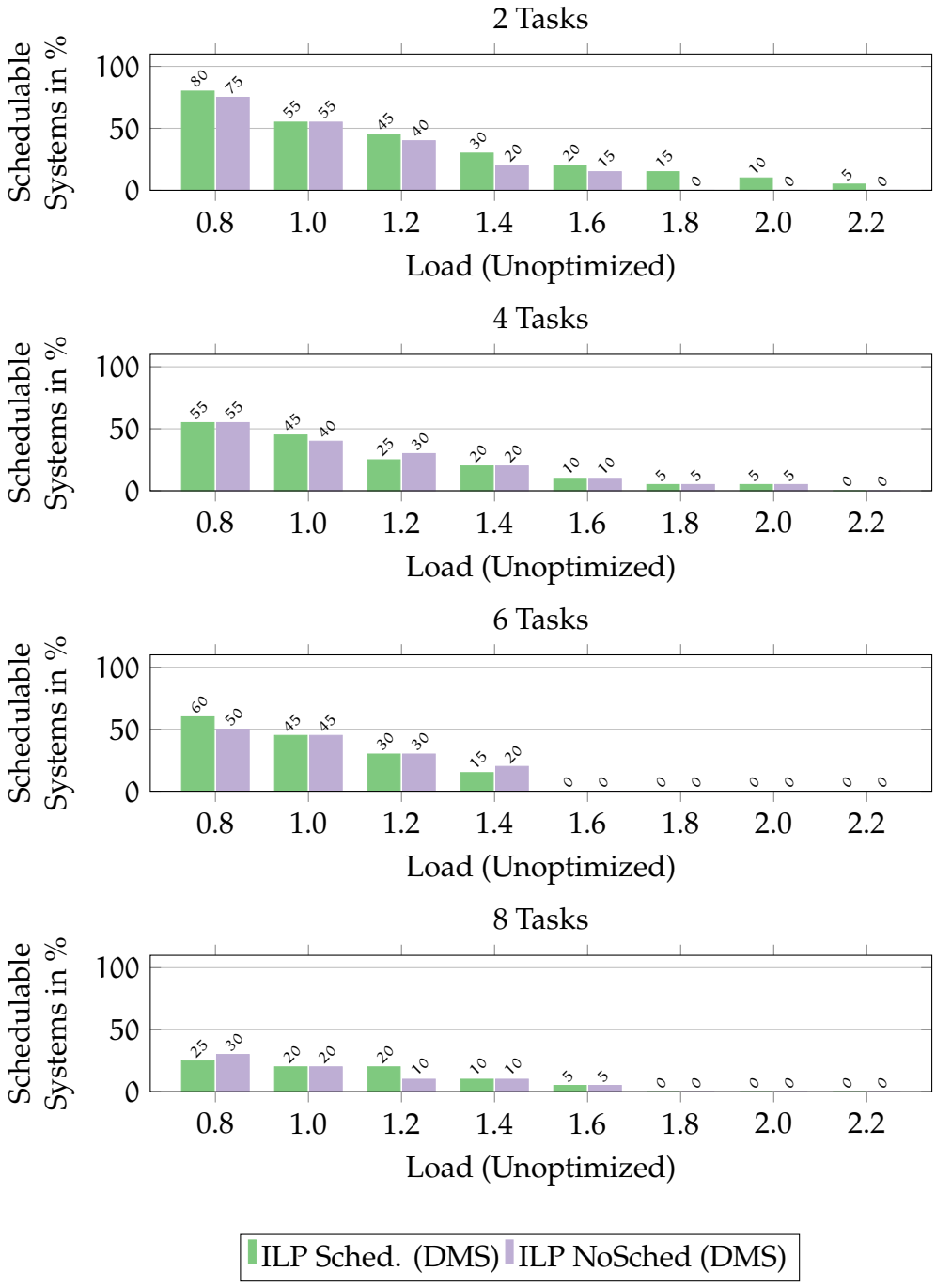


Figure 12.12: ILP-based repair rates for the SPM optimization on the TC1796 architecture with an additional scheduling task. The SPM size is 40 % of the size of each task set. The graphs compare the repair rates for ILPs with and without schedulability constraints.

75 % of the systems are schedulable after applying the schedulability-unaware optimization. Due to the increased number of scheduler preemptions, the repair rate significantly decreases for larger task sets. For the largest task set with 8 tasks, only 25 % of the task sets can be repaired using the schedulability-aware framework. For the ILP without schedulability-awareness, the repair rate is slightly higher with 30 %.

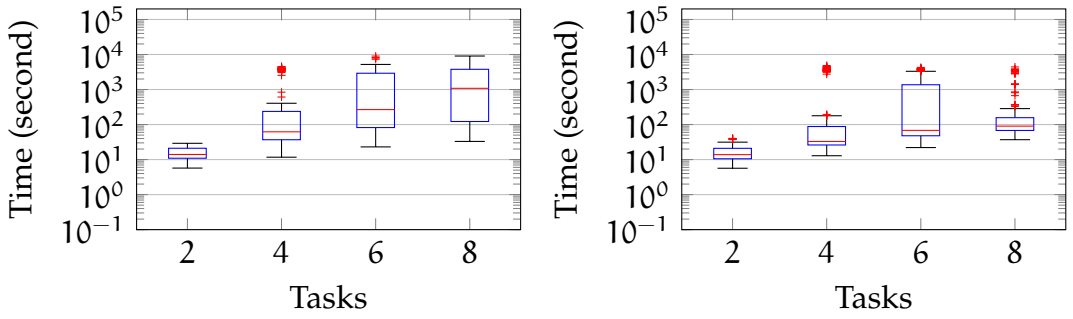
Overall, for the evaluated task sets, it can be seen that the repair rates are extremely lower than without the scheduler. E.g., for the task sets consisting of 6 tasks, no task set could be repaired at an initial load of 1.6 while 35 % of the task sets were repairable in Section 12.1.2 when the scheduler overhead was neglected.

It can be observed that for these small repair rates, the schedulability-aware ILP optimization framework cannot outperform the schedulability-unaware approach significantly. In the uttermost cases, however, the schedulability *aware* ILP optimization framework is either as good or slightly better. This stems from the fact that in these corner cases where hardly any system is repairable, any pessimism of the WCET estimation within the ILP can be fatal for repairing the system in a constraint-based optimization. Still, it can be seen that for the small task sets, the schedulability-aware approach is able to repair at least some task sets up to the load of 2.2 whereas the the reference optimization without schedulability-awareness fails to repair any system with an original load of 1.8 or higher.

However, the additional load itself which is inflicted by the scheduler is not the only reason for the bad repair rates. Despite the small size of the scheduler, the WCET of the unoptimized scheduler ranges from 1 132 CPU cycles for the smallest 2 task system to 3 684 CPU cycles for the largest task sets with 8 tasks. For comparison, the unoptimized WCET of, e.g., the `select` benchmark from the MRTC benchmark suite is 7 480 CPU cycles. Due to the nature of the scheduler, the actual task cannot be executed as long as it has not yet been dispatchted by the scheduler. This means that the execution of each task is delayed by the WCET of the scheduler. If such a small benchmark has a small period and tight deadline (relative to its own WCET), the resulting response time is so large that the task set simply cannot be repaired by any means.

Example 12.1 (Scheduler Impact on Small Tasks)
Assume the `select` *benchmark has the highest priority among all regular tasks in a task set of* 8 *tasks. Since the scheduler has the highest priority,* `select` *can therefore be denoted as* τ_1*. Further assume that* `select`*'s period is larger than its WCET. Then, if the scheduler is neglected, the WCRT* r_1 *is* $r_1 = c_1 = 7\,480$*.*

In a worst-case scenario (for `select`*), the* `select` *benchmark is triggered first for execution. Then, all lower-priority tasks are triggered subsequently afterwards. In this case, the scheduling task* τ_0 *is executed at least* 8 *times within one instance of* `select`*.*



(a) Schedulability-aware

(b) Schedulability-unaware

Figure 12.13: Compilation times for ILP-based SPM allocation for the TC1796 architecture with scheduler, including all WCET analyses.

When accounting for this, the WCRT increases to $r_1 \geqslant 7\,480 + 8 \cdot 3\,684 = 36\,952$. *This means that the WCRT is increased by at least a factor of* 4.94. *Depending on the activation frequencies of the other tasks, the number of preemptions by the scheduler may be even larger, leading to a further increase of* `select`*'s WCRT.*

Fig. 12.13 shows the execution times of the optimization. For the schedulability-aware ILP optimization framework, the results do not differ significantly from the results without accounting for the scheduler. The addition of the scheduling task does not add much complexity, as the scheduler's activation patern is harmonic to the activation periods of the regular tasks. Therefore, the event density function does not have any new points of discontinuity. As a result, the scheduling test needs not be performed for any additional time interval lengths. For the ILP without schedulability constraints, it can be seen that the whiskers are smaller. This simply stems from the fact that there are less task sets which are schedulable without any optimization necessary. Additionally, due to the fact that the ILP is about one magnitude faster without schedulability constraints, the additional task can add a larger relative increase of the compilation time than for the ILP with schedulability constraints.

Overall, this section illustrated that a scheduling task can be integrated within the schedulability-aware ILP optimization framework without any adjustments to the framework. Additionally, it showed that especially for task sets with some small tasks, neglecting the scheduler overhead can easily lead to a severe misprediction of the schedulability of the overall system.

12.4. Summary

This chapter evaluated the ILP-based schedulability-aware optimization framework which was proposed in Chapter 8. In order to illustrate both benefits and disadvantages, the approach was compared to an ILP-based WCET optimization which has no awareness of any effects on the schedulability of the overall system. Additionally, the framework was compared to an optimization approach based on a genetic algorithm which was proposed in Chapter 10. For the genetic approach, two different fitness functions were applied.

The ARM7TDMI architecture was used to provide an evaluation of the optimization framework on a relatively good predictable architecture. To show the performance on complex hardware, the Infineon TriCore TC1796 microcontroller was also evaluated. This platform provides a much smaller potential for memory-based optimizations as its complex architecture with memory block fetches, multiple pipelines and branch prediction can already compensate for slow memory access times.

Sections 12.1.1 to 12.1.4 evaluate the approaches for basic system setups with no caches and in case of negligible scheduling overhead. As a result, the ILP-based optimization outperforms the genetic approach by far for both architectures. For the ARM7TDMI-based architecture, the ILP-based schedulability-aware optimization framework also drastically outperforms the traditional ILP optimization. While having an inevitable increase in compilation time, the schedulability-aware optimization framework has a major impact on the repair rate. For the largest analyzed task sets and high system loads, the schedulability *aware* optimization framework can optimize more than twice as many task sets as the schedulability-*unaware* optimization for both fixed and dynamic priority scheduling. Even for an unoptimized system load of 300 %, half of the task sets could be repaired by the optimization framework for task sets consisting of 8 tasks.

The analysis of the TC1796 showed that the benefit of the schedulability-aware optimization framework decreases compared to ARM7TDMI. The main reasons for this are a significantly smaller optimization potential due to the TC1796's hardware features and more pessimistic WCET estimates within the ILP optimization framework.

Nevertheless, the schedulability-aware optimization framework provides significantly better repair rates in multiple scenarios. In a real-world scenario based on such complex architectures a system designer might first try to optimize the system without accounting for the schedulability constraints. If this approach does not lead so a schedulable system, schedulability constraints can be added to the existing ILP.

Section 12.2 investigated the impact of *caches* on the optimization framework. An extension of the ILP framework to explicitly model instruction caches was presented in Chapter 9 and has previously been successfully applied to single-tasking systems [LKF16]. The evaluation showed that the overhead of the explicit modeling of cache conflicts within the ILP leads to such an increase in the complexity of the resulting linear program that most evaluations resulted in a timeout.

However, the evaluation of cached systems also showed that the schedulability-aware ILP framework shows good repair rates even *without* explicit modeling of caches. I.e., while an explicit modeling of caching behavior within the ILP is computationally clearly infeasible, this is not needed in order to achieve noticeable repair rates up to original system loads of 160 %. Despite its imprecision, the proposed schedulability-aware ILP-based optimization framework significantly outperforms the genetic optimization.

It is subsequently possible to use the approach for optimizations on complex embedded target architectures. The evaluation showed that even for an unoptimized load of 160 %, a significant number of systems can be repaired without having to exactly model any side effect which can be caused by the optimization.

Finally, Section 12.3 investigated the real impact of an additionally available scheduler. In order to examine this issue, a low-overhead fixed-priority scheduler was proposed in Section 11.5 and was implemented as part of a bachelor thesis [Fis18]. The scheduler is assumed to be *event-based*. I.e., each time an event is triggered, the scheduler is executed and subsequently assumes whether the respective task may be executed or not. The evaluation showed two things: First, the ILP-based optimization framework is able to optimize such systems without any additions to the framework. The scheduler can be modeled as the highest-priority task with respective activation patterns. From the optimization's point of view, accounting for the scheduler does not add more complexity as any other additional task would add. Second, the results show that, especially for hard real-time systems with very small tasks, the overhead of even such a small scheduler may easily be significantly larger than the WCETs of some of the actual tasks. As a result, when applying any schedulability-aware optimizations for hard real-time systems, neglecting the scheduler overhead may easily lead to an unsound prediction of the system's schedulability. Tight deadlines which are only slightly larger than a small task's WCET may be impossible to hold, even in simple setups.

13. Conclusion and Future Work

13.1. Summary

This thesis presented a novel approach to integrate a multi-tasking hard real-time system's key property – schedulability – as a design constraint into compiler optimizations. In the past, code-level based optimizations focused on optimizing one single WCET of one selected task. However, this is no longer sufficient in a multi-tasking environment where multiple tasks must all finish execution prior to their respective deadlines. Optimizing one given task may easily influence the timing behavior of other tasks in the system – either directly by the time the task prevents a lower-priority task from execution – or indirectly, if the optimization of one task prevents other tasks from being optimized due to limited hardware capabilities. Therefore, it is of crucial importance to decide which task should be optimized to which extent in order to generate an overall schedulable system.

As a consequence, this thesis moves the focus from minimizing one single WCET of a specifically selected task to a holistic optimization which spans over the whole task set of a multi-tasking system. Chapter 8 shows how this can be achieved by proposing an ILP-based optimization framework. The framework incorporates system-level timing analyses into a code-level optimization platform. Timing analyses are based on the concept of event-triggered systems which are modeled using density and interval functions. This allows for the schedulability-aware optimization of arbitrarily triggered tasks with arbitrary deadlines. The approach supports the optimization of systems scheduled under both fixed-priority and dynamic-priority scheduling algorithms. The framework thereby closes the gap between system-level analysis techniques on the one hand and code-level based compiler optimizations on the other hand. Chapter 9 shows how the framework can be extended to precisely include the effects of commonly found caches using the LRU replacement policy into the schedulability-aware optimization framework.

As an alternative to the ILP, a genetic algorithm was proposed in Chapter 10 in order to compare the ILP-based approach both with regard to the framework's runtime and its optimization results.

