Semi-Automatic Optimization of Hardware Architectures in Embedded Systems

Dipl.-Inform. Eike Martin Thaden

CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Semi-Automatic Optimization of Hardware Architectures in Embedded Systems

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Dipl.-Inform. Eike Martin Thaden

Gutachter:

Prof. Dr. Werner Damm
Prof. Dr.-Ing. Wolfgang Nebel

Datum der Verteidigung: 30. Mai 2013

Office address:

Eike Martin Thaden
OFFIS
Escherweg 2
D-26121 Oldenburg
Germany

E-Mail:	eike.thaden@offis.de
	eike.thaden@informatik.uni-oldenburg.de

Home address:

Eike Martin Thaden
Masurenstraße 41b
D-26127 Oldenburg
Germany

E-Mail:	mail@eike-thaden.de

Zusammenfassung

Der Aufwand für die Entwicklung eines sicherheitskritischen eingebetteten Systems kann erheblich reduziert werden, wenn als Ausgangspunkt ein ähnliches System gewählt wird, das lediglich um zusätzliche Funktionalität ergänzt wird. Dies ist jedoch eine äußerst anspruchsvolle Aufgabe, da sowohl für die bereits vorhandenen als auch die neuen Systembestandteile in der Regel komplexe Randbedingungen erfüllt sein müssen, um deren korrekte Funktionsweise zu gewährleisten. Erschwerend kommt hinzu, dass größere eingebettete Systeme überwiegend als verteilte Systeme mit mehreren Prozessoren und einer komplexen Kommunikationsinfrastruktur realisiert sind. Bei der Erweiterung solcher Systeme ergeben sich hierdurch eine Vielzahl von Entwurfsalternativen, was eine manuelle Suche nach kostengünstigen Lösungen erschwert oder gänzlich unmöglich macht. Eine rein automatisierte Suche ist jedoch ebenso wenig erfolgversprechend, da häufig eine Vielzahl nicht-formaler Rahmenbedingungen zu beachten sind, die teilweise erst während der Suche nach möglichen Lösungen konkretisiert werden.

In dieser Arbeit wird ein teilautomatisiertes Verfahren zur Optimierung von Hardwarearchitekturen eingebetteter Systeme vorgestellt, das Entwickler bei der Erweiterung existierender Systeme um neue Funktionalität in Form von Softwaretasks unterstützt. Ein zweistufiges Optimierungsverfahren durchsucht den durch Rahmenbedingungen definierten Entwurfsraum nach gültigen Allokationen der Softwaretasks auf die Hardware-Architektur. Falls erforderlich können vorhandene Prozessoren durch leistungsfähigere ersetzt werden oder neue Prozessoren hinzugefügt werden, wobei nach möglichst kostengünstigen Hardware-Architekturen gesucht wird.

Das Optimierungsverfahren nutzt den häufig bei größeren eingebetteter Systemen vorzufindenden hierarchischen Aufbau des Gesamtsystems aus Hardware-Subsystemen aus: Zunächst werden in einem globalen Optimierungsschritt Vorplatzierungen der zusätzlichen Softwaretasks auf Subsysteme, basierend auf einer abstrakten Charakterisierung der benötigten und verfügbaren Rechenkapazität, bestimmt. Dann werden in für jedes Subsystem separat ausgeführten lokalen Optimierungsschritten die vorplatzierten Tasks unter Berücksichtigung aller Subsystem-spezifischen Rahmenbedingungen auf Prozessoren platziert. Softwaretasks, die nicht platziert werden konnten, werden iterativ in zusätzlichen globalen Optimierungsschritten weiter bearbeitet.

Sowohl für den globalen als auch für den lokalen Optimierungsschritt werden exakte Optimierungsverfahren vorgestellt. Anschließend werden die Ergebnisse einer umfassenden Evaluierung auf Basis dreier Benchmarks dargelegt. Im Rahmen dieser Evaluierung wurden beide Verfahren jeweils mit alternativen Ansätzen verglichen.

Abstract

The effort for the development of a safety-critical embedded system can be reduced tremendously if a similar existing system is used as basis which is then extended by additional functionality. However, this is a very challenging task because in general for both the already integrated and the new parts of such a system complex constraints have to be satisfied to guarantee their correct functionality. Furthermore, larger embedded systems are typically realized as distributed systems with multiple processors connected by a complex communication infrastructure. This leads to a huge number of design alternatives suitable for the extension of such a system thus complicating the manual search for cost-efficient solutions or even rendering it impossible. Searching entirely automatically is not too promising as well because usually lots of informal requirements have to be satisfied, some of which are concretized while already searching for possible solutions.

In this work a semi-automatic approach for the optimization of hardware architectures of embedded systems is presented that supports developers in extending existing systems by adding additional functionality implemented as software tasks. The two-tier optimization process explores the design space defined by constraints for valid allocations of the software tasks to the hardware architecture. If necessary, existing processors can be replaced by more powerful ones or additional processors can be integrated while aiming for a cost-efficient hardware architecture.

The optimization approach exploits that larger embedded systems typically use a hierarchical structure where the hardware architecture is composed from hardware subsystems: Firstly, a global (system-wide) optimization step computes pre-allocations of all additional software tasks onto subsystems based on an abstract characterization of the required and provided computation capacity. Separately for each subsystem, the pre-allocated tasks are then allocated to processors by local optimization steps under consideration of all subsystem-specific constraints. Software tasks that could not be allocated are handed back to the global tier for being allocated in later iterations.

Exact optimization methods are presented for both the global and the local optimization steps. Finally, the results of an extensive evaluation based on three benchmarks are presented. In this evaluation both optimization methods have been compared with alternative approaches.

Contents

List of Tables

List of Figures

Acronyms

ASIC	Application-Specific Integrated Circuit.
CAN	Controller Area Network.
DSE	Design Space Exploration.
DSP	Ditigal Signal Processor.
ECU	Electronic Control Unit.
GCC	GNU Compiler Collection.
GLPK	GNU Linear Programming Kit [Mak10].
TDMA	Time Division Multiplex Access.
WCET	Worst Case Execution Time.
WCRT	Worst Case Response Time.
WCTT	Worst Case Transmission Time.

List of Symbols

$\mathcal{A}$	Allocation.
taskalloc	Function of a given allocation for allocating tasks to ECUs.
$alloc^{\texttt{initial}}$	Initial allocation (part of the problem description).
msg	Function of a given configuration for assigning the corresponding set of messages to any given signal.
$\mathcal{A}^{\texttt{subsys}}$	Allocation projected on Subsystem *sub*.
allowed^{B}	Allowed Bus types function.
$\psi^{\texttt{ECUs}}$	Function which assigns the set of allowed ECUs to each task.
allowed^{E}	Functions for assigning the allowed ECU-types to each ECU.
$\psi^{\texttt{types}}$	Function which assigns the set of allowed ECU-types to each task.
hw	Architecture.
$G^{\texttt{hw}}$	Hardware Architecture Graph.
$G^{\texttt{hw}\star}$	Universe of hardware architecture graphs.
$slots^{\texttt{avail}}$	Set of available bus slots on a global bus used in a DSE problem.
$\text{memavail}^{E,\mathbb{E}}$	Function which maps to a given DSE configuration, a given ECU and a given ECU-type the maximal size of memory available for allocating additional tasks to that ECUs with that ECU-type.
$\text{utilavail}^{E,\mathbb{E}}$	Function which maps to a given DSE configuration, a given ECU and a given ECU-type the maximal utilization available for allocating additional tasks to that ECUs with that ECU-type.
mbl	Memory Base Load function for a given configuration, a given ECU and a given ECU-type.
$\text{mbl}^{Sub,\mathbb{E}}$	Memory Base Load function for a given configuration, a given subsystem and a given ECU-type.
ubl	Utilization Base Load for a given configuration, a given ECU and a given ECU-type.
$\text{ubl}^{Sub,\mathbb{E}}$	Utilization Base Load for a given configuration, a given subsystem and a given ECU-type.
$\text{choice}^{\texttt{exec}}$	User-specified partial function choosing for each task one executable per ECU-type.