The WCET-aware C compiler framework `WCC` was used to evaluate the proposed approaches. Evaluations were subsequently performed in Chapter 12 for both the

Infineon TriCore and the ARM7TDMI target architectures. The results show that the ILP-based approach is able to repair a significant amount of systems which initially did not comply with their timing constraints. It outperforms the genetic approach both with regard to optimization quality and runtime by far.

13.2. Future Work

Based on the results of this thesis, multiple directions of future research are promising:

Task Dependencies This thesis focused on optimizing *independent* tasks. A first approach on future research could thus be to extend the schedulability-aware ILP constraints towards explicitly modeling task dependencies.

Multi-Core Systems This thesis tackles single-core systems. It could be shown successfully, that both of the approaches proposed in this thesis can be applied to multi-core systems which are accessing a shared memory using a Time Division Multiple Access (TDMA)-scheduled bus [LOF20]. Despite showing the general applicability, the results also showed that the computational complexity of such an optimization framework is very high. Additionally, an ILP-based tight bus-analysis for single-task multi-core optimizations has been proposed in [OLF17]. However, integrating this into a multi-tasking setup is easily computationally too expensive even for trivial systems with two cores and two tasks per core.

Future research could therefore try to find a compromise between the micro-architectural view which is needed for code-level optimizations and the system-level view which allows for a fast analysis of multi-core timing behavior. A first promising approach on how this could be integrated into `WCC` has been previously presented by Oehlert et al. [OSF18].

Multi-Objective Optimizations Apart from the mandatory requirement of complying with all timing constraints, many embedded real-time systems are subject to further design constraints. On battery powered systems, *energy consumption* plays a major role. An energy analysis framework was included into `WCC` by the author of this thesis [RLF18], and first tests on optimizing both schedulability and average-case energy consumption using the framework proposed in this thesis look promising.

Further design restrictions stem from limited memories in embedded systems. An embedded system should therefore not only be optimized for performance, but also for *code size*. A possible extension of the thesis at hand is to remodel it into *multi-criteria optimizations* [RLF14] focusing WCET

and code size. The general applicability of this idea has previously been shown [MLF18; MLF19].

This way, an optimizing compiler could, e.g., optimize a system for the least energy consumption while also ensuring schedulability and code size constraints. Apart from this, future work could also comprise adding reliability optimizations like, e.g., memory aging as additional design constraints. This was outlined in [OLF18b].

Sytem-Level Design Integration `WCC` has previously been used successfully in order to automatically generate code for timing-predictable hardware architectures [Pag+18]. Additionally, it has also been shown that `WCC`'s capabilities can already be applied in order to analyze external buses like, e.g., CAN or FlexRay [OF18]. Combining these results with the schedulability-aware optimization framework proposed in this thesis could lead one step forward towards a holistic synthesis tool which is able to propagate system-level requirements down to fine-grained code-level optimizations. The general idea how such a framework could interact with `WCC` has originally been outlined in [Lup+13].

List of Figures

List of Tables

Bibliography

[Abs19] AbsInt Angewandte Informatik, GmbH. *TimeWeaver: Hybrid Worst-Case Timing Analysis*. 2019.

[ADM12] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems". In: *Real-Time Systems* 48.5 (June 2012), pp. 499–526. DOI: 10.1007/s11241-012-9152-2.

[AG04] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge / United Kingdom: Cambridge University Press, 2004. ISBN: 0521607655.

[Aho+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, & Tools*. Ed. by Michael Hirsch. 2nd Edition. Boston, MA / USA: Pearson Education, Inc., 2007. ISBN: 9780321486813.

[All70] Frances E. Allen. "Control Flow Analysis". In: *ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization* 5 (July 1970), pp. 1–19. DOI: 10.1145/390013.808479.

[And98] Allan H. Anderson. *Scalable C and VHDL Simulators for a SAR Image Processor. User Guide*. Tech. rep. Lexington, Massachusetts / USA: Massachusetts Institute of Technology. Lincoln Laboratory, June 1998.

[ARM04] ARM Limited. *ARM7TDMI. Revision: r4p1. Technical Reference Manual*. 2004.

[AS04] Karsten Albers and Frank Slomka. "An Event Stream Driven Approximation for the Analysis of Real-Time Systems". In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*. Catania / Italy, June 2004, pp. 187–195. DOI: 10.1109/emrts.2004.1311020.

[Bar03] Sanjoy K. Baruah. "Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks". In: *Real-Time Systems* 24.1 (Jan. 2003), pp. 93–128. DOI: 10.1023/A:1021711220939.

[BB05] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30.1 (May 2005), pp. 129–154. DOI: 10.1007/s11241-005-0507-9.

[BC18] Martin Becker and Samarjit Chakraborty. "Optimizing Worst-Case Execution Times Using Mainstream Compilers". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Sankt Goar, Germany, May 2018, pp. 10–13. DOI: 10.1145/3207719.3207739.

[BDB06] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. "Sensitivity Analysis for Fixed-Priority Real-Time Systems". In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*. Dresden / Germany, July 2006, pp. 13–22. DOI: 10.1007/s11241-006-9010-1.

[BE00] Alan Burns and Stewart Edgar. "Predicting Computation Time for Advanced Processor Architectures". In: *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*. Stockholm / Sweden, June 2000, pp. 89–96. DOI: 10.1109/emrts.2000.853996.

[Bis17] Johannes Bisschop. *AIMMS Optimization Modeling*. Haarlem / Netherlands: AIMMS B.V., Apr. 2017. ISBN: 9781847539120.

[Ble+03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. "PISA — A Platform and Programming Language Independent Interface for Search Algorithms". In: *Proceedings of the Second International Converence on Evolutionary Multi-Criterion Optimization (EMO)*. Ed. by Carlos M. Fonseca, Peter J Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Vol. 2632. Lecture Notes in Computer Science (LNCS). Faro / Portugal: Springer, Apr. 2003, pp. 494–508. DOI: 10.1007/3-540-36970-8_35.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor". In: *Proceedings of 11th Real-Time Systems Symposium (RTSS)*. Lake Buena Vista, FL / USA, Dec. 1990, pp. 192–190. DOI: 10.1109/real.1990.128746.

[Bon+17] Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. "Early WCET Prediction Using Machine Learning". In: *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis (WCET)*. June 2017, 5:1–5:9. DOI: 10.4230/OASIcs.WCET.2017.5.

[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)*. Los Angeles, California / USA, Jan. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[Com19] Synopsys, Inc. *Synopsys Virtualizer*. 2019. URL: https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html.

[Cpl19] IBM Corporation. *IBM ILOG CPLEX Optimization Studio*. 2019. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[DAE12] Jonas Diemer, Philip Axer, and Rolf Ernst. "Compositional Performance Analysis in Python with pyCPA ". In: *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Pisa / Italy, July 2012, pp. 27–32.

[Dav+18] Robert I. Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. "An extensible framework for multicore response time analysis". In: *Real-Time Systems* 54.3 (July 2018), pp. 607–661. DOI: 10.1007/s11241-017-9285-4.

[DB11] Robert I. Davis and Alan Burns. "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Computing Surveys* 43.4 (Oct. 2011), 35:1–35:44. DOI: 10.1145/1978802.1978814.

[DOW55] George B. Dantzig, Alexander Orden, and Philip Wolfe. "The Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints". In: *Pacific Journal of Mathematics* 5.2 (Oct. 1955), pp. 183–195. DOI: 10.2140/pjm.1955.5.183.

[Duf88] Tom Duff. *Subject: Re: Explanation, please! Message-ID: <8144@alice.UUCP>*. Aug. 29, 1988. URL: https://www.lysator.liu.se/c/duffs-device.html (visited on 06/28/2018).

[Eng02] Jakob Engblom. "Processor Pipelines and Static Worst-Case Execution Time Analysis". PhD thesis. Uppsala / Sweden: Acta Universitatis Upsaliensis, 2002.

[Fal+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research". In: *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Toulouse / France, 2016, 2:1–2:10. DOI: 10.4230/OASIcs.WCET.2016.2.

[FH04] Christian Ferdinand and Reinhold Heckmann. "aiT: Worst-Case Execution Time Prediction by Static Program Analysis". In: *Building the Information Society*. IFIP International Federation for Information Processing 156 (Jan. 2004), pp. 377–383. DOI: 10.1007/978-1-4020-8157-6_29.

[Fis18] Thilo Fischer. "Automatic Scheduler Generation for Hard Real-Time Systems using a WCET-Aware Compiler". Bachelor Thesis. Hamburg / Germany: Institute of Embedded Systems, Hamburg University of Technology, Dec. 2018.

[FK09] Heiko Falk and Jan C. Kleinsorge. "Optimal Static WCET-aware Scratchpad Allocation of Program Code". In: *Proceedings of the 46th Design Automation Conference (DAC)*. San Francisco / USA, July 2009, pp. 732–737. DOI: 10.1145/1629911.1630101.

[FK11] Heiko Falk and Helena Kotthaus. "WCET-driven Cache-aware Code Positioning". In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. Taipei / Taiwan, Oct. 2011, pp. 145–154. DOI: 10.1145/2038698.2038722.

[FL10] Heiko Falk and Paul Lokuciejewski. "A compiler framework for the reduction of worst-case execution times". In: *Real-Time Systems* 46.2 (July 2010), pp. 251–300. DOI: 10.1007/s11241-010-9101-x.

[FS06] Heiko Falk and Martin Schwarzer. "Loop Nest Splitting for WCET-Optimization and Predictability Improvement". In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dresden / Germany, July 2006. DOI: 10.4230/OASIcs.WCET.2006.674.

[FSS11] Heiko Falk, Norman Schmitz, and Florian Schmoll. "WCET-aware Register Allocation Based on Integer-Linear Programming". In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*. Porto / Portugal, July 2011, pp. 13–22. DOI: 10.1109/ecrts.2011.10.

[FW99] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-Time Systems". In: *Real-Time Systems* 17.2-3 (Nov. 1999). Ed. by Wolfgang Halang and Reinhard Wilhelm, pp. 131–181. DOI: 10.1023/A:1008186323068.

[Gaj+09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design. Modeling, Synthesis and Verification*. New York, NY / USA: Springer Science+Business Media, LLC, 2009. DOI: 10.1007/978-1-4419-0504-8.

[Geb10] Gernot Gebhard. "Timing Anomalies Reloaded". In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Brussels / Belgium, July 2010, pp. 1–10. DOI: 10.4230/OASIcs.WCET.2010.1.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Repr. with corr. Reading, Mass / USA: Addison-Wesley Publishing Company, Inc., 1989. ISBN: 0201157675.

[Gom58] Ralph E. Gomory. "Outline of an Algorithm for Integer Solutions to Linear Programs". In: *Bulletin of the American Mathematical Society* 64 (1958), pp. 275–278. DOI: 10.1090/s0002-9904-1958-10224-4.

[Gre93a] Klaus Gresser. "An Event Model for Deadline Verification of Hard Real-Time Systems". In: *Proceedings of 5th Fifth Euromicro Workshop on Real-Time Systems*. Oulu / Finland, June 1993, pp. 118–123. DOI: 10.1109/emwrt.1993.639067.

[Gre93b] Klaus Gresser. "Echtzeitnachweis erreignisgesteuerter realzeitsysteme". PhD thesis. Munich / Germany: Technische Universität München / Germany, 1993.

[Gue11] Raphael Pereira Oliveira de Guerra. "A Gravitational Task Model for Target Sensitive Real-Time Applications". PhD thesis. Kaiserslautern / Germany: Technische Universität Kaiserslautern, 2011.

[Gur19] Gurobi Optimization, Inc. *Gurobi Optimizer*. 2019. URL: https://www.gurobi.com.

[Gus+03] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. "A Tool for Automatic Flow Analysis of C-programs for WCET Calculation". In: *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*. Ed. by Bob Werner. Los Alamitos, CA / USA, Jan. 2003, pp. 106–112. DOI: 10.1109/words.2003.1218072.

[Gus+06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution". In: *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*. Rio de Janeiro / Brazil, Dec. 2006, pp. 57–66. DOI: 10.1109/rtss.2006.12.

[Gus+10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. "The Mälardalen WCET Benchmarks – Past, Present and Future". In: *Proceedings of the 10th International Workshop an Worst-Case Execution Time Analysis (WCET)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). Brussels / Belgium, July 2010, pp. 137–147. DOI: 10.4230/OASIcs.WCET.2010.136.

[Gut+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC)*. Austin, Texas / USA, Dec. 2001, pp. 3–14. DOI: 10.1109/wwc.2001.990739.

[Hea+98] Christopher Healy, Mikael Sjodin, Viresh Rustagi, and David Whalley. "Bounding Loop Iterations for Timing Analysis". In: *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*. Denver, CO / USA, June 1998, pp. 12–21. DOI: 10.1109/rttas.1998.683183.

[Hen+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. "System level performance analysis - the SymTA/S approach". In: *IEE Proceedings - Computers and Digital Techniques* 152.2 (Mar. 2005), pp. 148–166. DOI: 10.1049/pbcs018e_ch2.

[HLS00] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. "Using a worst-case execution time tool for real-time verification of the debie software". In: *European Space Agency-Publications-ESA SP* 457 (2000), pp. 307–312.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Ed. by Todd Green and Nate McFadden. 5th edition. Waltham, MA / USA: Morgan Kaufmann, 2012. ISBN: 9780123838728.

[HTT89] Jiawei Hong, Xiaonan Tan, and Don Towsley. "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System". In: *IEEE Transactions on Computers* 38.12 (Dec. 1989), pp. 1736–1744. DOI: 10.1109/12.40851.

[INF08] Infineon Technologies AG. *Data Sheet TC1796. 32-Bit Single-Chip Microcontroller. TriCore.* Version V1.0. Munich / Germany, Apr. 2008.

[JP86] Mathai Joseph and Paritosh Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29.5 (Jan. 1986), pp. 390–395. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA / USA: Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[Kar84] Narendra Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". In: *Combinatorica* 4.4 (Dec. 1984), pp. 373–395. DOI: 10.1007/bf02579150.

[Kel15] Timon Kelter. "WCET Analysis and Optimization for Multi-Core Real-Time Systems". PhD thesis. Dortmund / Germany: TU Dortmund University, 2015.

[KFM11] Jan C. Kleinsorge, Heiko Falk, and Peter Marwedel. "A Synergetic Approach to Accurate Analysis of Cache-Related Preemption Delay". In: *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. Taipei / Taiwan, Oct. 2011, pp. 329–338. DOI: 10.1145/2038642.2038693.

[Kim+12] Jinwoo Kim, Hyunok Oh, Hyojin Ha, Shin-haeng Kang, Junchul Choi, and Soonhoi Ha. "An ILP-based Worst-case Performance Analysis Technique for Distributed Real-time Embedded Systems". In: *Proceedings of the 2012 IEEE 33rd Real-Time Systems Symposium (RTSS)*. San Juan / Puerto Rico, Dec. 2012, pp. 363–372. DOI: 10.1109/rtss.2012.86.

[Kle15] Jan C. Kleinsorge. "Tight Integration of Cache, Path and Task-interference Modeling for the Analysis of Hard Real time Systems". PhD thesis. Dortmund / Germany: TU Dortmund University, 2015.

[Kol+10] Steffen Kollmann, Victor Pollex, Kilian Kempf, and Frank Slomka. "A Scalable Approach for the Description of Dependencies in Hard Real-Time Systems". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6416. Berlin, Heidelberg / Germany: Springer Berlin Heidelberg, 2010, pp. 397–411. DOI: 10.1007/978-3-642-16561-0_37.

[Kop91] Hermann Kopetz. "Event-Triggered versus Time-Triggered Real-Time Systems". In: *Operating Systems of the 90s and Beyond*. Ed. by Arthur Karshmer and Jürgen Nehmer. Berlin, Heidelberg / Germany: Springer Berlin Heidelberg, 1991, pp. 86–101. DOI: 10.1007/bfb0024530.

[Kün+07] Simon Künzli, Arne Hamann, Rolf Ernst, and Lothar Thiele. "Combined Approach to System Level Performance Analysis of Embedded Systems". In: *Proceedings of the 5th IEEE/ACM/IFPI International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Salzburg / Austria, Sept. 2007, pp. 63–68. DOI: 10.1145/1289816.1289835.

[LCS19] Corinna G. Lee, Paul Chow, and Mark G. Stoodley. *UTDSP Benchmark Suite*. 2019. URL: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[Leh90] John Lehoczky. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines". In: *Proceedings of 11th Real-Time Systems Symposium (RTSS)*. Lake Buena Vista, FL / USA, Dec. 1990, pp. 201–209. DOI: 10.1109/real.1990.128748.

[LF14] Arno Luppold and Heiko Falk. "Schedulability-Oriented WCET-Optimization of Hard Real-Time Multitasking Systems". In: *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. Versailles / France, Oct. 2014, pp. 9–12.

[LF15a] Arno Luppold and Heiko Falk. "Code Optimization of Periodic Preemptive Hard Real-Time Multitasking Systems". In: *Proceedings of the 18th International Symposium on Real-Time Distributed Computing (ISORC)*. Auckland / New Zealand, Apr. 2015, pp. 35–42. DOI: 10.1109/isorc.2015.8.

[LF15b] Arno Luppold and Heiko Falk. "Schedulability aware WCET-Optimization of Periodic Preemptive Hard Real-Time Multitasking Systems". In: *Proceedings of the 18th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2015, pp. 101–104. DOI: 10.1145/2764967.2771930.

[LF17] Arno Luppold and Heiko Falk. "Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns". In: *Proceedings of Design, Automation and Test in Europe (DATE)*. Lausanne / Switzerland, Mar. 2017, pp. 1074–1079. DOI: 10.23919/date.2017.7927149.

[Li+07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. "Chronos: A timing analyzer for embedded software". In: *Science of Computer Programming* 69.1 (Oct. 2007), pp. 56–67. DOI: 10.1016/j.scico.2007.01.014.

[LKF16] Arno Luppold, Christina Kittsteiner, and Heiko Falk. "Cache-Aware Instruction SPM Allocation for Hard Real-Time Systems". In: *Proceedings of the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2016, pp. 77–85. DOI: 10.1145/2906363.2906369.

[LKM10] Paul Lokuciejewski, Timon Kelter, and Peter Marwedel. "Superblock-Based Source Code Optimizations for WCET Reduction". In: *2010 10th IEEE International Conference on Computer and Information Technology*. June 2010, pp. 1918–1925. DOI: 10.1109/cit.2010.327.

[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 (Jan. 1973), pp. 46–61. DOI: 10.1016/b978-155860702-6/50016-8.

[LM11] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Ed. by Nikil D. Dutt, Peter Marwedel, and Grant Martin. Embedded Systems. Dordrecht / Netherlands, Heidelberg / Germany, London / UK, New York / USA: Springer Science+Business Media B.V., 2011. DOI: 10.1007/978-90-481-9929-7.

[LM95] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration". In: *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC)*. San Francisco / USA, June 1995, pp. 456–461. DOI: 10.1145/216633.216666.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches". In: *Proceedings of 17th IEEE Real-Time Systems Symposium (RTSS)*. Washington, DC / USA, Dec. 1996, pp. 254–263. DOI: 10.1109/real.1996.563722.

[LOF18] Arno Luppold, Dominic Oehlert, and Heiko Falk. *Evaluating the Performance of Solvers for Integer-Linear Programming*. Technical Report. Hamburg / Germany: Institute of Embedded Systems, Hamburg University of Technology, Nov. 2018. DOI: 10.15480/882.1839.

[LOF20] Arno Luppold, Dominic Oehlert, and Heiko Falk. "Compiling for the Worst Case: Memory Allocation for Multi-Task and Multi-Core Hard Real-Time Systems". In: *ACM Transactions for Embedded Computing Systems* 19.2 (Mar. 2020), 14:1–14:26. DOI: 10.1145/3381752.

[Lok+09a] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models". In: *Proceedings of International Symposium on Code Generation and Optimization (CGO)*. Seattle, WA / USA, Mar. 2009, pp. 136–146. DOI: 10.1109/cgo.2009.17.

[Lok+09b] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. "Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining". In: *Proceedings of the 3rd workshop on Statistical and Machine Learning Approaches to Architecture and Compilation (SMART)*. 2009, pp. 1–15.

[LPC08] NXP B.V. *LPC2880; Preliminary data sheet*. Version Rev. 03. Apr. 2008.

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. Research Triangle Park, North Carolina / USA, Dec. 1997, pp. 330–335. DOI: 10.1109/micro.1997.645830.

[Lps19] lp_solve. *lpSolve*. http://lpsolve.sourceforge.net/5.5/. 2019. URL: http://lpsolve.sourceforge.net/5.5/.

[LS99] Thomas Lundqvist and Per Stenström. "Timing Anomalies in Dynamically Scheduled Microprocessors". In: *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*. Phoenix, AZ / USA, Dec. 1999, pp. 12–21. DOI: 10.1109/real.1999.818824.

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". In: *Proceedings of Real-Time Systems Symposium (RTSS)*. Santa Monica / USA, Dec. 1989, pp. 166–171. DOI: 10.1109/real.1989.63567.

[Lup+13] Arno Luppold, Benjamin Menhorn, Heiko Falk, and Frank Slomka. "A New Concept for System-Level Design of Runtime Reconfigurable Real-Time Systems". In: *ACM SIGBED Review* 10.4 (Dec. 2013), pp. 57–60. DOI: 10.1145/2583687.2583701.

[Man67] Glenn K. Manacher. "Production and Stabilization of Real-Time Task Schedules". In: *Journal of the ACM (JACM)* 14.3 (July 1967), pp. 439–465. DOI: 10.1145/321406.321408.

[Mar18] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Ed. by Nikil D. Dutt, Martin Grant, and Peter Marwedel. 3rd edition. Embedded Systems. Springer International Publishing AG, 2018. DOI: 10.1007/978-3-319-56045-8.

[MLF18] Kateryna Muts, Arno Luppold, and Heiko Falk. "Multi-Criteria Compiler-Based Optimization of Hard Real-Time Systems". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 54–57. DOI: 10.1145/3207719.3207730.

[MLF19] Kateryna Muts, Arno Luppold, and Heiko Falk. "Compiler-Based Code Compression for Hard Real-Time Systems". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2019, pp. 72–81. DOI: 10.1145/3323439.3323976.

[Muc14] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Haryana / India: Morgan Kaufmann Publishers, Inc., 2014. ISBN: 9788131214039.

[Nem+06] Fadia Nemer, Hugues Casse, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. "PapaBench: a Free Real-Time Benchmark". In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Frank Mueller. Vol. 4. Dagstuhl / Germany, 2006. DOI: 10.4230/OASIcs.WCET.2006.678.

[Neu+13] Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. "Sensitivity Analysis for Arbitrary Activation Patterns in Real-time Systems". In: *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Grenoble / France, Mar. 2013, pp. 135–140. DOI: 10.7873/date.2013.041.

[OF18] Dominic Oehlert and Heiko Falk. "WCET Analysis of Automative Buses using WCC". In: *Proceedings of the DATE Workshop on New Platforms of Future Cars*. Dresden / Germany, Mar. 2018.

[OLF16] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Practical Challenges of ILP-based SPM Allocation Optimizations". In: *Proceedings of the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2016, pp. 86–89. DOI: 10.1145/2906363.2906371.

[OLF17] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems". In: *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*. Dubrovnik / Croatia, June 2017, 1:1–1:22. DOI: 10.4230/LIPIcs.ECRTS.2017.1.

[OLF18a] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Compilation for Real-Time Systems - An Overview of the WCET-Aware C Compiler WCC". In: *Proceedings of the 9th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Barcelona / Spain, July 2018, pp. 1–3. DOI: 10.15480/882.2271.

[OLF18b] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Mitigating Data Cache Aging through Compiler-Driven Memory Allocation". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 58–61. DOI: 10.1145/3207719.3207731.

[OLF19] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Favorable Adjustment of Periods for Reduced Hyperperiods in Real-Time Systems". In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2019, pp. 82–85. DOI: 10.1145/3323439.3323975.

[OSF18] Dominic Oehlert, Selma Saidi, and Heiko Falk. "Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis". In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*. Barcelona / Spain, July 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.4.

[Pag+18] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. "Automated generation of time-predictable multi-core hardware executables on multi-core". In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS)*. Poitiers / France, Oct. 2018, pp. 104–113. DOI: 10.1145/3273905.3273907.

[Pla+11] Sascha Plazar, Jan Kleinsorge, Peter Marwedel, and Heiko Falk. "WCET-driven Branch Prediction aware Code Positioning". In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. Taipei / Taiwan, Oct. 2011, pp. 165–174. DOI: 10.1145/2038698.2038724.

[PLB12] Markos Papageorgiou, Marion Leibold, and Martin Buss. *Optimierung. Statische, dynamsiche, stochastische Verfahren für die Anwendung*. 3rd edition. Berlin, Heidelberg / Germany: Springer Vieweg, 2012. DOI: 10.1007/978-3-540-34013-3.

[PLM09] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. "WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems". In: *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). Dublin / Ireland, July 2009, pp. 1–11. DOI: 10.4230/OASIcs.WCET.2009.2286.

[Pol+09] Victor Pollex, Steffen Kollmann, Karsten Albers, and Frank Slomka. "Improved Worst-Case Response-Time Calculations by Upper-Bound Conditions". In: *Proceedings of Design, Automation and Test in Europe (DATE)*. Nice / France, Apr. 2009, pp. 105–110. DOI: 10.1109/date.2009.5090641.

[Pus03] Peter Puschner. "The Single-Path Approach Towards WCET-Analysable Software". In: *IEEE International Conference on Industrial Technology*. Maribor / Slovenia, Dec. 2003, pp. 699–704. DOI: 10.1109/icit.2003.1290740.

[PY19] Louis-Noel Pouchet and Tomofumi Yuki. *PolyBench/C - The Polyhedral Benchmark Suite*. 2019. URL: http://www.cs.ucla.edu/~pouchet/software/polybench/.

[QMM10] Muhammad Yasir Qadri, Dorian Matichard, and Klaus D. McDonald Maier. "JetBench: An Open Source Real-Time Multiprocessor Benchmark". In: *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*. Hannover / Germany, Feb. 2010, pp. 211–221. DOI: 10.1007/978-3-642-11950-7_19.

[Rei08] Jan Reineke. "Caches in WCET Analysis". PhD thesis. Saarbrücken / Germany: Universität des Saarlandes, 2008.

[Ric05] Kai Richter. "Compositional Scheduling Analysis Using Standard Event Models. The SymTA/S Approach". PhD thesis. Braunschweig / Germany: Technical University of Braunschweig, 2005.

[RLF14] Nicolas Roeser, Arno Luppold, and Heiko Falk. "Multi-Criteria Optimization of Hard Real-Time Systems". In: *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. Versailles / France, Oct. 2014, pp. 49–52.

[RLF18] Mikko Roth, Arno Luppold, and Heiko Falk. "Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. St. Goar / Germany, May 2018, pp. 86–89. DOI: 10.1145/3207719.3207729.

[Sam88] Arthur L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers. I". In: *Computer Games I*. Ed. by David N. L. Levy. New York, NY / USA: Springer New York, 1988, pp. 335–365. DOI: 10.1007/978-1-4613-8716-9_14.

[SGM17] Luca Santinelli, Fabrice Guet, and Jerome Morio. "Revising Measurement-Based Probabilistic Timing Analysis". In: *Proceedings of the 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Pittsburgh, PA / USA, Apr. 2017, pp. 199–208. DOI: 10.1109/rtas.2017.16.

[SH98] Mikael Sjödin and Hans Hansson. "Improved Response-Time Analysis Calculations". In: *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS)*. Madrid / Spain, Dec. 1998, pp. 399–408. DOI: 10.1109/real.1998.739773.

[SSE05] Jan Staschulat, Simon Schliecker, and Rolf Ernst. "Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay". In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*. Washington, DC / USA, July 2005, pp. 41–48. DOI: 10.1109/ecrts.2005.26.

[Sta16] William Stallings. *Computer Organization and Architecture. Designing for Performance*. Ed. by Tracy Johnson. 10th edition. Essex, England / UK: Pearson Education Limited, 2016. ISBN: 9781292096858.

[Str19] StreamIt Community. *The StreamIt Benchmark Suite*. 2019. URL: `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`.

[Suh+05] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. "WCET Centric Data Allocation to Scratchpad Memory". In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*. Miami / USA, Dec. 2005, pp. 223–232. DOI: `10.1109/rtss.2005.45`.

[Suh+06] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. "Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis". In: *Proceedings of the 43rd ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA / USA, July 2006, pp. 358–363. DOI: `10.1145/1146909.1147002`.

[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 (Mar. 1994), pp. 133–151. DOI: `10.1007/bf01088593`.

[Tur37] Alan M. Turing. "On Computable Numbers, With an Application to The Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: `10.2307/2268810`.

[Wan06] Ernesto Wandeler. "Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems". PhD thesis. Zurich / Switzerland: Swiss Federal Institute of Technology Zurich, 2006.

[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* 7.3 (Apr. 2008), 36:1–36:53. DOI: `10.1145/1347375.1347389`.

[Wil+10] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. "Static Timing Analysis for Hard Real-Time Systems". In: *Verification, Model Checking, and Abstract Interpretation: 11th International Conference, VMCAI 2010. Madrid, Spain, 1 2010. Proceedings*. Ed. by Gilles Barthe and Manuel Hermenegildo. Vol. 5944. Berlin, Heidelberg / Germany: Springer Berlin Heidelberg, 2010. Chap. Static Timing Analysis for

Hard Real-Time Systems, pp. 3–22. DOI: 10.1007/978-3-642-11319-2_3.