$Bin^\star$	Universe of binary executables.
$block$	Blocking Time.
b	Bus.
bandwidth	Function which the bandwidth in bytes per second to each bus type.
B	Bus set.
buses	Function for deriving the set of buses of a given hardware design space.
$slot$	A bus slot.
slots	Totally ordered set of all bus slots belonging to the given bus, where the order reflects the position in one TDMA cycle.
length	Length of a TDMA Bus Slot in [s].
$size^{\mathtt{global}}$	Fixed Length of all Bus Slots of the Global Bus in [s].
$Slots^\star$	Universe of bus slots.
$\mathfrak{b}$	Bus Type.
class	Class of a given Bus Type, either priority based (`PB`) or time (slot) based (`TB`).
$\mathbb{B}$	Bus Type Set.
$\mathbb{B}^\star$	Universe of bus types.
$B^\star$	Universe of buses.
ω	Busy Period.
$conf$	Configuration.
$CONF^\star$	Universe of configurations.
$\psi^{\mathtt{allowedSubsys}}$	Constraint: Specifies for each tasks a set of allowed subsystems.
$\psi^{\mathtt{sameSubsys}}$	Constraint: Specifies sets of tasks which always must be allocated to the same subsystem.
$\psi^{\mathtt{onBus}}$	Constraint: Function assigning to each bus the set of signals which must be allocated there (via a message) in any case.
$\psi^{\mathtt{always}}$	Constraint: Set of sets of tasks which should always go on same ECU.
$\psi^{\mathtt{diffSubsys}}$	Constraint: Specifies sets of tasks which must never be allocated to the same subsystem.
$\psi^{\mathtt{never}}$	Constraint: Set of all sets of tasks which should never go on same ECU.
$constr$	Constraint specification.
cost	Cost function.

$cost^{\mathtt{max}}$	Cost Limit for one Hardware Subsystem.
$cost$	Function giving the cost for a given subsystem configuration.
d	Deadline.
$\mathrm{deadline}^{\mathtt{msg}}$	Function assigning local deadline to messages either as message properties (no subscript) or in the context of a given configuration.
$\mathrm{deadline}^{\mathtt{signal}}$	Function assigning local deadline to signals either as signal properties (no subscript) or in the context of a given configuration.
$\mathrm{deadline}^{\mathtt{task}}$	Function assigning local deadline to tasks either as task properties (no subscript) or in the context of a given configuration.
deg	Degree of graph vertex (number of connected edges).
dom	Domain of a function.
p	DSE Problem instance or Local Analysis Problem if Subsystem is specified in subscript.
$\mathcal{P}^\star$	Universe of DSE Problems.
Sol	DSE Solution instance.
e	ECU.
E	ECU Set.
$E^{\mathtt{empty}}$	Set of ECUs onto which no tasks are allocated by given configuration.
ecus	Function for deriving the set of ECUs of a given hardware design space.
$E^{\mathtt{NotEmpty}}$	Set of ECUs onto which at least one task is allocated by given configuration.
t	ECU Type.
type	ECU type function assigning ECU-types to ECUs in the context of a given configuration.
mem	Function which assigns the size of the available memory to each ECU Type for a given DSE problem.
$\mathbb{E}$	ECU Type Set.
$\mathbb{E}^\star$	Universe of ECU-types.
$E^\star$	Universe of ECUs.
$Edges^{\mathtt{hw}}$	Set of edges used in a hardware architecture tree.
$Edges^{\mathtt{sw}}$	Set of edges of a task network.
$\mathrm{deadline}^{\mathtt{e2e}}$	(Partial) function for assigning end-to-end deadlines to paths from the root vertex to a leaf vertex of a task tree.

$:\Longleftrightarrow$	Equivalent by definition.
$\texttt{false}$	Boolean *false*.
κ	Fixed-priority function for signals with MaxWCET.
θ	Fixed-priority function for signal with Spare-Time.
δ	Fixed-priority function for task with MaxWCET.
γ	Fixed-priority function for task with for Spare-Time.
$p^{\texttt{glob}}$	Global Analysis Problem instance.
$b^{\texttt{global}}$	Global bus.
$\mathrm{prio}^{>}$	Function returning a set of tasks (or signals) which are allocated to the same resource and have a priority higher than the specified task (signal).
$\deg^{-}$	In-degree of a vertex of a directed graph (number of incoming edges).
isGw	Boolean function: True if ECU is gateway to global bus, otherwise false.
$util^{\texttt{max}}$	Maximal Utilization Constant.
$\mathfrak{m}$	MaxWCET.
m	Message.
msgalloc	Function of a given allocation for allocating messages to buses.
M	Set of Messages.
$M^{\star}$	Universe of messages.
$\mathbb{N}$	Natural numbers.
$\perp$	Empty ECU type (e.g. no ECU type chosen).
NP	Class of non-deterministic polynomial-time problems.
typecount	Function which gives for a given DSE configuration the number of ECUs typed by a given ECU-type in a given subsystem.
$slots^{\texttt{remain}}$	Number of of bus slots on a global bus that are still available according to a given configuration.
$\mathcal{T}^{\texttt{odd}}$	Odd Set: Contains all pre-allocated tasks which could not be allocated to a subsystem during local analysis.

$\deg^+$	Out-degree of a vertex of a directed graph (number of outgoing edges).
$\nrightarrow$	Partial Function.
paths	Set of all those paths found in the set of task trees, where the root vertex and one leaf vertex are contained.
$penalty^{\mathtt{deadsynth}}$	Sum of penalties caused by deadline synthesis.
$\mathrm{penalty}$	Penalty function which assigns penalty cost to each task incuring if the task is not allocated during local analysis.
$penalty^{\mathtt{slot}}$	Sum of penalties caused for each additionally used global bus slot.
$\mathrm{period}^{\mathtt{msg}}$	Period Function for messages.
$\mathrm{period}^{\mathtt{sig}}$	Period Function for signals.
$\mathrm{period}^{\mathtt{task}}$	Period Function for tasks.
P	Class of deterministic polynomial-time problems.
$\mathcal{P}$	Power set (set of all subsets).
$conf^{\mathtt{pre}}$	Pre-allocation configuration.
$\mathrm{needsGlobal}$	Predicate: True if message on global bus is required for signal, otherwise false.
$\mathrm{needsGlobal}^{\mathtt{GA}}$	Predicate used in Global Analysis: True if message on global bus is required for signal, otherwise false for a given global analysis problem, a pre-allocation and a signal.
$\mathrm{needsLocal}$	Predicate: True if message on local subsystem bus is required for signal, otherwise false.
$\hat{d}$	Pseudo Deadline.
$\hat{\mathbb{D}}$	Set of pseudo deadlines.
ran	Range of a function.
$\mathbb{R}$	Real numbers.
recv	Set of receiver tasks of a given signal in a given task network.
$\mathrm{reqmsg}^{\mathtt{t2t}}$	Set of messages required for transmitting a signal from a sender task to a receiver task.
$\mathrm{slotsreq}$	Number of required Slots on the global bus for a given message in the context of a given DSE problem.
sender	Sender Task for a signal in a given task network.
$\dot{\cup}$	Disjoint set union.
s	Signal.
S	Set of all signals.