[WT19] Ernesto Wandeler and Lothar Thiele. *Real-Time Calculus (RTC) Toolbox*. 2019. URL: http://www.mpa.ethz.ch/Rtctoolbox.

[Xu10] Jia Xu. "A Method for Adjusting the Periods of Periodic Processes to Reduce the Least Common Multiple of the Period Lengths in Real-Time Embedded Systems". In: *Proceedings of 2010 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*. QingDao / China, July 2010, pp. 288–294. DOI: 10.1109/mesa.2010.5552058.

[ZBB11] Fengxiang Zhang, Alan Burns, and Sanjoy Baruah. "Sensitivity analysis of arbitrary deadline real-time systems with EDF scheduling". In: *Real-Time Systems* 47.3 (Apr. 2011), pp. 224–252. DOI: 10.1007/s11241-011-9124-y.

[Ziv+94] Vojin Zivojnovic, Juan Martinez Velarde, Christian Schläger, and Heinrich Meyr. "DSPstone: A DSP-Oriented Benchmarking Methodology". In: *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*. Dallas / USA, Oct. 1994.

Appendices

A. Evaluation of SPM Sizes

The following sections show the results for the static SPM allocation as discussed in Section 11.7 for different relative SPM sizes. Appendix A.1 shows the results for the ARM7TDMI architecture. Appendix A.2 continues with the results for TriCore TC1796.

Table A.1: Repair rates for the ARM7TDMI target for very low loads when performing an ILP-based SPM allocation for 20 % SPM size. u denotes the unoptimized system load.

Tasks	Initial Load u	ILP Sched. (DMS)	ILP NoSched (DMS)	ILP Sched. (EDF)	ILP NoSched (EDF)
2	0.8	100	100	100	100
2	1.0	95	90	95	90
4	0.8	100	100	100	100
4	1.0	100	100	100	100
6	0.8	100	100	100	100
6	1.0	100	100	100	100
8	0.8	100	100	100	95
8	1.0	90	95	95	95

Table A.2: Repair rates for the ARM7TDMI target for very low loads when performing an ILP-based SPM allocation for 40 % SPM size. u denotes the unoptimized system load.

Tasks	Initial Load u	ILP Sched. (DMS)	ILP NoSched (DMS)	ILP Sched. (EDF)	ILP NoSched (EDF)
2	0.8	100	100	100	100
2	1.0	100	95	100	95
4	0.8	100	100	100	100
4	1.0	100	100	100	100
6	0.8	100	100	100	100
6	1.0	100	100	100	100
8	0.8	100	100	100	100
8	1.0	90	100	95	100

A.1. ARM7TDMI

For the ARM7TDMI target, repair rates are very good when using the ILP-based optimization framework. As a result, additional loads of 2.6 and 3.0 have been evaluated for this scenario. In order to keep the diagrams in Chapter 12 clear, the repair rates for the low loads of 0.8 and 1.0 are not depicted. However, they were evaluated. The results are given in the tables below for each SPM size. The respective unoptimized system load is abbreviated by u.

Table A.3: Repair rates for the ARM7TDMI target for very low loads when performing an ILP-based SPM allocation for 60 % SPM size. $\mathfrak{u}$ denotes the unoptimized system load.

Tasks	Initial Load $\mathfrak{u}$	ILP Sched. (DMS)	ILP NoSched (DMS)	ILP Sched. (EDF)	ILP NoSched (EDF)
2	0.8	100	100	100	100
2	1.0	100	95	100	95
4	0.8	100	100	100	100
4	1.0	100	100	100	100
6	0.8	100	100	100	100
6	1.0	100	100	100	100
8	0.8	100	100	100	100
8	1.0	90	100	90	100

Table A.4: Repair rates for the ARM7TDMI target for very low loads when performing an ILP-based SPM allocation for 80 % SPM size. $\mathfrak{u}$ denotes the unoptimized system load.

Tasks	Initial Load $\mathfrak{u}$	ILP Sched. (DMS)	ILP NoSched (DMS)	ILP Sched. (EDF)	ILP NoSched (EDF)
2	0.8	100	100	100	100
2	1.0	100	100	100	100
4	0.8	100	100	100	100
4	1.0	100	100	100	100
6	0.8	100	100	100	100
6	1.0	95	100	100	100
8	0.8	100	100	100	100
8	1.0	90	100	95	100

Table A.5: Repair rates for the ARM7TDMI target for very low loads when performing an ILP-based SPM allocation for 100 % SPM size. $\mathfrak{u}$ denotes the unoptimized system load.

Tasks	Initial Load $\mathfrak{u}$	ILP Sched. (DMS)	ILP NoSched (DMS)	ILP Sched. (EDF)	ILP NoSched (EDF)
2	0.8	100	100	100	100
2	1.0	100	100	100	100
4	0.8	100	100	100	100
4	1.0	100	100	100	100
6	0.8	100	100	100	100
6	1.0	100	100	100	100
8	0.8	100	95	100	100
8	1.0	90	100	100	100

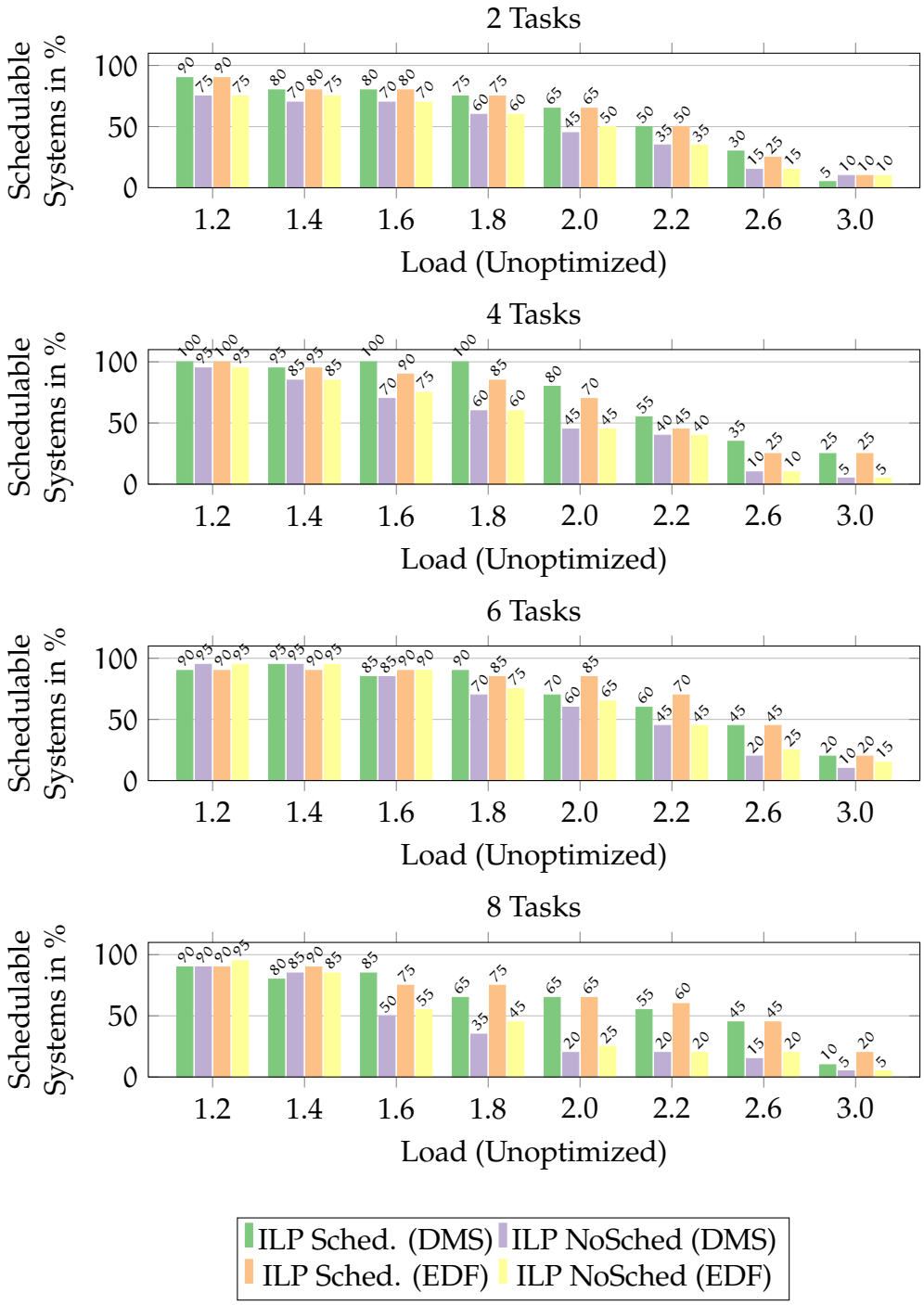


Figure A.1: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 20 % of the size of each task set.

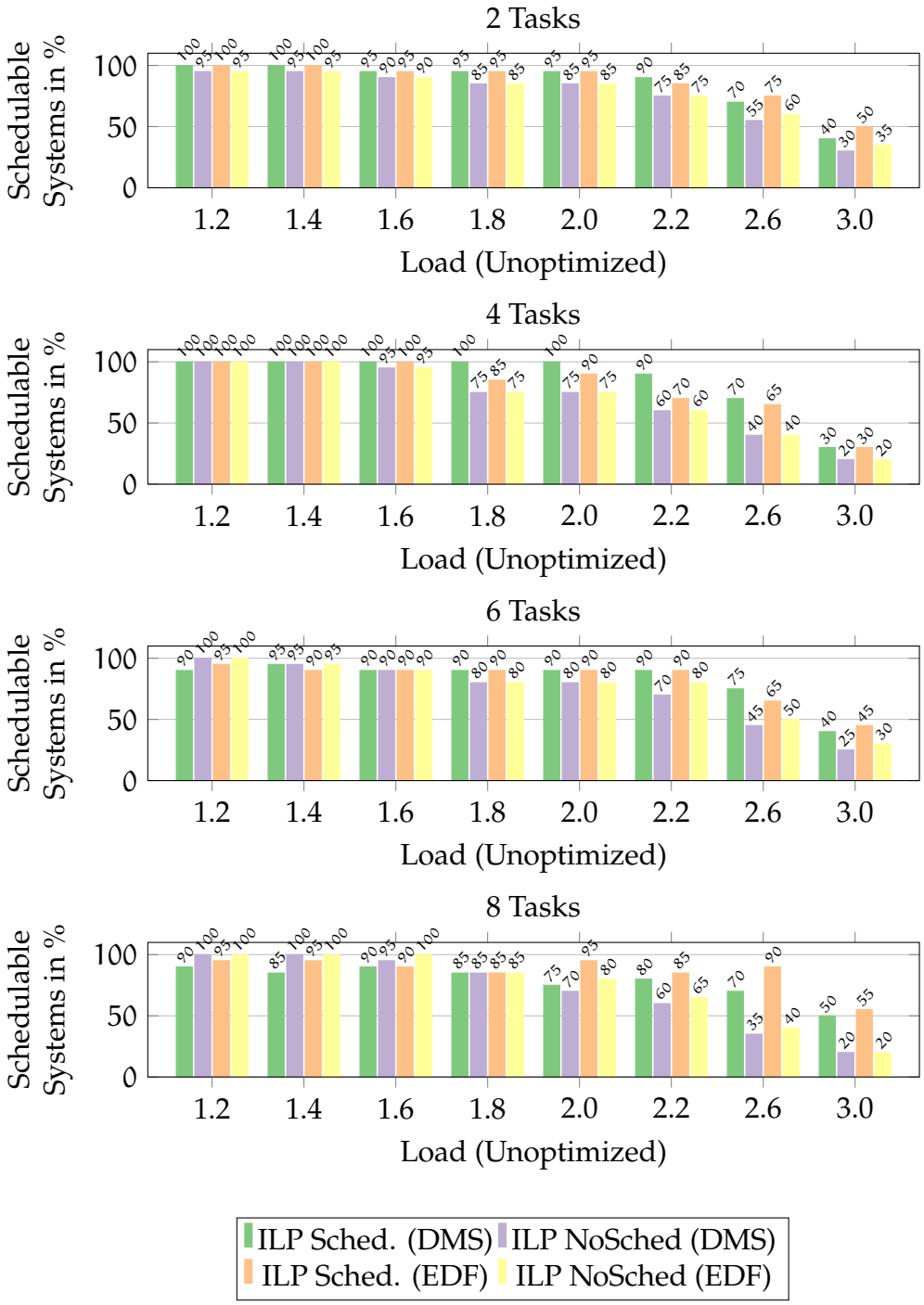


Figure A.2: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 40 % of the size of each task set.

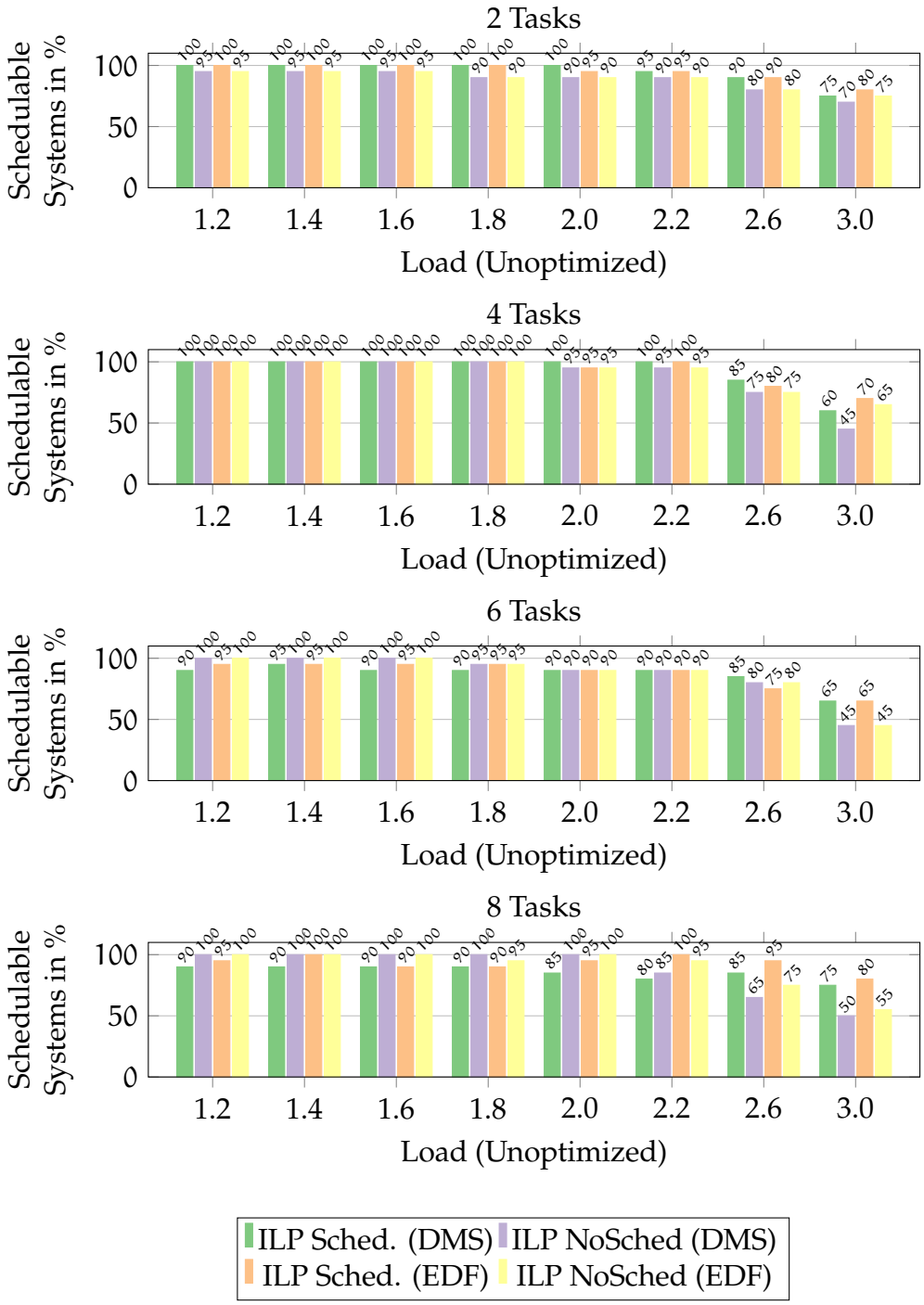


Figure A.3: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 60 % of the size of each task set.

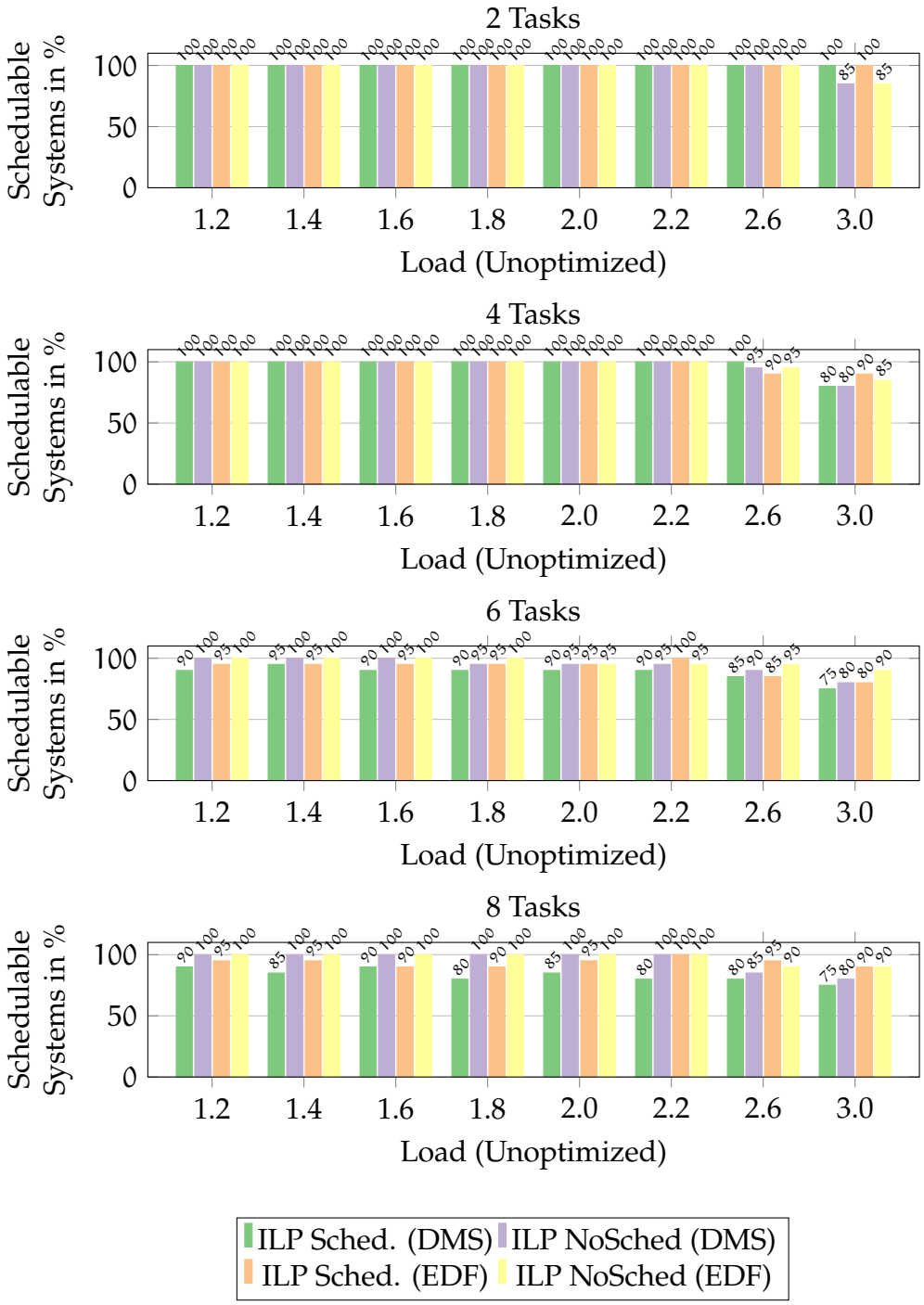


Figure A.4: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 80 % of the size of each task set.

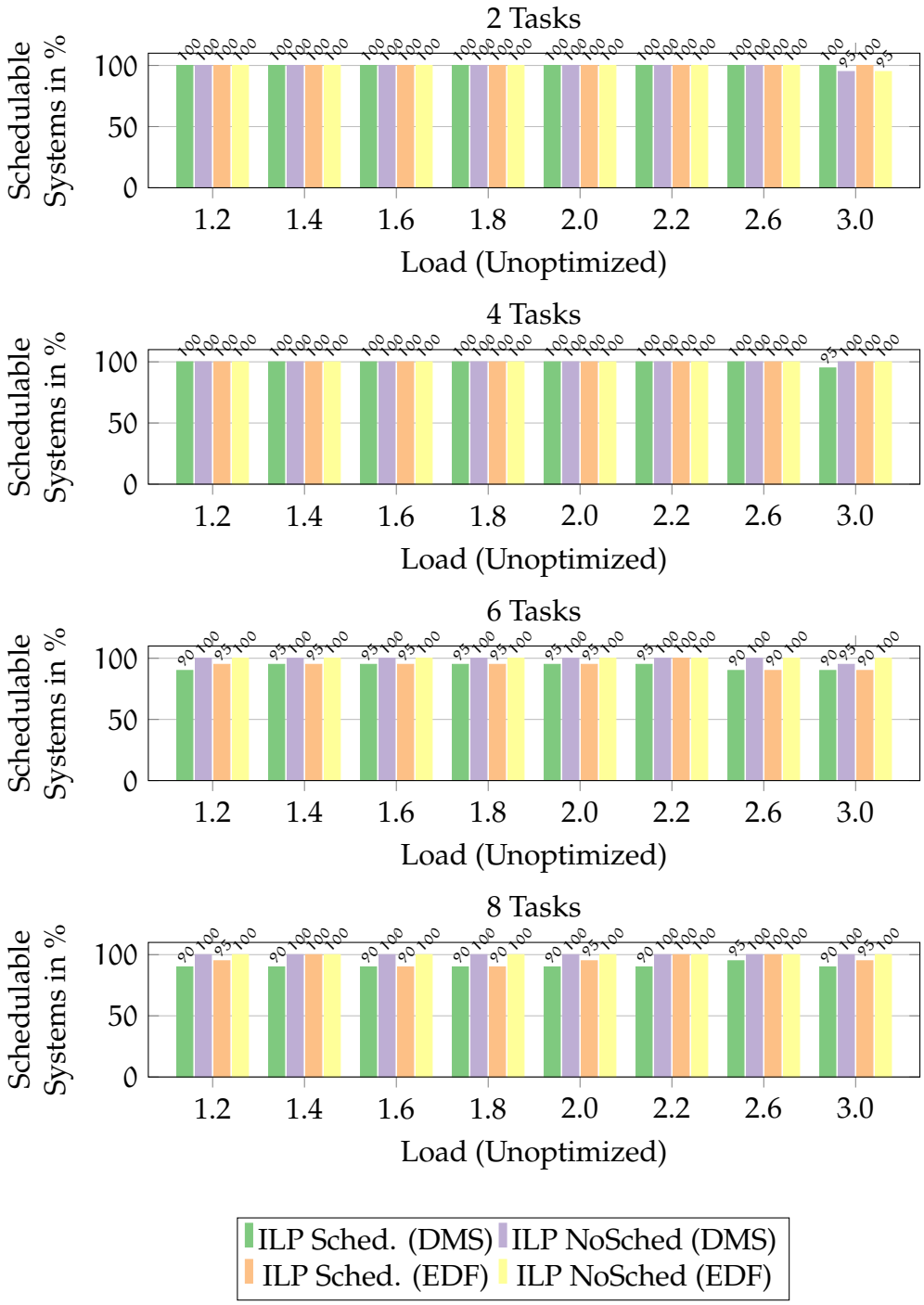


Figure A.5: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 100 % of the size of each task set.

A.2. TriCore TC1796

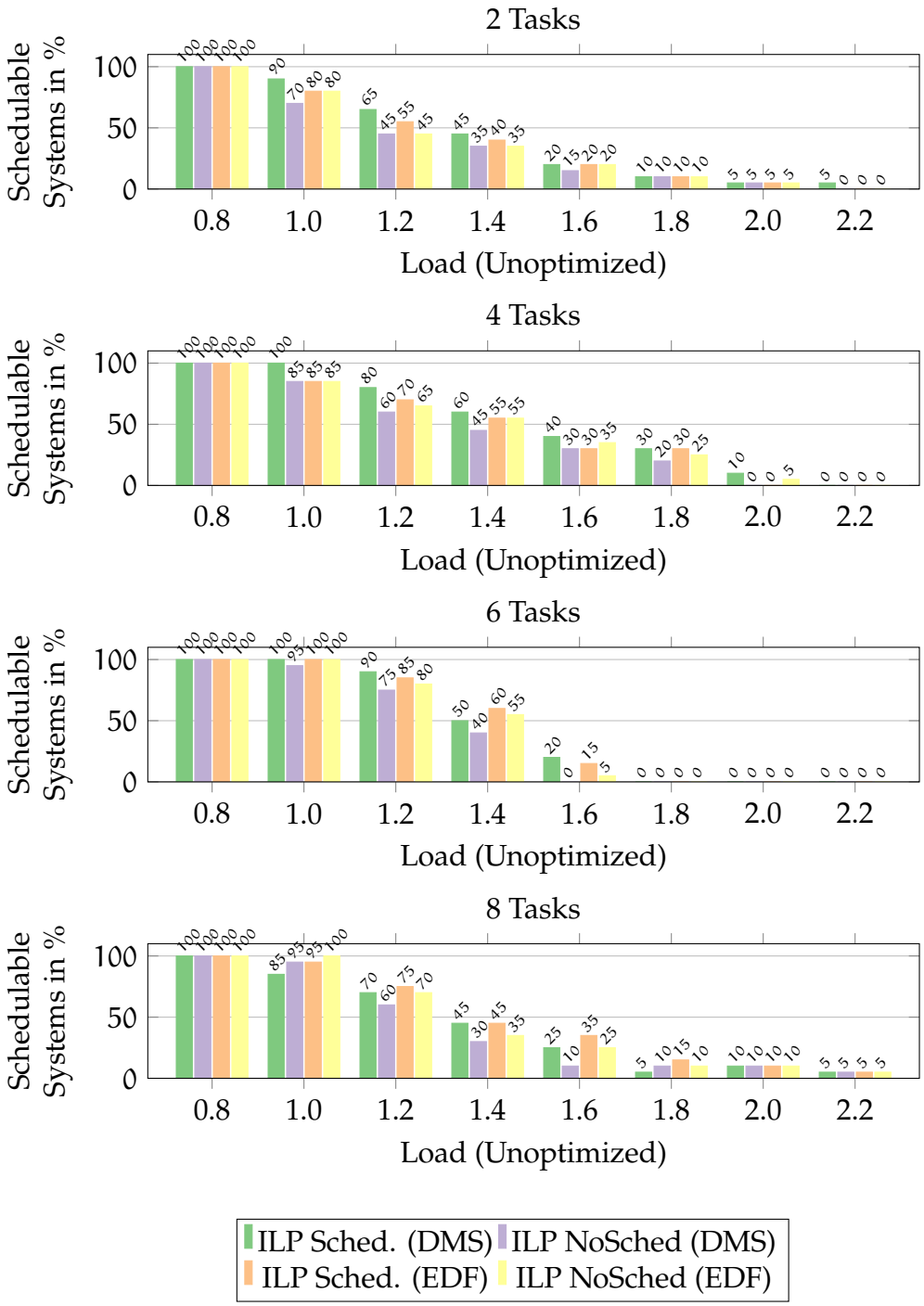


Figure A.6: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 20 % of the size of each task set.