$S^{\texttt{alloc}}$	Set of all signals which are allocated via a message to a bus.
$S^{\texttt{always}}$	The set of signals which have to be on the global bus according to a given configuration.
signals	Function giving the finite set of signals contained in a task network.
$S^{\texttt{maybe}}$	The set of signals which may have to be be on the global bus according to a given configuration (useful for e.g. partial allocations).
$S^{\texttt{never}}$	The set of signals which must not to be on the global bus according to a given configuration.
$\overline{S}$	Set of pre-allocated signals where the sender or one receiver are (pre-)allocated to a subsystem but which have no message on the local bus yet.
$S^{\star}$	Universe of signals.
bytes	Number of bytes of a signal.
$\mathfrak{s}$	Spare-Time.
$\mathrm{utilavail}^{Sub,\mathbb{E}}$	Function which maps to a given DSE configuration, a given subsystem and a given ECU-type the maximal utilization available for allocating additional tasks to ECUs in that subsystem with that ECU-type.
sub	Subsystem.
subsys	Subsystem function.
Sub	Subsystem Set.
subsysset	Function giving the set of subsystems for a given hardware architecture tree.
$\mathrm{subsys}^{\mathcal{T}}$	Function which maps a tuple consisting of an DSE Problem, a configuration and a task to the subsystem where the task is allocated or $\perp$ if the task is not (yet) allocated.
$S^{\star}$	Universe of subsystems.
$\mathrm{memavail}^{Sub,\mathbb{E}}$	Function calculating for a given configuration the memory available in a given subsystem for a given ECU-type.
$\mathrm{memreq}^{Sub,\mathbb{E}}$	Function calculating for a given configuration the required memory in a given subsystem for a given ECU-type.
$\mathrm{utilreq}^{Sub,\mathbb{E}}$	Function which maps to a given DSE configuration, a given subsystem and a given ECU-type the utilization caused by all the tasks pre-allocated to that subsystem and that ECU-type (not including the tasks which are already allocated to specific ECUs.

$\mathrm{memreq}^{Bin^\star}$	Partial function which assigns to binary executables and ECU-types the required memory size.
memreq	Function which assigns the required memory size for each ECU Type to each task in the context of a given DSE problem.
tn	Task Network.
$TN^\star$	Universe of task networks.
$\mathcal{T}$	Task Set.
$\mathcal{T}^{\mathtt{alloc}}$	Set of allocated tasks for a given allocation function.
tasks	Function giving the finite set of tasks contained in a task network.
$\mathcal{T}^{\mathtt{pre}}$	Set of pre-allocated tasks (to a specified subsystem).
tasktype	Function which assigns an ECU Type to each Task (used for Global Analysis).
tasksubsys	Function which assigns a Subsystem to each Task (used for Global Analysis).
$tree$	Task tree instance.
$G^{\mathtt{sw}}$	Task tree graph.
$Trees$	Set of task trees.
$TREE^\star$	Universe of task trees.
$\mathcal{T}^\star$	Universe of tasks.
util	Utilization caused by a software task in the context of a given DSE problem.
τ	Software Task.
$\lambda^{\mathtt{TDMA}}$	Length of the TDMA round on the global bus for a given DSE problem in [s].
$cost^{\mathtt{all}}$	Total Cost of a given DSE Solution.
`true`	Boolean *true*.
$\mathcal{T}^{\mathtt{unalloc}}$	For a given configuration: Set of unallocated tasks part of the corresponding DSE problem.
`undef`	Undefined.
vertices	Set of vertices of given task tree or task tree set.
$wcet$	Worst Case Execution Time.
$\mathrm{wcet}^{Bin^\star}$	Partial function assigning to binary executables and ECU-types the corresponding worst case execution time.

$wcet^{\texttt{eff}}$	Effective Worst Case Execution Time (depending on the ECU-type) per ECU (0 if task is not ECU).
wcet	Function assigning a worst case execution time to tasks in the context of a given DSE problem.
r	Worst Case Response Time.
wctt	Function for assigning the Worst Case Transmission Time to messages for each of the bus type.
$\mathrm{prio}^{>}$	Predicate is true if signal given as first argument would have higher priority on the same bus than signal given as second argument.
$\mathrm{prio}^{>}$	Predicate is true if the task given as first argument would have higher priority on the same ECU than the task given as second argument.

1. Introduction

Most product innovations of the last ten to twenty years for improving our daily life would not have been possible without the use of modern computer technology. While in many cases the customers of such products are well aware of the presence of the computers, for example in smart phones and home entertainment systems, the majority of computer systems today are used for control and regulation jobs as integral part of technical systems without ever being noticed by the customers. Such computer systems are called **embedded systems**. An embedded system performing tasks which may lead to injury or death of people if the system fails (for example an anti-lock breaking system) is called a **safety-critical embedded system**.

The development of safety-critical embedded systems requires a well-structured development process ranging from initial requirements analysis to hardware software co-design to certification and delivery of the final product. But the product life-cycle management does not end with the delivery of the product but — as the term suggests — consists of additional activities such as maintenance, expansion of the original product, etc. spanning the whole life-cycle of the product. The most famous development process suitable for the development of safety-critical embedded systems is the V-Model. Developing products using such a rigid process model usually is expensive and time-consuming though unavoidable for successful certification of safety-critical systems.

Development cost and time can be reduced tremendously if some of the comprehensive development process steps can be carried out only for parts of the new system instead of the whole system or if they can be avoided completely. For this reason new products are often developed based on already existing products thus enabling the re-use of many parts of the former product during the development process. If done well, effort for development and testing of those existing parts can be avoided or at least significantly reduced. As an example, the automotive industry typically re-uses up to 90% of electronic components of previous car generations. However, the re-use of software components is only done to a very small extent ($\sim 10\%$, see [Bro06] for reference).

1.1. Contributions of this Work

This PhD thesis contributes an approach for extending safety-critical embedded systems with the focus on integrating additional functionality implemented as software tasks and signals representing the data flow between tasks. A modular **two-tier Design Space Exploration (DSE) process** is proposed which supports engineers in finding solutions with cost-minimal modifications of the system's hardware architecture while ensuring satisfaction of all specified constraints. Engineers can guide the DSE process by running it iteratively and tighten the explicitly specified constraints based on their expert knowledge. This semi-automatic user-driven optimization process avoids counter-intuitive solutions and therefore helps to increase acceptance by engineers.

The approach exploits the hierarchical structure of embedded systems composed of multiple hardware subsystems by decomposing the optimization problem into two tiers: A system-wide **global analysis** pre-allocates new software tasks to subsystems by predicting and balancing the required and the available computation capacity per subsystem and additionally considering the communication between tasks on different subsystems. A subsystem-level **local analysis** calculates an allocation of the pre-allocated tasks to ECUs of the subsystem without considering the other subsystems. All tasks which could not be allocated are returned to the global analysis and pre-allocated again to other subsystems in the following iteration of the whole process until the DSE process terminates.

The two-tier optimization process aims for scalability by abstracting from subsystem-specific details during the global analysis and abstracting from the details of other subsystems during the local analysis.

This work demonstrates the benefits of this concept by contributing two exact optimization algorithms, one for the global and one for the local analysis. The exact global analysis approach has been published in [CST11], the exact local analysis approach in [Tha+10].

The results of an extensive evaluation are presented using multiple benchmarks consisting of DSE problem instances based on an academic example (see [TBW92]), artificially generated problem instances, and problem instances derived from the results of the case study ViDAs. The case study is based on the results of a student's project group with the same name. The students participating in that project group had the task to create a driver assistance system.

1.2. Context

This work has been created in the context of the transregional collaborative research center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, see [Bec+12]) but was also inspired by many research projects realized at the Institute for Information Technology OFFIS (see [Anob]).

Figure 1.1 shows the whole process developed in the subproject R2 of AVACS with a focus on the process steps to be carried out before running the actual design space exploration as proposed in this work. The upper left side of the process is dedicated to the extraction of timing-related properties from Matlab Simulink models and present that information using the formalism of function networks, as proposed in [Bük12]. The right side describes the necessary steps for generating code directly from the Matlab model and calculating the execution times for that code. Finally a task network is obtained.

The process step on the lower left named "Design Space Exploration" represents the task solved in this PhD thesis.

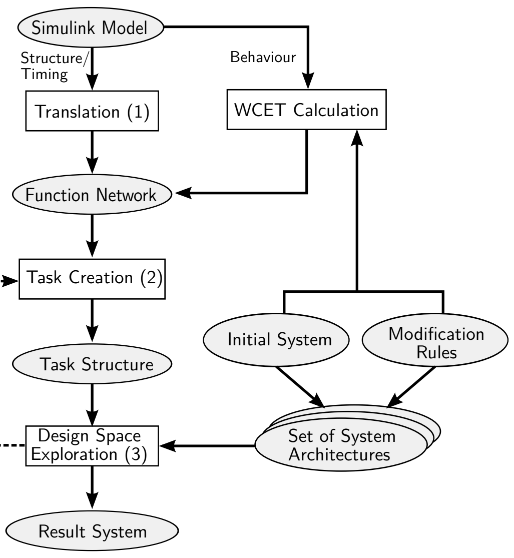


Figure 1.1. – AVACS, Subproject R2: Task Creation and WCET Analysis

1.3. Overview of Related Work

Many publications have defined different design space exploration problems for the development of embedded systems in the past and proposed various approaches for solving them.