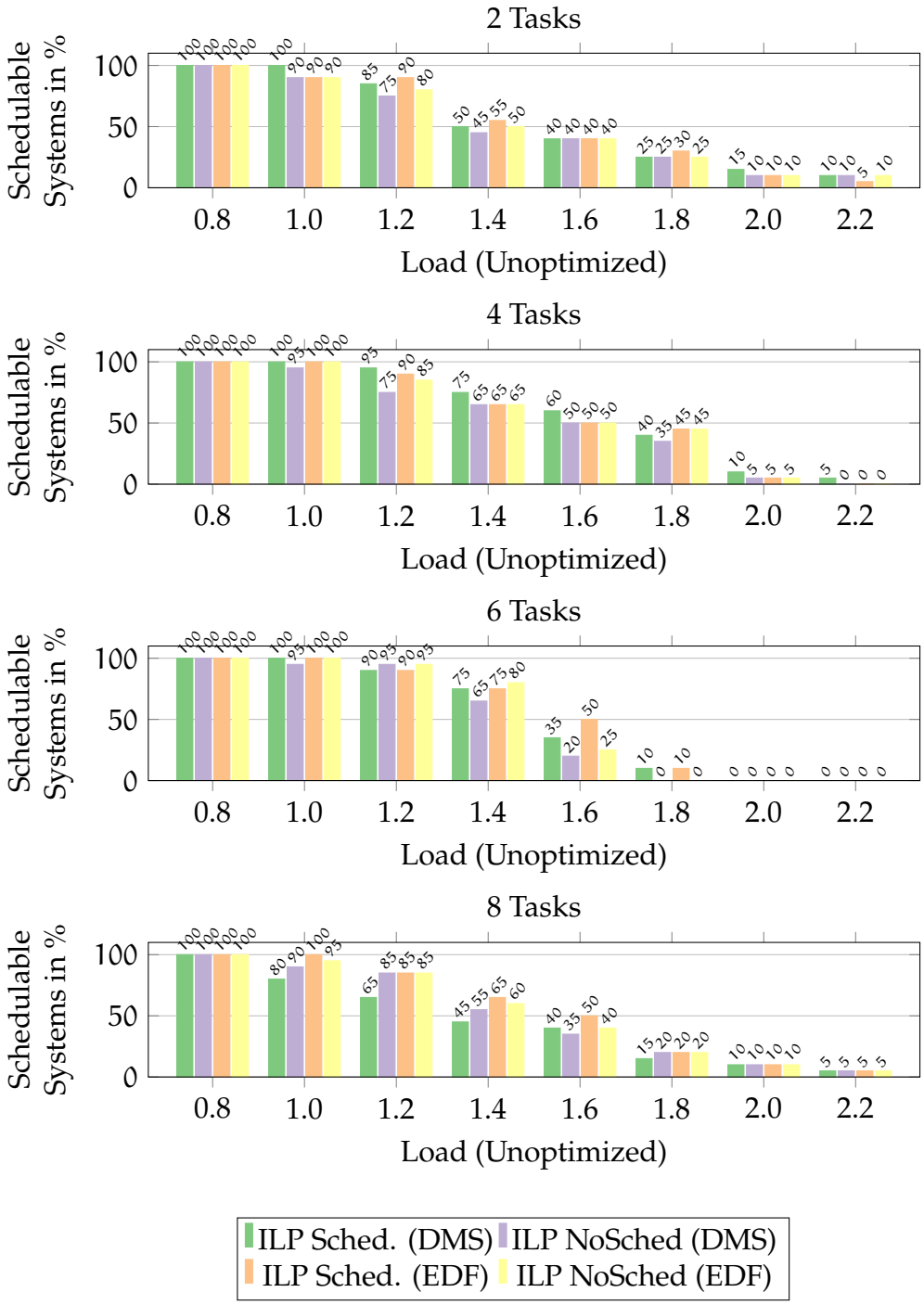


Figure A.7: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 40 % of the size of each task set.

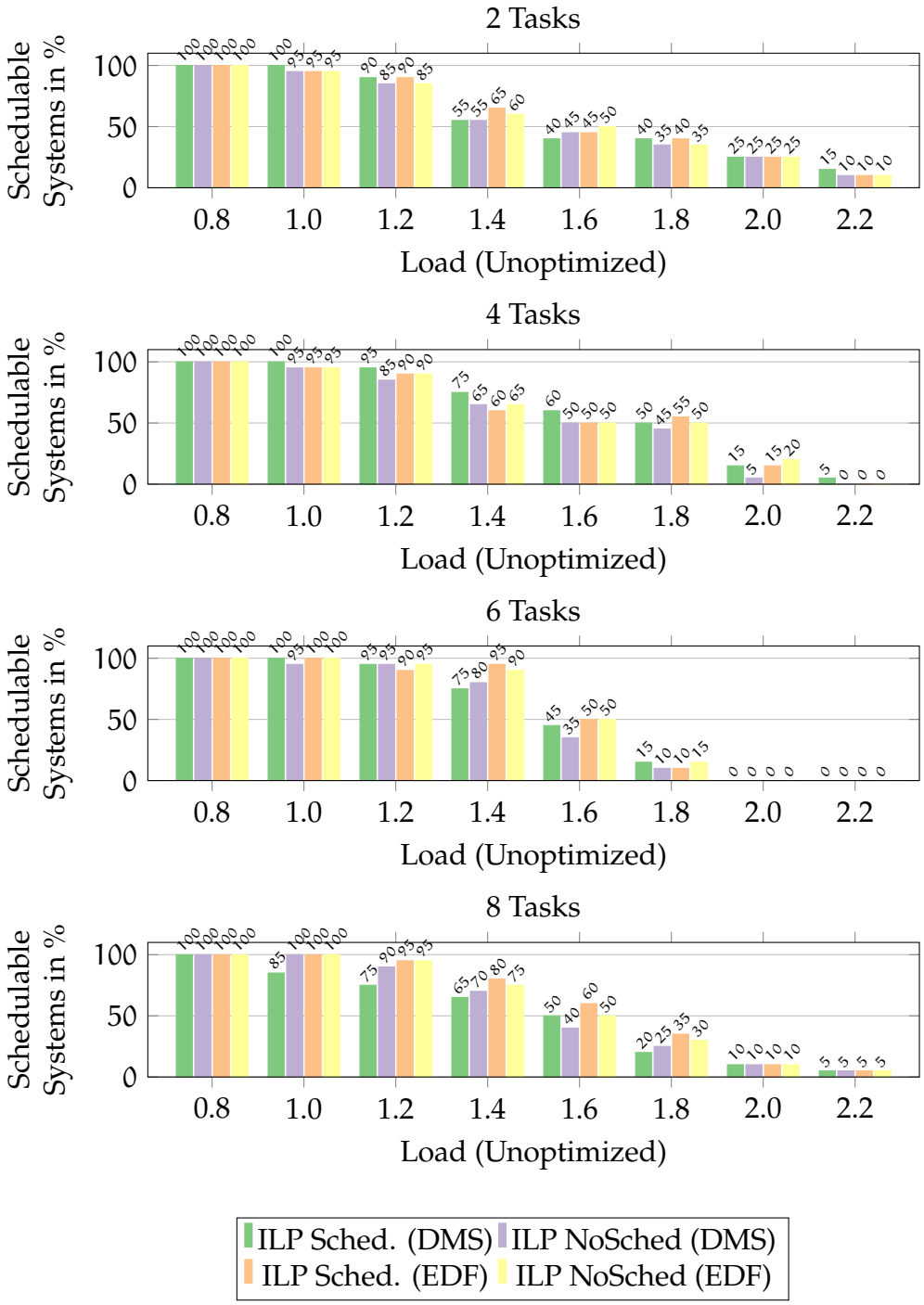


Figure A.8: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 60 % of the size of each task set.

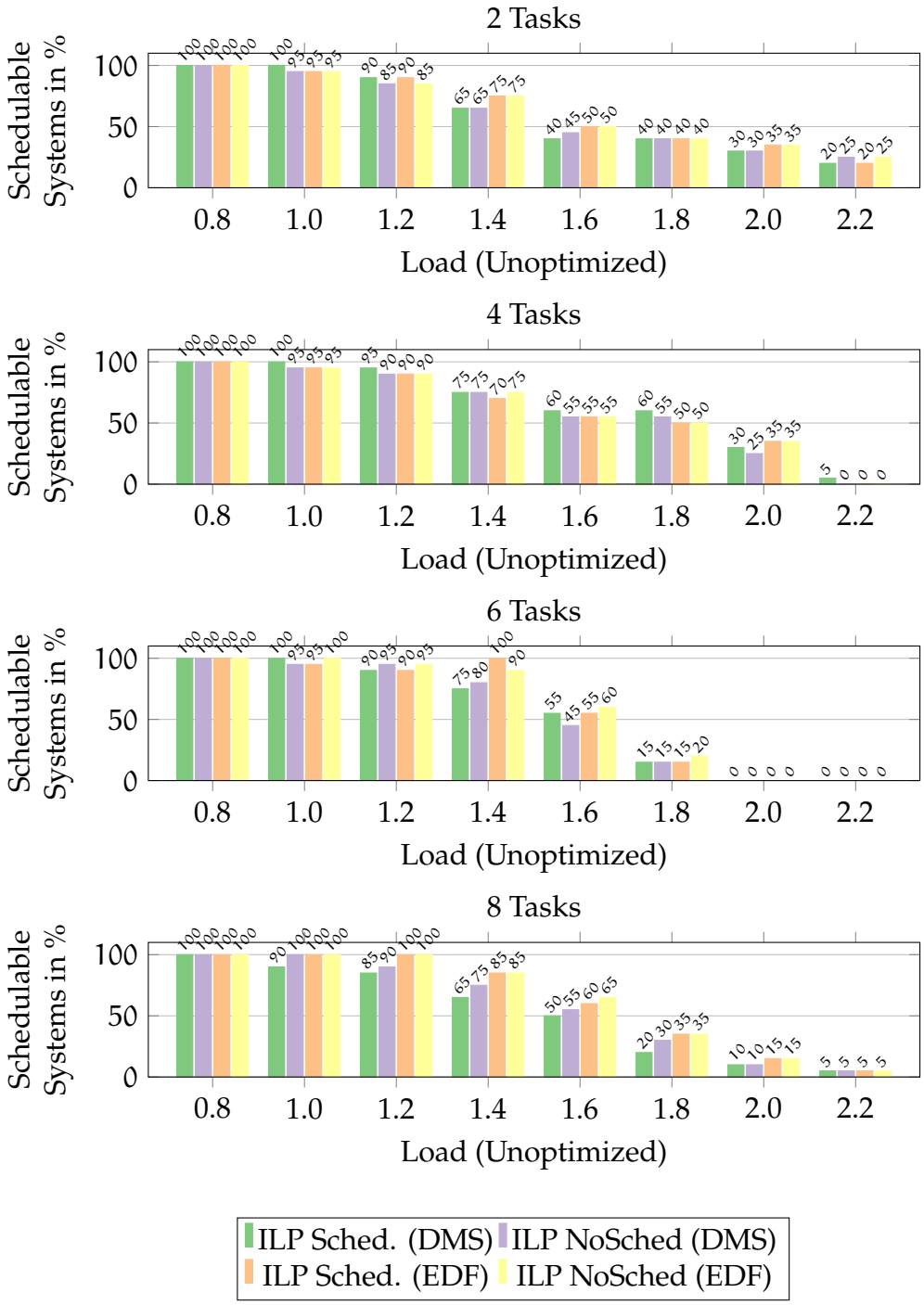


Figure A.9: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 80 % of the size of each task set.

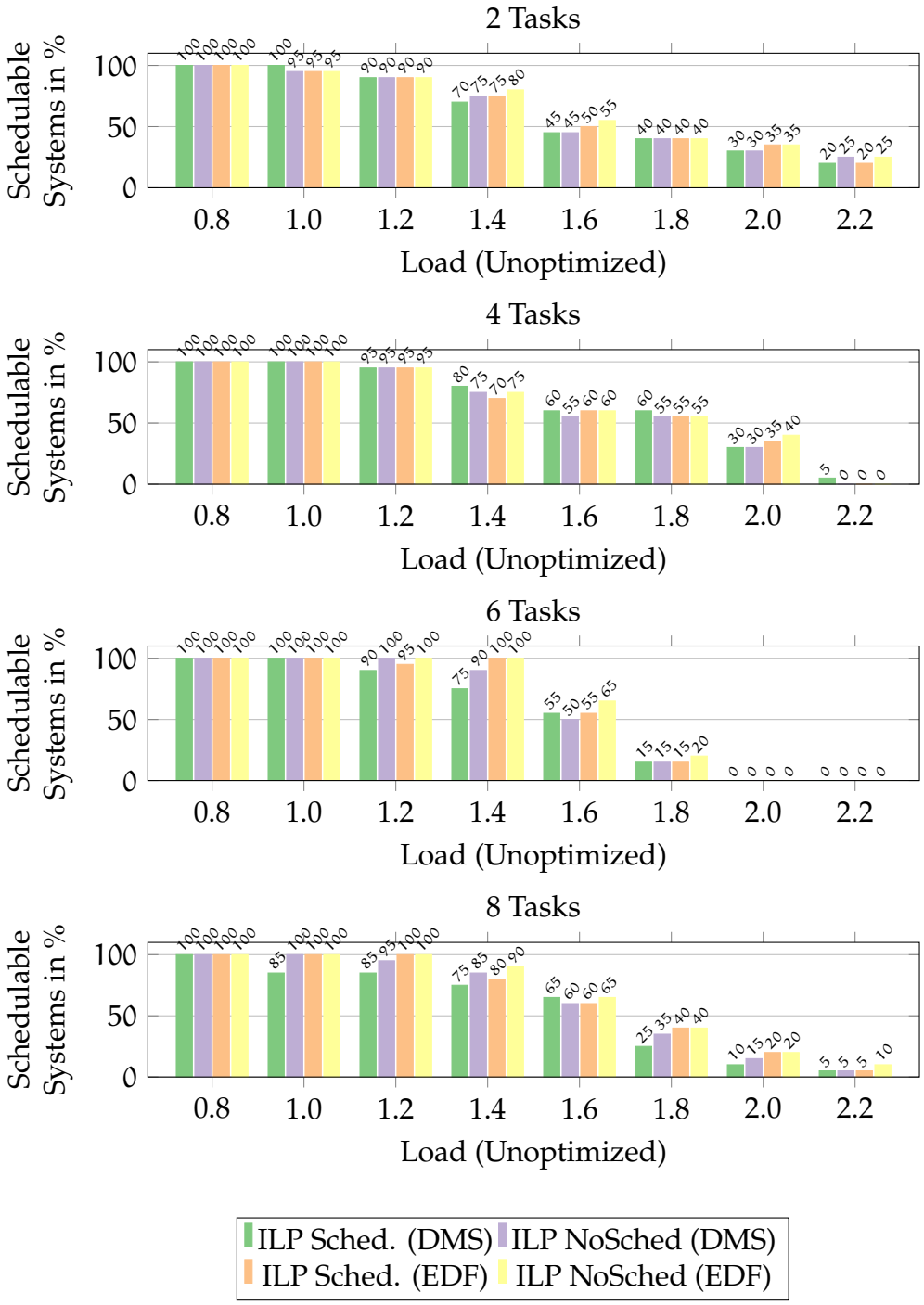


Figure A.10: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 100 % of the size of each task set.

B. Sensitivity Analysis for Evolutionary Algorithm

For the SPM allocation based on a genetic algorithm proposed in Chapter 10, multiple parameters have to be determined prior to performing the optimization. These are, e.g., the mutation probability or the heuristic which is used for repairing broken individuals in order to provide a good trade-off between repair speed and quality of each individual. Due to the high runtime of such an analysis, the optimization was run on the single-task benchmarks from the MRTC benchmark suite described in Section 11.1. All optimizations were performed using a relative SPM size of 40 %.

Each optimization was called with a timeout of 7 200 s and using `WCC`'s `-O2` optimization level (cf. Section 11.6.3). After the timeout, the genetic algorithm is terminated and the best result is taken as final result. Alternatively, the genetic algorithm terminates prior to the timeout if no improvement of the WCET could be achieved over one generation. Then, compilation is finished and a final WCET analysis is performed. Thus, the total compilation time may exceed the 7 200 s.

The following subsections of this appendix provide an overview of the results for each parameter which was tuned. Appendix B.1 evaluates the effect of different mutation probabilities. Appendix B.2 evaluates different settings for the repair heuristic which provides a fast way to repair a broken individual.

B.1. Mutation Probability

This section tackles the impact of different mutation probabilities on the resulting WCET. For both single-bit and multi-bit mutation (cf. Section 10.3), mutation probabilities of $0.1, 0.2, \ldots, 0.9$ were evaluated for the Infineon TriCore TC1796 architecture.

Fig. B.1 shows the WCET of each benchmark from the MRTC benchmark suite for multi-bit mutation. It can be seen that the results are very similar. I.e., from a quality point-of-view, there are no significant differences between the mutation probabilities. All mutation probabilities between 0.1 or at 0.9 will lead to approximately the same optimized WCET.

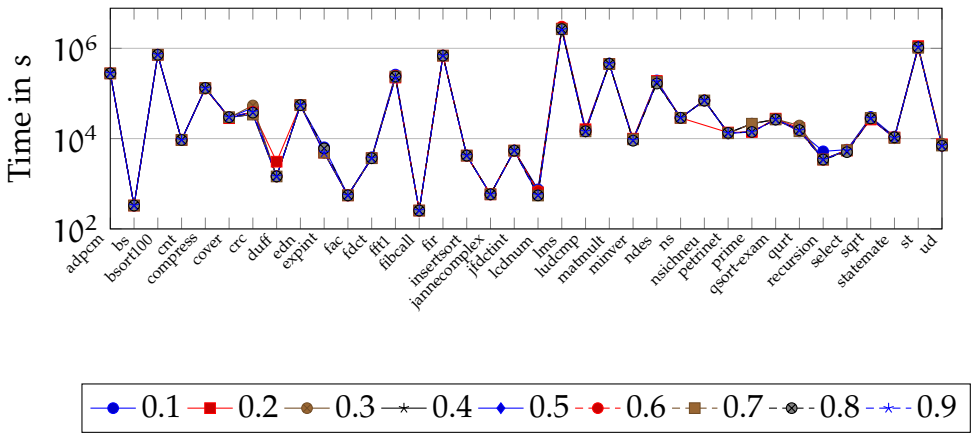


Figure B.1: WCETs of the MRTC benchmarks after applying the genetic SPM allocation for different mutation probabilities using multi-bit mutation.

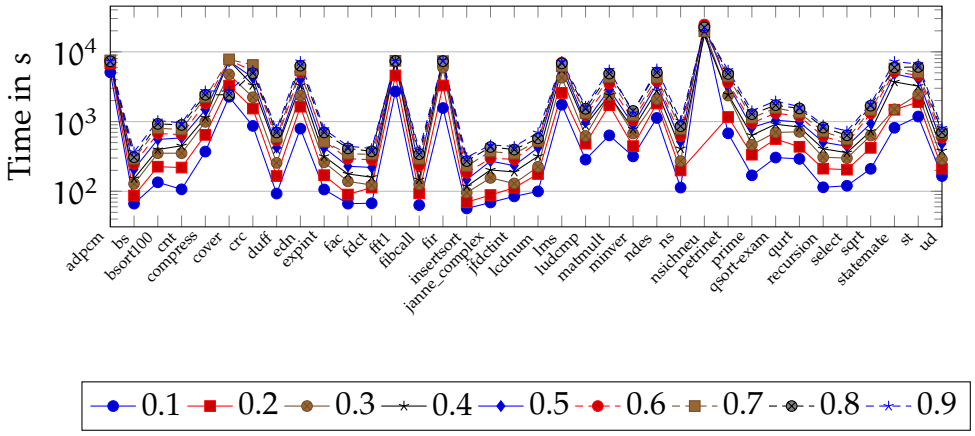


Figure B.2: Execution times of `WCC` when applying the genetic SPM allocation on the MRTC benchmarks using different mutation probabilities and multi-bit mutation.

Fig. B.2 shows the corresponding execution time of `WCC`. Here, significant differences can be observed. There is a clear trend that for a mutation probability of 0.1, the compilation and optimization process finishes the fastest, i.e., the genetic algorithm converges significantly faster than for high mutation probabilities.

For the `nsichneu` benchmark, some results are missing. The benchmark is large and features many conditional **`if`** statements. This leads to an extremely high runtime of `WCC`'s standard optimizations like, e.g., constant propagation and dead code elimination. As a result, sporadic timeouts can be observed prior to retrieving any result.

Fig. B.3 shows the WCET of each of the MRTC benchmarks for one-bit mutation. It can be observed that, as for multi-bit mutation, there is no notable difference

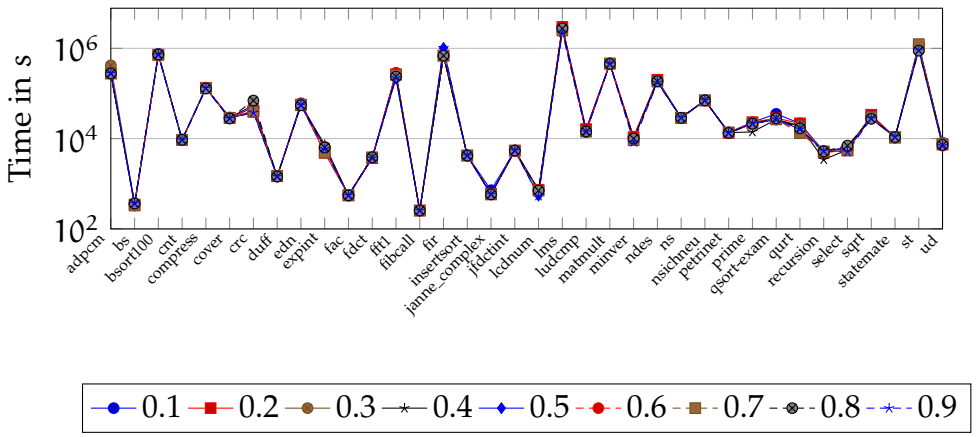


Figure B.3: WCETs of the MRTC benchmarks after applying the genetic SPM allocation for different mutation probabilities using one-bit mutation.

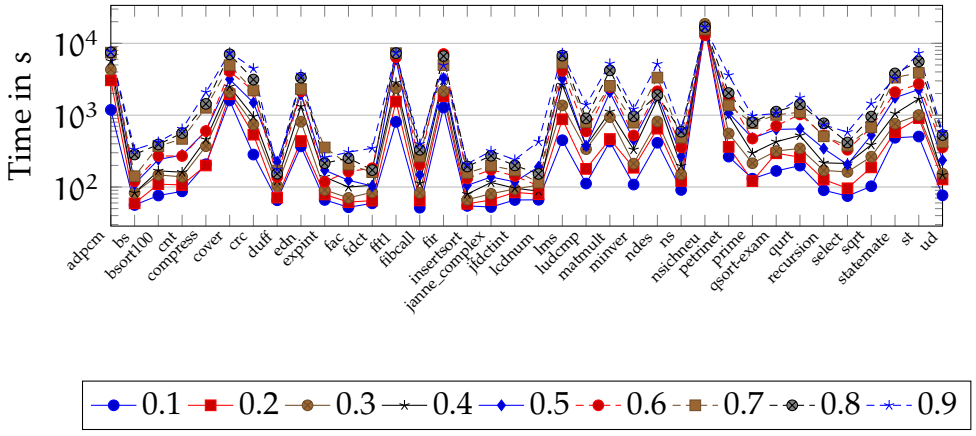


Figure B.4: Execution times of `WCC` when applying the genetic SPM allocation on the MRTC benchmarks using different mutation probabilities and one-bit mutation.

in the resulting WCETs for different mutation probabilities. Also, the resulting WCETs for multi-bit and one-bit mutation do not differ significantly. Fig. B.4 shows the corresponding overall execution times of `WCC`. Again, as already observed for multi-bit mutation, the lowest mutation probability leads to the fastest convergence of the results. However, the results for 0.1 and 0.2 are closer, whereas for multi-bit mutation, a mutation probability of 0.1 was always clearly outperforming 0.2. When comparing the timing results from multi-bit and single-bit mutation, differences are also not that big. Despite this, one-bit mutation clearly converges faster for some benchmarks.

As a result of this evaluation for the genetic algorithm, mutation processes are performed using one-bit mutation with a mutation probability of 0.1.

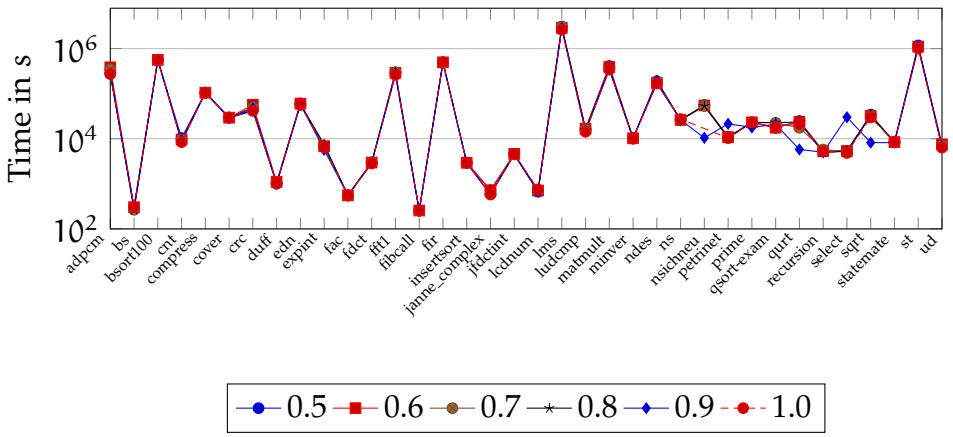


Figure B.5: WCETs of the MRTC benchmarks when applying different repair ratios when fixing an overfull SPM.

B.2. Repair Heuristic

In case of an overfull SPM, the repair function (cf. Section 10.4) randomly selects basic blocks and moves them back from SPM to the slower main memory. Afterwards, the jump correction routine has to be applied in order to restore a correct control flow. These corrections may introduce additional instructions to the SPM which in turn may lead to a once again overfull SPM. For small single-tasking benchmarks, iteratively repeating this basic block removal until the SPM is no longer overfull is usually quite fast. However, for large multi-tasking benchmarks, this may easily lead to high computational overhead, thus slowing down the genetic algorithm's performance.

To counter this, the repair function can remove more basic blocks as needed. I.e., a number of basic blocks is randomly removed from an overfull SPM such that the SPM is only filled up to a given percentage. After that, the jump correction is run only once afterwards. This reduces the number of jump correction runs drastically. However, if too many basic blocks are removed from SPM, the fast memory is no longer used as efficiently as possible, thus possibly degrading optimization results. The repair function was discussed in detail in Section 10.4.

This section illustrates the impact of different percentages on the resulting WCET and `WCC`'s execution time. For the mutation, one-bit mutation with a mutation probability of 0.1 was used as evaluated in the previous sections.

Fig. B.5 shows the resulting WCETs. As in the previous section, some results are missing for `nsichneu` due to timeouts. Apart from this, all results are very close with regard to the optimized WCET. For few benchmarks, a repair ratio of 0.9 sometimes performs slightly better, but sometimes also performs slightly

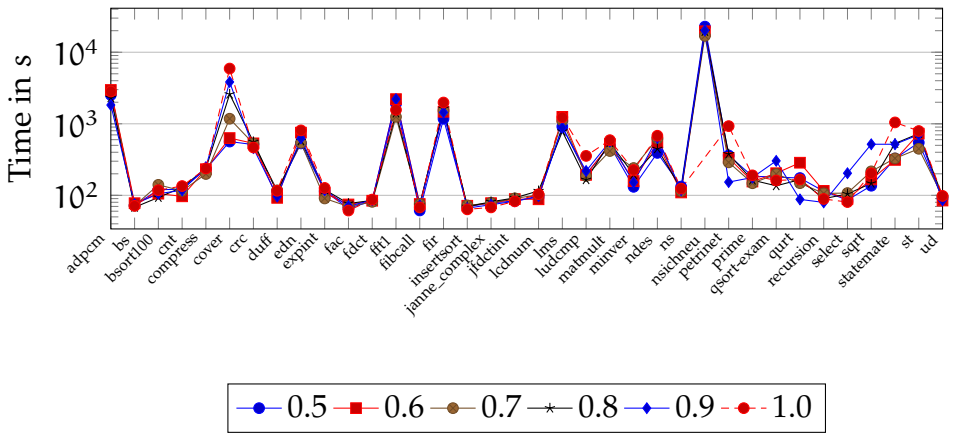


Figure B.6: Execution times of WCC when applying different repair ratios in order to fix an overfull SPM.

worse than the other repair ratios. Therefore, from a purely optimization *quality* point of view, basically any repair rate may be chosen.

Fig. B.6 shows the corresponding overall runtimes of the WCC compiler. It can be seen that even for these relatively small single-tasking benchmarks from the MRTC benchmark suite, high repair ratios lead to small but visibly higher runtimes. Therefore, it can clearly be seen that removing more basic blocks than necessary within one execution of the repair function can lead to smaller compilation runs without degrading the optimization's quality with regard to the final WCET. As a result, a repair ratio of 0.8 was chosen for the evaluations in Chapter 12. This provides a compromise between less jump correction calls and an efficient SPM usage.

C. Evaluation Results for Task Sets from All Available Benchmark Suites

Chapter 12 evaluated task sets which were assembled from tasks of the MRTC benchmark suite. This chapter complements this evaluation by showing the results when assembling the task sets using benchmarks from multiple benchmark suites (cf. Section 11.1). Apart from extending the range of benchmarks, the system setup is left identical to Section 12.1. I.e., a relative SPM size of 40 % is used.

Appendix C.1 shows the results for the ILP-based optimization framework for ARM7TDMI. Appendix C.2 shows the results for the TriCore TC1796. The genetic algorithm was not evaluated with these extended task sets. The reason for this is that the task sets from these extended benchmark suites are significantly larger than the sole MRTC benchmarks. Due to the poor results for even the small MRTC tasks (cf. Sections 12.1.3 and 12.1.4), the genetic algorithm is not able to repair a significant amount of task sets for this more complex setup within reasonable amount of time.

C.1. ILP Optimization on ARM7TDMI

This section shows the results of the ILP-based optimization approach for the ARM7TDMI architecture. Apart from the different benchmarks, the evaluation is equivalent to the one in Section 12.1.1.

Fig. C.1 shows the repair rates for the task sets. Identical to Section 12.1.1, the X-axis of each graph denotes the original, unoptimized system load. The Y-axis gives the percentage of task sets which are schedulable after applying the respective compiler optimizations. The bars depict the arithmetic mean over the repair rates for DMS and EDF for each evaluated optimization and system load.

The overall trend of the evaluation is the same as discussed in Section 12.1.1 for the MRTC benchmarks. This section will therefore focus on a brief discussion of the differences.