However, the specific design space exploration problem presented in this thesis, which incorporates both the problem of allocating a set of software tasks onto a hardware architecture and the problem of optimizing that hardware architecture for cost, and the proposed solution based on exact optimization algorithms exploiting the hierarchical hardware structure of embedded systems, makes the presented approach unique.

The problem of allocating a set of software tasks onto a hardware architecture is addressed in many publications but the majority of them assume that the hardware architecture is not changed (for example [TBW92]). Publications on design space exploration of hardware architectures mainly focus on the hardware level without considering allocation problems (for example [Sil+11]). Only few publications combine the allocation problem with a design space exploration of the hardware architecture, one of which is [Mad+07].

Hierarchical optimization approaches exploiting the hierarchical structures of embedded systems can be found in some of the works (for example [ARS00]), but in most of them the hierarchical optimization is not based on exact optimization methods.

In fact, exact optimization methods are rarely applied to design space exploration problems due to the problem's huge complexity. In most publications heuristic methods are favored such as simulated annealing or evolutionary algorithms (for example [Hau+03]).

A detailed discussion of related work can be found in Chapter 7 on page 185.

1.4. Outline

The remainder of the thesis is organized as follows. A description of the foundations of this work is given is given in Chapter 2. Chapter 3 formally defines the problem to be solved and describes the approach presented in this work. In Chapter 4 an exact optimization approach for the global analysis problem is described. An approach for the local analysis problem called Spare-Time/MaxWCET analysis is presented in Chapter 5. In Chapter 6 first the implementation of the DSE tool Zerg is described, then the results of the evaluation of the proposed methods in comparison with alternative approaches is presented. The work is concluded in Chapter 7 with a summary of the findings, a detailed discussion of related work and remarks on future work.

2. Foundations

This chapter provides an overview on the research area in which this work is situated.

2.1. Terms and Definitions

This section provides a short overview on embedded systems and introduces some terms used throughout this thesis. In the context of this work an **embedded system** is a hardware architecture providing means for performing computations and internal and external communication, and software for realizing the systems functions expressed as a **task network**. A glossary can be found in the appendix of this work.

2.1.1. Embedded System

In the published scientific literature in the field of computer science there is no generally accepted or commonly used definition of the term **embedded system**.

In the context of this work an **embedded system** is defined as a computer system integrated in a technical product with the purpose of interacting with the environment of the product via sensors and actuators in order to perform control and/or regulation tasks. An embedded system performing tasks which may lead to injury or death of people if the system fails is called a **safety-critical embedded system**. An **embedded real-time system** has to satisfy hard timing requirements. In a **software-intensive embedded system** the functionality is mainly implemented with software.

The focus of this work is on **software-intensive safety-critical embedded real-time systems**.

2.1.2. Hardware Architecture

The **hardware architecture** of an embedded systems usually consists of various different classes of hardware components. Sensors are used to gather information about the system's environment. Processing units such as microprocessors, application-specific integrated circuits (ASICs), digital signal processors (DSPs), etc. perform computations based on that information, communicate with each other over communication channels and provide data to actuators which manipulate the system's environment. For an excellent overview on embedded system's hardware see [Chapter 3 in Mar10, pp. 119–175]. Several websites such as [Anoa] and [Ano13] provide introduction to the subject and lots of additional information. In this work ASICs and DSPs are not considered.

2.1.3. Design Space

The term **design space** describes the set of valid solutions for a given design task by specifying constraints.

2.1.4. Design Space Exploration

Design Space Exploration describes the process of searching a given **design space** for valid solutions. Usually an **objective function** is used to define what solutions to search for.

2.1.5. Hardware Design Space

In this work, the **hardware design space** is defined by specifying a **hardware architectural pattern** (see Section 2.1.6). The hardware architecture is limited to ECUs and buses and does not incorporate sensors and actuators.

2.1.6. Hardware Architectural Pattern

A **hardware architectural pattern** determines a set of possible hardware architectures (the hardware architecture design space) by specifying mandatory and optional hardware components. A hardware architectural pattern consists of a set of interconnected **logical hardware units** and specifies for each of them a set of allowed **physical hardware types**. The notion of logical hardware unit is used in this thesis to allow for specifying interconnection structures without directly including detailed specifications of the used hardware units. Not all hardware components of typical embedded systems are represented in this thesis' system model. The kinds of logical hardware units currently considered during the optimization process are **logical electronic control units (ECUs)** representing processing units and **logical buses** representing communication units.

A hardware architectural pattern is instantiated by choosing for each of the mandatory logical hardware units one of the allowed physical hardware types. Assigning physical hardware types is not required for optional logical hardware units. Only logical hardware units to which a physical hardware type has been assigned are part of the instance of the hardware architectural pattern.

In this thesis, only certain hardware architectural patterns are considered. All patterns have in common that they contain at least two different hardware subsystems and one bus called **global bus**. Each hardware subsystem consists of a set of ECUs connected via a **local bus**. Each ECU is part of exactly on hardware subsystem. In each hardware subsystem exactly one ECU is used as a gateway. Only the gateway ECUs are connected to both their subsystem-local bus and the global bus. A formal definition is given in Chapter 3.

2.1.7. Logical Electronic Control Unit (ECU)

In this work the term **Logical ECU** is used as part of a hardware architectural pattern as a placeholder for a computing resource without specifying the actually type of that resource. In the remainder of this thesis the term **ECU** is used interchangeable with **logical ECU**. An ECU is typed by an ECU-type.

2.1.8. ECU-Type

An **ECU-type** characterizes all properties of an embedded microprocessor with all directly connected hardware components such as (external) caches and memory, I/O ports, communication controllers, etc. This includes which physical microprocessor is used, which memory hierarchy is used, the size of the memory, how many I/O ports are available and how they are configured, etc. Most of these parameters like e.g. the configuration of the cache hierarchy are not directly used during the optimization process. But they are influencing properties of software tasks running on that type of hardware, namely their execution times which are determined previously to the optimization phase by tools such as *aiT* (see [Wil+08; Abs]). The only property of an ECU-types used during the design space exploration process is its cost value (e.g. the price of the physical hardware).

2.1.9. Logical Bus

Analogously to the concept of ECU each **Logical bus** represents a communication bus as part of an architectural pattern without further specifying the hardware. In the remainder of this thesis the term **bus** is used interchangeable with **logical ECU**. Each bus is typed by a bus-type.

2.1.10. Bus-Type

In embedded systems with multiple electronic control units those units usually are interconnected via physical links. Such links can be realized as simple as a wire between only two ECUs for unidirectional or bidirectional communication via standard or proprietary communication protocols. More complex setups use **communication buses**. A communication bus is a communication channel shared by two or more ECUs. Many different communication protocols exists for communication buses such as Controller Area Network (CAN), FlexRay, Local Interconnect Network (LIN) and Media Oriented Systems Transport (MOST) in the automotive industry, Avionics Full Duplex Switched Ethernet (AFDX) in aerospace industry, and ProfiBus and EtherCAT (automation industry), only to name a few.

In this thesis, the **bus-types** are used to type logical buses. Two classes of bus types are considered: A TDMA bus type which uses a time-triggered bus access protocol (e.g. FlexRay using only the static segment) and a priority-based bus type (e.g. a controller area network CAN bus). In [Kop98] a comparison of both protocols can be found.

Each bus type is characterized by the bus type class (TDMA or Priority-based) and its bandwidth.

2.1.11. Bus Slot

A **bus slot** represents one time slice of a TDMA-bus where messages can be allocated.