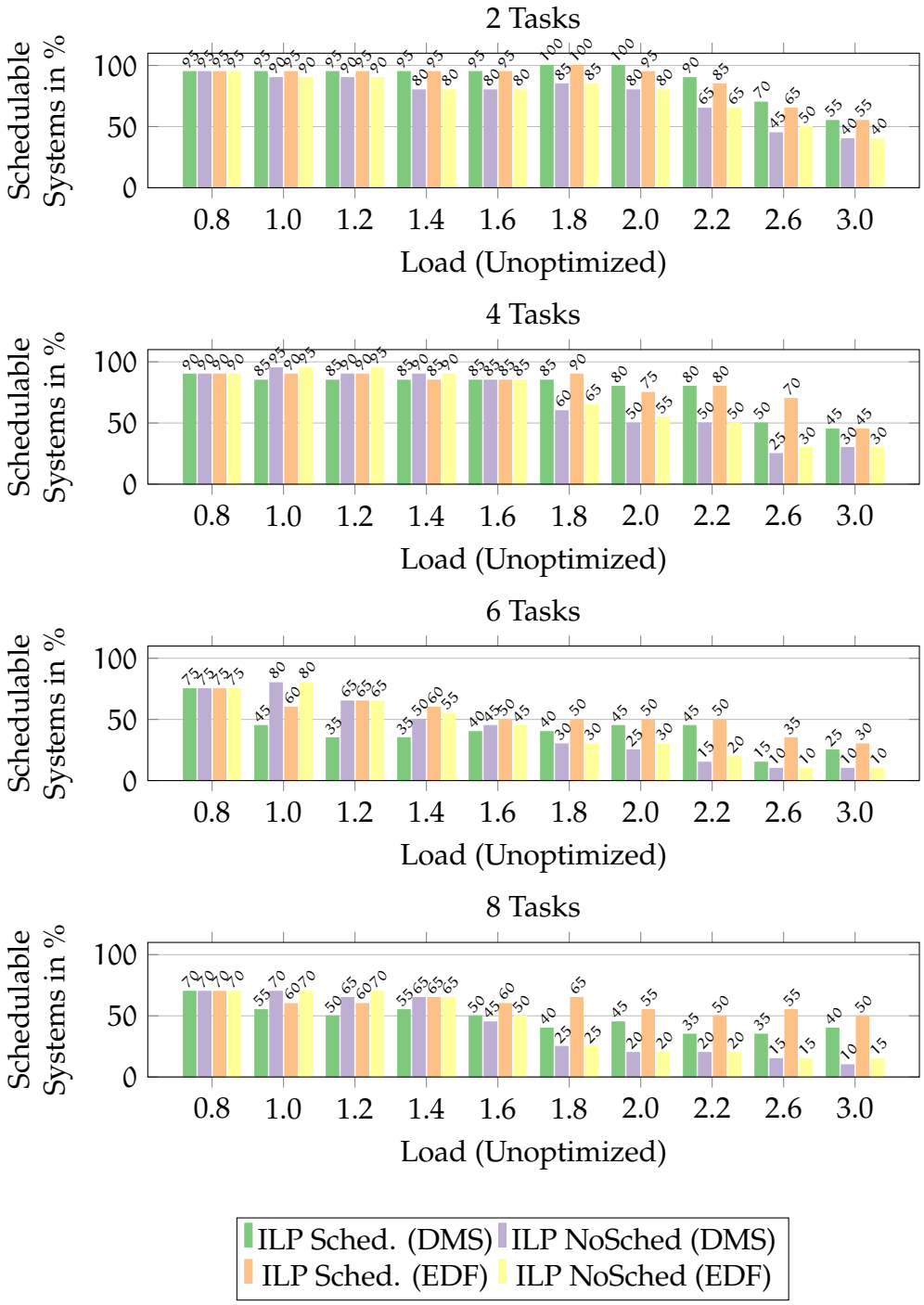
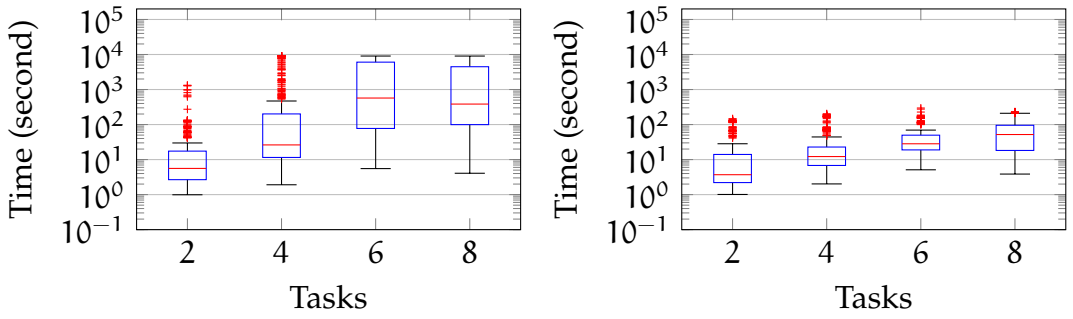


Figure C.1: ILP-based results for the SPM optimization on the ARM7TDMI architecture. The SPM size is 40 % of the size of each task set.



(a) Schedulability-aware

(b) Schedulability-unaware

Figure C.2: Compilation times for ILP-based SPM allocation for the ARM7TDMI architecture, including all WCET analyses.

There are two main differences: First, the overall repair rates are significantly lower than for the MRTC benchmarks. Second, for initial loads up to 1.4, the ILP optimization *without* schedulability constraints outperforms the schedulability-*aware* ILP for all but the smallest task set size. The reason for both issues stems from the huge size of several benchmarks. E.g., the `dijkstra` benchmark from the MiBench benchmark suite has an initial WCET of more than 3.4×10^9 CPU cycles. The ARM7TDMI microcontroller used in this evaluation features a maximum frequency of 60 MHz (cf. Section 11.4). This means, that the unoptimized WCET of *one* task is well beyond 50 s. Other benchmarks like, e.g., the `fft` benchmarks from DSPstone or the Jetbench benchmarks feature similar WCETs beyond 1.0×10^9 CPU cycles. For multiple of these benchmarks, the ILP optimization *with* schedulability constraints is not able to finish within the given two hour time limit. As a result, at loads up to 1.6, the ILP optimization without schedulability constraints may provide better repair rates due to its significantly lower complexity.

For the large task sets of 6 and 8 tasks, a significant amount of systems is already not schedulable at a load of 0.8. This mainly stems from the fact that the hyper-period of the resulting task set gets so large that it cannot be safely reduced using the coefficients presented in Section 11.3. In some cases, the hyper-period exceeded 1.0×10^{40} CPU cycles.

Despite these effects, the overall impact of the schedulability-aware optimization framework persists. At higher loads, the repair rates of the schedulability-aware ILP significantly outperform the traditional ILP optimization without schedulability awareness. E.g., for the largest task sets and highest initial load of 3.0, 40 % of all task sets can be repaired using DMS and even 50 % are repairable under EDF scheduling. In contrast to that, the ILP without schedulability-constraints is only able to repair 10 % under DMS and 15 % with EDF scheduling.

Fig. C.2 shows the corresponding compilation times. Compared to the MRTC benchmarks, the compilation times increase for both optimizations. As already mentioned above, for the schedulability-aware optimizations, significantly more timeouts occur for the larger task sets of 6 and 8 tasks.

C.2. ILP Optimization on TriCore

This section shows the results of the ILP-based schedulability-aware SPM allocation for the Infineon TriCore TC1796 architecture.

Fig. C.3 shows the repair rates. At a first glance it can be seen that even at low loads of 0.8, a significant amount of task sets cannot be repaired using any approach. This stems from the fact, that more than 11 % of all task sets could not be compiled at all due to an overfull `.data` section. For the largest task sets of 8 tasks, this applied to even 25 % of the task sets. Several benchmarks feature huge arrays with data which are encoded or decoded by the benchmark. When these benchmarks are combined into 8-task systems, the TriCore's memories are not large enough, thus the assembled task set cannot be compiled at all. Decreasing the size of the data arrays, or excluding any data-intensive benchmarks would bias the evaluation. Manually creating feasible task sets by cherry-picking benchmarks such that the `.data` section is no longer overfull would also lead to an evaluation which is no longer unbiased. Therefore, the results are given here unchanged. However, it should be stressed, that especially for the larger task sets, the high number of non-compilable task sets significantly decreases the significance of the results.

Apart from the generally worse repair rate, the results are comparable to the ones in Section 12.1.2 for the MRTC benchmarks. Due to the complex TriCore architecture, the gap between the schedulability-aware and schedulability-unaware optimizations is smaller than for the ARM7TDMI architecture. As previously discussed in Section 12.1.2, over-approximations of the WCET can lead to the case that a system appears to be non-repairable by the constraint-based schedulability-aware ILP framework. However, the schedulabiltiy-*un*aware optimization framework may succeed if – in fact – the system is repairable. This can be especially seen for the task sets consisting of 4 tasks at a load of 1.0. Here, the system is barely unschedulable, but due to over-approximations of the WCET, for DMS only 70 % of the task sets can be repaired using the schedulability-aware optimization framework. The ILP without schedulability-awareness succeeds at repairing these tasks as – indeed – relatively small changes to each task's WCET already lead to a schedulable system. However, as soon as the load increases, the schedulability-*un*aware optimization fails at identifying those tasks that are critical for the system's schedulability. As a result, at a load of 1.2, only 45 % of all tasks

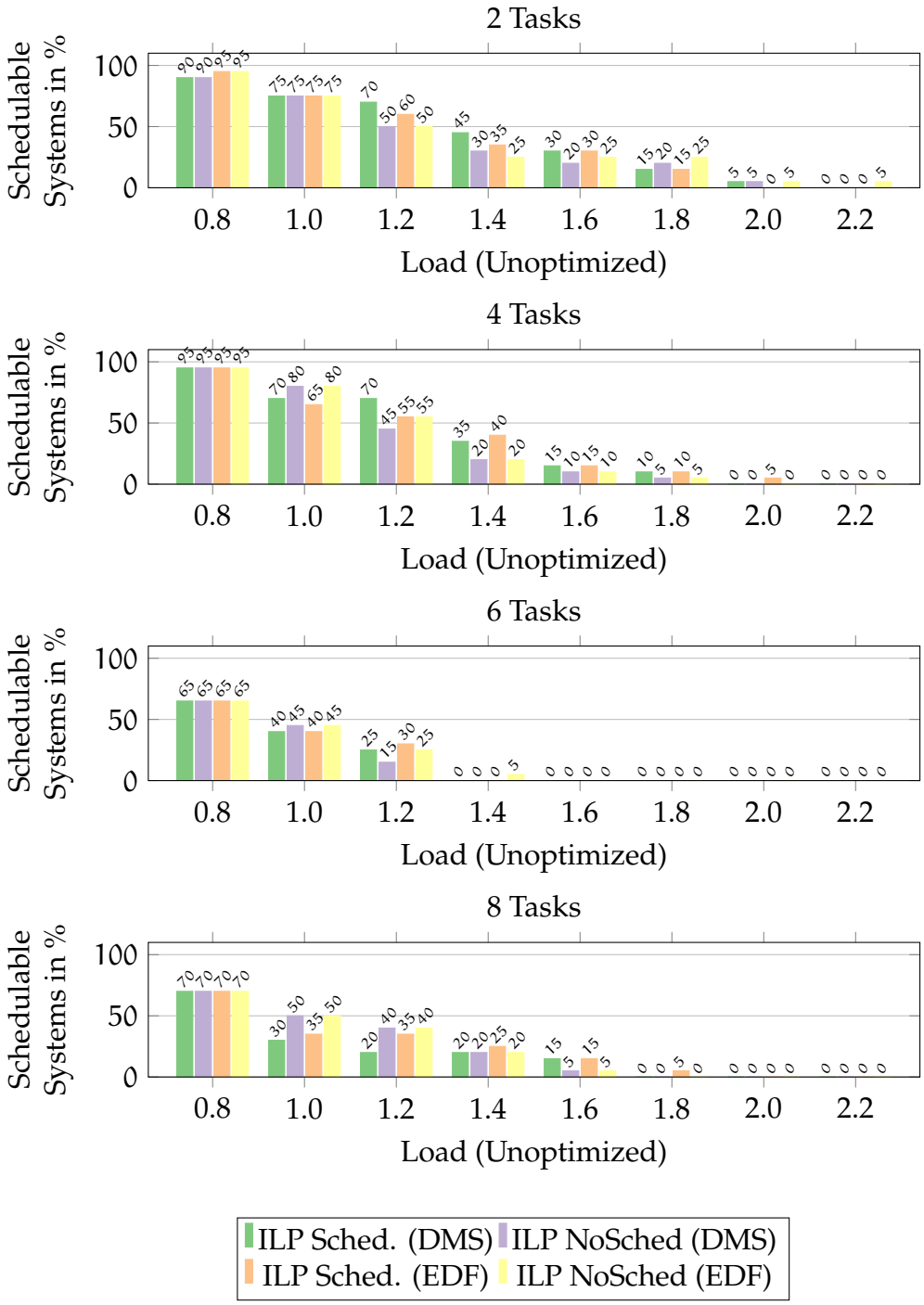
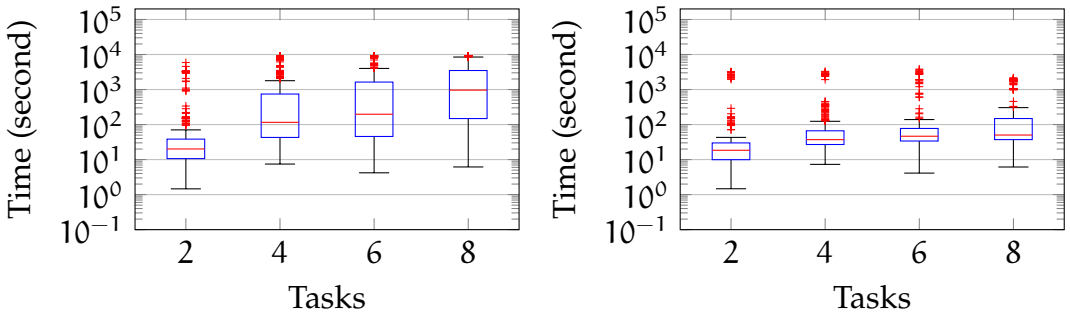


Figure C.3: ILP-based results for the SPM optimization on the TC1796 architecture. The SPM size is 40 % of the size of each task set.



(a) Schedulability-aware

(b) Schedulability-unaware

Figure C.4: Compilation times for ILP-based SPM allocation for the TC1796 architecture, including all WCET analyses.

are repairable *without* the schedulabiltiy-aware constraints. On the other hand, the schedulability-aware optimization framework is able to repair 70 % of the task sets. This trend continues for larger loads and task sets. The same effect can also be seen for EDF scheduling. For the task set size of 2 tasks, the original load of 1.8 forms an outlier to this trend, as more systems are repairable here using the schedulability-*un*aware optimizations due to over-approximations of the WCET within the ILP.

Fig. C.4 shows the respective runtimes of the `WCC` compiler. The runtimes are very similar to the results for the MRTC benchmarks which were previously discussed in Section 12.1.2 with the schedulability-aware ILP optimization framework needing about one magnitude more compilation time than the ILP without schedulability-awareness.