2.1.12. Software Task

The part of an embedded system implemented in software usually consists of a set of communicating software processes which are running concurrently on a complex embedded hardware architecture with multiple ECUs connected by communication buses. In the embedded community such software processes are called software tasks. A precise definition of the term **task** is provided by IEEE: A task is "[...] a sequence of instructions treated as a basic unit of work by the supervisory program of an operating system. [...]" [IEE90]. In [LY03] a software task is defined as "an independent thread of execution that can compete with other concurrent tasks for processor execution time." Software tasks are derived by iteratively decomposing the system's functionality into smaller functional units until a suitable level of granularity has been obtained.

In the context of this work, the term **software task** is used to refer to a software artifact which

- realizes one or more specific functions of a system as a sequence of I/O and computation steps
- is small enough to be realized as one single operating system process which is regularly scheduled for execution by the scheduler of an operating system
- runs to completion every time it is started (but might be preempted by higher priority tasks allocated to the same processor)

Furthermore, it is assumed that the implementation of all software tasks follows a certain schematic: During the execution of a software task first the task's inputs are read (optionally), then all computations are done, and finally the task's output is written (optionally).

Each software task is assumed to have a non-empty set of functionally equivalent implementations. An implementation may be optimized for a particular hardware platform. Binary executables are compiled for all hardware platforms for which a suitable implementation has been provided by using platform-specific compiler tool chains.

In Figure 2.1 an example of a software task is given. The function *Velocity Calculation* is specified as a MatLab/Simulink™ model and source code has been generated directly from the model using TargetLink™. The result is a generic source code file Source$_1$. This file is now compiled to binary code for an ECU-type *ARM5-NoDSP*, which uses an ARM5 processor but has no on-board DSP. Two different compilers are used, first the GNU Compiler Collection (GCC), resulting in Binary$_{1,1}$ and second the compiler included in the ARM Development Studio-5 (DS-5) resulting in Binary$_{1,2}$. In the example,

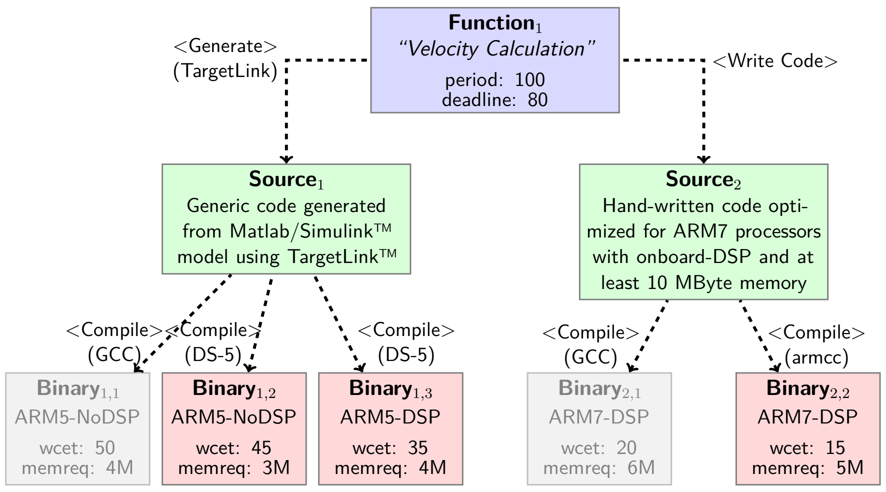

Function₁
"Velocity Calculation"
period: 100
deadline: 80
<Generate> (TargetLink)
<Write Code>
Source₁
Generic code generated from Matlab/Simulink™ model using TargetLink™
Source₂
Hand-written code optimized for ARM7 processors with onboard-DSP and at least 10 MByte memory
<Compile> (GCC)
<Compile> (DS-5)
<Compile> (DS-5)
<Compile> (GCC)
<Compile> (armcc)
Binary₁,₁ ARM5-NoDSP wcet: 50 memreq: 4M
Binary₁,₂ ARM5-NoDSP wcet: 45 memreq: 3M
Binary₁,₃ ARM5-DSP wcet: 35 memreq: 4M
Binary₂,₁ ARM7-DSP wcet: 20 memreq: 6M
Binary₂,₂ ARM7-DSP wcet: 15 memreq: 5M

Figure 2.1. – Example: Software Task with Source and Binary Code, Annotations of Worst Case Execution Time and Memory Consumption. The unshaded Binaries have been selected for Design Space Exploration.

it is assumed that a second ECU-type *ARM5-DSP* featuring an on-board DSP is not supported by GCC (or that GCC would not be able to make use of the DSP). Therefore only a binary Binary$_{1,3}$ is compiled using the DS-5 compiler with enabled optimizations for utilizing the DSP.

Additionally, a more expensive ECU-type *ARM7-DSP* with on-board DSP and 10 MByte of RAM is evaluated. For this hardware platform hand-written source code is developed. This implementation uses all available hardware features, resulting in source code Source$_2$. It is compiled using GCC (resulting in Binary$_{2,1}$) and *armcc* for optimized binary code (resulting in Binary$_{2,2}$).

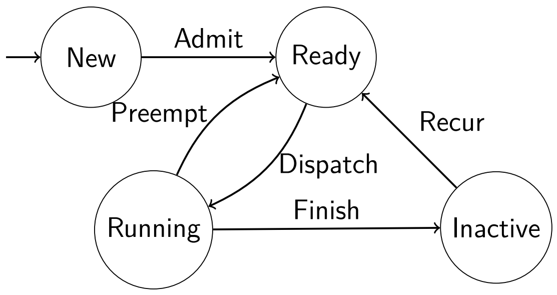


Figure 2.2. – Life-cycle of a Software Task

Whenever multiple software tasks are allocated to the same ECU an **(real-time) operating system** is required which schedules the set of task. This work focuses on preemptive scheduling.

Figure 2.2 shows the states during the life-cycle of a software task that is scheduled by a (preemptive) scheduler of a real-time operating system. Both the automaton and the used terminology are loosely based on the *Five-State Model* described in [Sta12, pp. 136-140]. The life of newly created task begins in the *New* state. In this thesis it is assumed that all tasks are created during system startup. The task is then admitted after some time and enters the *Ready* state. When dispatched by the operating system scheduler the task enters the *Running* state. If task is preempted it enters the *Ready* state. Dispatching a preempted task resumes its execution. When the task finishes its execution it enters the *Inactive* state. After a specified time interval it recurs and enters the *Ready* state again. The main differences compared to [Sta12] are that tasks never block during execution (exclusive access to resources is not considered, which means that there is no *Blocked* task state) and that tasks recur after a specific time duration. Each single recurrence of the same task is called a *task occurrence* in the following text.

The *activation* of a task is the event of entering the *Ready* state after it has been newly created or after being in the *Inactive* state. The **activation period** is the time interval between two consecutive activations of a software task.

To each task a local **deadline** can be associated. The local task deadline is an upper bound for the time interval in which every single occurrences of that task is executed starting from the point in time where that occurrence initially becomes ready until it finishes its execution. Specifying local deadlines for software tasks is optional. The synthesis of local deadlines is part of the design space exploration for all tasks where no local deadline has been specified yet.

In this work it is assumed that tasks may receive maximal one signal (for details see Section 3.1.2.2).

While the activation period and the deadline of a software task are independent from the hardware platform, the **worst case execution time** (upper bound for the time required to execute the task without considering preemptions by higher priority tasks) and the maximal **memory consumption** depend on how a corresponding binary implementation performs on the specific hardware platform it has been allocated to. Both properties depend on the ECU-type which has been chosen for the ECU where the task is allocated.

The calculation of the WCET and memory consumption is not part of this work but instead has to be done in a separate analysis step beforehand. During this analysis step the values are determined separately for each of the ECU-type based on the platform-specific binary code either by applying measurement tools (e.g. *Chronos* [Cha+12] or *ChronSIM* [INC12] with extension *ChronEst*) or by using formal analysis tools (e.g. *aiT* [Abs]). Neither for the measurement nor for the analysis based approaches it is important how the binary code was created, as long as all information required for applying the measurements/analysis is available (which might include code annotations regarding upper bounds for the number of iterations of loops and similar) including detailed information about the ECU-type.

Figure2.1 also shows for each binary implementation (fictitious) values for the worst case execution time (Property name: `wcet`) and maximal memory consumption (Property name: `memreq`) respectively. If for a software task multiple binary executables exist for an ECU-type, the user has to decide which of them to use during the DSE phase. In the figure, the binaries which have not been chosen by the user are shaded. If for a given software task no binary implementation exists for a given ECU-types than both its WCET and its memory consumption are assumed to be 0 and a constraint has to be added prohibiting the allocation of the task to any ECUs typed by that ECU-type (see Section 3.1.4 on page 38).

2.1.13. Signal

A **signal** represents the abstract data flow between software tasks without specifying how the data would be transmitted from the sender task to the receiver tasks.

For example, if the sender and all receiver tasks are residing on the same ECU then the data flow would be implemented by using internal mechanisms of the operating system e.g. shared memory. If however at least one receiver task is allocated to a different ECU then the signal data has to be transmitted on one or more communication buses using a messages. If a signal has to be transmitted via multiple buses connected via gateway ECUs, multiple messages are required, one for each bus.

2.1.14. Message

A **message** is used to transmit data (represented by an abstract signal) via a bus of a hardware architecture. An allocation of a task network onto a hardware architecture associates to each signal which has to be transmitted between different ECUs a set of messages and allocates them to buses. Each message consists of one or more data packets called data frames, designated to transmit the data flow of a given signal over a bus. The distinction between messages and frames is due to the bus protocols that require a fixed or maximum size for each transmitted data packet.

2.1.15. Task Network

In this thesis the software architecture of an embedded system is represented as a **task network** consisting of tasks and signals representing data flow between tasks. See Chapter 3 for details.

2.1.16. Allocation

According to the IEEE standard 610.12-1990, **allocation** is "(1) The process of distributing requirements, resources, or other entities among the components of a system or program. (2) The result of the distribution in (1)" [IEE90].

In the context of this work an allocation is the link between task networks and hardware architectures. An allocation assigns some or all tasks of a task network to ECUs, each message to one signal and each message to one bus. Allocations are used for both the specification of a DSE problem (partial allocation) and as part of solutions for DSE problems (total allocations).

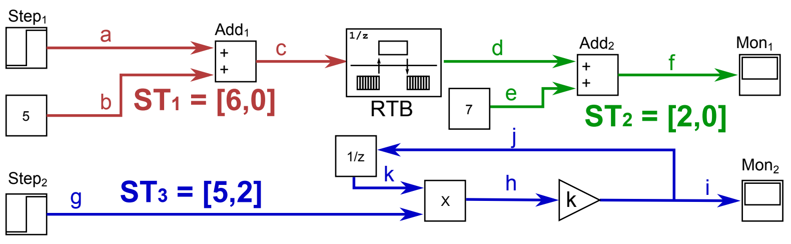


Figure 2.3. – Example Matlab Simulink Model (Source: [Bük12])

2.2. Real-Time

This section gives an overview of Matlab Simulink to show how commercial off-the-shelf modeling tools are applied in an industrial context for specifying real-time systems. Then it describes the concept of *function networks* (proposed in [Bük12]) which has been specifically designed to formally capture the semantics of Matlab Simulink models with the focus on real-time requirements. Then execution time analysis is introduced and finally concepts for scheduling and schedulability analysis are explained.

2.2.1. Overview: Matlab Simulink

Matlab Simulink® "is a block diagram environment for multi-domain simulation and Model-Based Design" [Mat12]. Matlab Simulink is widely used in industry because it combines a graphical modeling language suitable for modeling differential equations with a powerful simulation engine and code generators for creating C and C++ source code. An example Simulink model is shown in Figure 2.3.

On the left side of the example three blocks are specified. The top and the bottom blocks are *step blocks* used for modeling step functions (by generating steps between two specified constant levels at specified times). The block in the center of the left half provides a constant. The output of the upper block and the constant block are used as input for an *add block*. Those three blocks are configured to have the same sample time of $[6, 0]$ which means that they are running periodically every 6 time units, with an initial offset of 0 time units. The output of the *add block* is connected to the input of a so-called *rate transition block*. This block translates the sample rate of the three blocks to a new sample rate of $[2, 0]$ which is then used for the blocks on the upper right of the diagram. A constant provided by a second *constant block* is added to the output of the *rate transition block* and the result is sent to a *monitor block*. The lower half of the example model shows that loops are possible in Matlab, here by using an unit *delay block* which provides the result of the previous computation back to the *multiplier block* (in the center). All the blocks in the lower half are synchronized to a sample rate of $[5, 2]$.

When simulating with Simulink the order in which the results of blocks are computed is determined by several rules ensuring that all dependencies between blocks are resolved properly. It is assumed that the computation of the single blocks does not consume time, and therefore by definition all computations perfectly fit into whatever sample rate has been chosen. Obviously some of these implicit assumptions are not valid anymore once source code has been generated for the blocks which then has been compiled to processor-specific machine code which has been distributed to multiple ECUs of an embedded systems. Of course, executing the corresponding machine code on a processor consumes time. Running multiple parts of the original Simulink model in parallel may result in race conditions if not done properly. For example, if in the above example the code generated for the add block on the upper right side is executed in parallel to the code for the rate transition block it might read an outdated value because the updated value is provided not fast enough by the rate transition block.

As a consequence, a semantics-preserving approach for utilizing the commercial code generators for Simulink models and for allocating the generated software tasks to the target hardware architecture is required which avoids the pitfalls stemming from the implicit assumptions used in Matlab Simulink. Such an approach is presented in [Bük12] where the implicit timing requirements used in Simulink models are made explicit using the formalism of **function networks**.

2.2.2. Overview: Function Networks

The notion of function networks has been invented first in [BMS09] to "extend the expressiveness of classical task networks by functional elements, and a finer differentiation of data and control flow by using data nodes and specific channels." [BMS09]. Functions networks are bridging the semantic gap between industrial high-level modeling tools such as Matlab Simulink on the one hand and the scientific field of schedulability theory on the other hand.

For more detailed information about function networks and their properties see [Bük12, Chapter 3]; for the transition between Matlab Simulink models into function networks see [Bük12, Chapter 4].

On the way from high-level modeling tools to distributed embedded systems obtaining function networks is only an intermediate step. As stated before, the generation of source code and the compilation of that code for one or more ECU-types in order to obtain machine code can be done in parallel to the extraction of timing requirements. The commercial tools available for the code generation include Matlab Coder™ (formerly known as Real-Time Workshop®) and dSpace TargetLink®.

The actual size of the code fragments generated for each of the Simulink blocks can vary depending on the complexity of the operation to be performed by the blocks. While the source code required to implement a Simulink add block might consists only of a single line of code other blocks might require more lines of code. In the end, the code generation process shall result in software tasks which can then be allocated to ECUs part of an embedded system where they are executed under the control of embedded operating systems. But multiple aspects have to be considered when deciding how to obtain those software tasks from the code generator.

There are two extreme approaches. In the first approach each Simulink block is translated into a separate software task. Following this approach, a huge number of software tasks would be created of which the majority would be very small in the sense that their execution times on the given ECU-type is very short. Running many small software tasks on the ECUs would cause huge overhead for the management of those tasks by the embedded operating system (e.g. due to lots of context switches whenever tasks are preempted) compared to only a relatively small fraction of the computation time remaining for performing the software tasks actual purpose.

The other approach is to combine all Simulink blocks (or at least all blocks with the same sample rate) into one huge software task (or one software task for each of the sample rates found in the Simulink model). With such a huge software task (or a small number of large tasks) the advantages of distributing software tasks among multiple ECUs of embedded hardware architectures could not be used as would be possible with a greater number of slightly smaller software tasks. Huge software tasks could even restrict the choice which ECU-types to use for each of the ECUs to the more powerful (and more expensive) types, because the task's execution times on less powerful ECU-types would be too long.

[Bük12, Chapter 5] describes a flexible approach for finding good compromises between the above extreme approaches.

2.2.3. Execution Time

An important property of software tasks is their execution time. The execution time is the time required for executing the corresponding binary code of a software task on a suitable ECU-type. More precisely, a software task can only be allocated to an ECU with an ECU-type if a binary implementation exists for that ECU-type and both a lower bound called **best-case execution time** (BCET) and an upper bound called **worst-case execution time** for the execution time of that binary on the ECU-type is known because those values are required for performing schedulability analysis.

Note that the execution time explicitly does not consider effects caused by other tasks on the same ECU, such as preemptions.

At first glance, automatically determining best-case and worst-case execution times of a software task represented by binary code compiled for a given ECU-type might look simple. But in fact it has proven to be a very complex problem (for source code implemented using a Turing complete language it is even undecidable, see [Wil+08]). This is because the (embedded) processors available today use lots of sophisticated techniques for improving the overall computation performance, for example pipelining, out-of-order-execution (data-driven execution) and speculative execution, complex memory hierarchies based on multi-level caches, and specialized machine instructions for speeding up common computations. While all those techniques today are indispensable for processor architecture to be commercially successful, they pose serious problems to timing analysis methods. It is very hard to predict the effects of those techniques on the execution time of software tasks because a perfect prediction would require complete knowledge about the internal state of the microprocessor, its caches and so on. An additional problem is that the execution time of software tasks in general depends on the inputs presented to the tasks, too. An overview on this research area is given in [Wil+08]. The paper distinguishes static methods which "do not rely on executing code on real hardware or on a simulator" and measurement-based methods which "execute the task or task parts on the given hardware or a simulator for some set of inputs" [Wil+08, p. 7]. Static methods have the disadvantage that they require precise processor models with lots of information about the internal structure of the processors and the memory hierarchies. Static analysis is capable of providing guaranteed lower/upper bounds by abstracting the processor and cache internal states. Furthermore, additional annotations to the source code are required for the analysis tools to determine properties such as bounds for the number of iterations for loops.

In contrary, measurement-based methods do not require knowledge about the internal states of the ECUs. The binary code is run on the actual ECU many times with different inputs provided while the execution time is measured. The lowest and largest measured values are used to derive lower/upper "bounds" by subtracting a pre-defined safety-margin from the lowest value (for the lower bound aka best-case execution time) or adding that safety-margin to the largest value (for the upper bound aka worst case execution time). The measurement-based method cannot guarantee that the best-case and worst-case execution times are really safe. Some rare combination of inputs or an unexpected internal state of the processor might lead to an execution time which is far smaller or larger than the smallest/largest observed one including the safety margin.

As execution time analysis is not subject of this thesis, it is assumed that the worst-case execution times of the software tasks of the design space exploration instance are given for all or a subset of the available ECU-types.

2.2.4. Overview: Scheduling and Schedulability Analysis

Whenever multiple software tasks are allocated to the same ECU **scheduling** has to be used to ensure that the ECU is made available to all the software tasks according to specific rules. The software tasks are said to be **scheduled**. A software task is **schedulable** if its worst case response time is smaller or equal to its deadline. The **worst case response time (WCRT)** of a task is the longest possible time duration measured from the invocation of a software task to the time the task finishes its execution considering all preemptions by other tasks and the operating system.

Arbitrating the access of multiple ECUs to a common communication bus is in many aspects similar to the scheduling of microprocessors. According to a set of rules exclusive access to the communication bus is granted to one of the connected ECUs for a limited time interval. Many different hardware implementations and bus protocols exists for different applications. In this thesis two different bus protocols are used: A Time Division Multiple Access (TDMA) protocol is used for the global bus which interconnects all hardware subsystems and a priority-based protocol for all local buses residing inside of those subsystems which interconnect all ECUs part of their subsystem. By using the TDMA protocol for the global bus a temporal decoupling of the hardware subsystems, for example release jitter — though not considered in this thesis — is avoided while communicating between subsystems.

An ECU is schedulable if all software tasks allocated to that ECU are schedulable. While the technical mechanisms required for arbitrating access to a communication bus are different than for ECUs many basic principles are similar. Analogously, a message allocated to a communication bus is said to be schedulable if its worst case transmission time is smaller or equal to its deadline. The **worst case transmission time** of a message is the longest possible time interval starting from the time at which an instance of a message has been written to the send queue of the sender task's ECU to the time at which this message has been written to the receive buffer of a receiver task's ECU (remind that multiple tasks may receive the same message).

A communication bus is schedulable if all its messages are schedulable. Finally, an embedded system is schedulable if all of its ECU and buses are schedulable. **Schedulability analysis** is used to test whether or not a given task/message is schedulable. How the schedulability test is done depends on the actually used scheduling method.

Several different approaches for scheduling exists. Two classes of scheduling approaches are distinguished: Static scheduling and dynamic scheduling.

In **static scheduling** the schedule, i.e. the time instances when tasks are dispatched, is statically configured. One way to achieve this is to combine all software tasks on the same processor during system design time into one huge software task. While combining the tasks different activation periods are considered. The resulting huge task is during runtime (re-)activated periodically by a timer of the operating system, depending on the actual implementation (see [Tan01, pp. 132]).

The scheduling methods used in this thesis belong to the class of **dynamic scheduling** approaches (scheduling decisions are taken at runtime), and can be further classified as **non-preemptive** and **preemptive scheduling**. Non-preemptive scheduling means that software tasks cannot be preempted by the operating system. Instead, they have to actively hand control back (cooperative scheduling). In contrary, when using preemptive scheduling, the operating system can actively preempt a currently running task at any time. Depending on the purpose and/or configuration of an operating system processor time is granted to software tasks based on different scheduling algorithms. This works focuses on preemptive scheduling.

In **round-robin scheduling** a time interval is pre-defined and one by one the software tasks are granted access to the processor for that time interval after which they are preempted and the next task is activated. Round-robin scheduling is probably the best method for guaranteeing fairness for all scheduled tasks.

Another scheduling method which is especially suitable for real-time systems is **earliest deadline first (EDF) scheduling**. Here, deadlines are assigned to all software tasks a priori. At runtime, the EDF scheduler keeps track of the activation times of all currently active or blocked tasks and executes the task whose deadline is the next to become due.

A third method on which this work focuses is **fixed-priority preemptive scheduling (FPS)**. A unique fixed priority is assigned to each task allocated to a given ECU a priori (at system design time). The scheduler ensures that always from the set of currently active tasks the one with the highest priority is granted access to the processor. While EDF scheduling has the advantage of being able to guarantee the deadlines of tasks even at high processor load, FPS scheduling is still widely used in industrial contexts which was the key factor for focusing on FPS scheduling in this thesis.

2.2.5. Fixed-Priority Preemptive Scheduling of ECUs

2.2.5.1. Deadline-Monotonic Priority Assignment

Using fixed-priority preemptive scheduling requires that a priority is assigned to each software task which is unique in the scope of the ECU where the task is allocated. A common approach is to assign deadlines rate-monotonic (where rate is a another term for activation period) such that of each two software tasks allocated to the same ECU the one with the smaller activation period gets the higher priority. One advantage of the rate-monotonic priority assignment is that priorities can be already assigned even if the task deadlines are not known yet. Rate-monotonic assignment of priorities has been proven to be optimal (under certain premises) if the deadlines of all software tasks are equal to their periods (see [BW01]).

The approach used in this thesis is to assign task priorities **deadline-monotonic**. Analogously to the rate-monotonic approach, of any two tasks allocated to the same ECU the task with the smaller deadline gets the higher priority. The deadline-monotonic priority assignment has been proven to be optimal if the following premises are valid:

- the deadlines of all tasks are smaller or equal to their activation periods.
- the worst-case execution times of all tasks are equal to or less than their deadlines
- all tasks are independent from each other (e.g. they do not block each other by using mutually exclusive resources)
- tasks do not suspend themselves
- there are no scheduling overheads
- tasks do not have a release jitter

The proof of optimality can be found in [BW01, p. 485].

2.2.5.2. Utilization-based Schedulability Tests

One of the earliest and most cited publication in the real-time research community is [LL73]. The authors provide a sufficient condition for the schedulability of a single ECU based on the ECU's utilization under certain premises. Those premises are:

- the activation of all software tasks is strictly periodic,
- the deadline of each software task is identical to its activation period, and
- the tasks are independent from each other.

The original paper also states that the "run-time for each task is constant for that task and does not vary with time." [LL73]. This premise can be relaxed to state that the runtime of each task never exceeds a certain maximal value (worst case execution time).

The utilization of an ECU with tasks $\tau_1, \ldots \tau_m$ is defined as:

$$U = \sum_{i=1}^{m} (\frac{C_i}{T_i})$$

where C_i denotes the worst case execution time of a task τ_i and T_i the task's activation period. Note that these notations and variable names are used as presented in the original paper for simplicity. In the following chapters, the notations are refined (see Chapter 3) to better match the requirements of this thesis.

The sufficient schedulability test of Liu and Layland states that if for an ECU with m tasks the condition $U \leq m(2^{\frac{1}{m}} - 1)$ holds, then this ECU is schedulable. The authors also state that the upper bound $m(2^{\frac{1}{m}} - 1) \simeq \ln 2 \approx 69\%$ for large values of m. Of course, an ECU with a higher utilization than 69% might be schedulable, too (as the schedulability test is sufficient but not necessary). But then other schedulability tests have to be applied for proofing the schedulability of that ECU.

The Liu and Layland schedulability test has been revisited in [DG00] where the authors first show that the original proof is incomplete and then complete and correct that proof.

The definition of software tasks used in this thesis requires that the deadline of each task is equal to or smaller than the task's activation period. Therefore the schedulability test of Liu and Layland is not applicable (because the premise that deadline and periods have to be equal is violated). However the software module responsible for the construction of the artificial benchmark used for evaluation (see Section 6.5.2 on page 149) makes use of this schedulability test to be able to construct feasible benchmark models.

2.2.5.3. Schedulability Analysis for Deadlines less or equal Periods

As stated above, the Liu and Layland schedulability test is not applicable if software tasks are having deadlines smaller than their activation periods. A schedulability test for systems where tasks have deadlines equal to or less than their activation periods has been proposed in [JP86] and (independently) in [Aud90]. Their approach is based on the concept of **critical instance**. An example of a critical instance is shown in Figure2.4. For the task model used throughout this thesis a critical instance on a given ECU is a (hypothetical) time instant where all tasks allocated to that ECU arrive (become ready for execution) simultaneously. Obviously, only one of those tasks can be granted access to the ECU at a time. In this case the task with the highest priority is processed first, as can be seen in Figure2.4 where task τ_1 has the highest priority on the ECU and is served first. In that figure task τ_2 arrives at the same time instant but is directly preempted. It remains preempted until task τ_1 finishes its execution. Then task τ_2 is executed until the higher priority task preempts its execution again. The critical instance describes a worst case scenario. If all tasks are schedulable in this scenario, they are schedulable in all other scenarios as well.

Based on the construction of the critical instance, the schedulability test validates that for every task the worst case response time is smaller than or equal to the task's deadline. The worst case response time r_i of a task τ_i can be calculated by solving

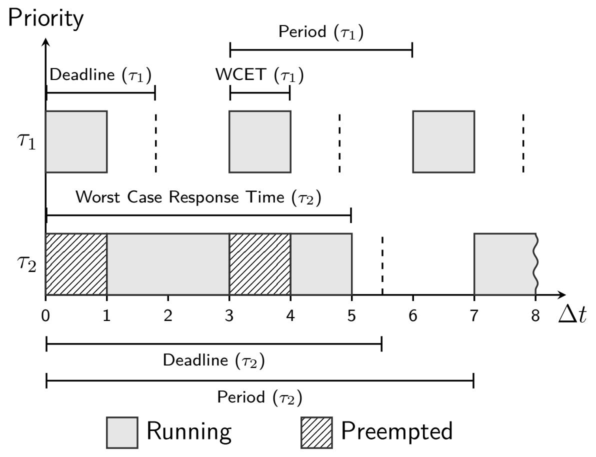


Figure 2.4. – Fixed-priority Preemptive Scheduling: Critical Instance

Fixed-point Equation (2.1). Equation (2.2) shows the corresponding schedulability test. The equations are provided here in the standard notation used in the real-time research community. In the following chapters the notion is further refined.

$$r_i = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j, \text{ where} \tag{2.1}$$

$$r_i \leq D_i \tag{2.2}$$

where r_i denotes the worst case response time of task τ_i, C_i the worst case execution time of task τ_i, T_j the activation period of a (higher-priority) task τ_j and D_i the deadline of task τ_i. In this and the following definitions, the function hp maps the index of every task to the set of indexes of all tasks allocated to the same ECU with a higher priority.

Computing the worst case response time can be done iteratively by starting with the worst case execution time of the task under consideration (Equation (2.3)) and using in all following iterations the result of the previous iteration for calculating the sum operator on the right-hand side (Equation (2.4)):

$$r_i^0 = C_i \tag{2.3}$$

$$r_i^{n+1} = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{2.4}$$

Note that the existence of a fixed-point is not guaranteed. As the fixed-point equation is monotonically increasing the iterative computation of the fixed-point equation will always terminate with one of the following results: In the first case a fixed-point is found which is smaller than or equal to the deadline. Formally: Task τ_i is schedulable if for an iteration $n \in \mathbb{N}$ it holds that

$$r_i^{n+1} = r_i^n \;\wedge\; r_i^n \leq D_i$$

Otherwise the task is not schedulable and the calculation is terminated as soon as an $n \in \mathbb{N}$ is reached such that r_i^n exceeds the task's deadline ($r_i^n > D_i$).

Equation (2.5) presents the fixed-point equation in a different notation more suitable for understanding first the nature of this fixed-point equation and second the findings presented in Chapter 5. In this notation the function ($\gamma_i(x)$) defined in Equation (2.5) equals to the right hand side of Equation (2.1). The fixed-point condition is specified separately as part of Equation (2.6). Note that γ_i does **not** compute the minimal fixed-point.

$$\gamma_i(x) = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{x}{T_j} \right\rceil C_j \tag{2.5}$$

The worst case response time (if existent) is the smallest $r_i \in \mathbb{N}$ such that

$$\gamma_i(r_i) = r_i \wedge r_i \leq D_i \tag{2.6}$$

If a worst case response time smaller than or equal to the deadline exists, it equals by definition to the smallest fixed-point of γ_i (for the definition of smallest fixed-point see Section A.4 in the appendix). In general, fixed-point equations may have more than one fixed-point.

Depending on its parameters it is possible that γ_i has no fixed-point, because all function values are too large, formally: $\forall x \in \mathbb{N} : \gamma_i(x) > x$). But if there is at least one fixed-point, the smallest fixed-point can be calculated as in Theorem A.5 on page 191.

2.2.6. Scheduling and Schedulability of TDMA-Based Buses

The mutual exclusive access to pure TDMA-based buses is organized by specifying a time interval called **communication cycle** or **TDMA round** which is further partitioned into a number of **time slots**. Some TDMA-based buses such as FlexRay are capable of providing a mixed-mode where the TDMA round is partitioned into a static segment and a dynamic segment. The static segment is then again partitioned into time slots while the dynamic segment is independently arbitrated via a priority-based protocol. Access to the time slots is arbitrated by specifying maximal one bus participant (connected ECU at design time which has exclusive access to that slot. There might be bus slots which are not assigned to any bus participant yet, e.g. as reserve for future extensions. To each bus participants multiple bus slots can be assigned. In this work, it is assumed for simplicity that for each message a bus participant has to send, one or more bus slots (of the set of bus slots assigned to the bus participant) are reserved exclusively. Schedulability analysis of pure TDMA-based (without a dynamic segment) involves the calculation of the worst-case response time of each message. This is done by identifying for every message the longest time interval its sending gateway has to wait for a bus slot reserved for that message. Adding the actual time required for transmitting the message to that time interval gives the worst-case response time of the message.

The analysis and optimization of TDMA buses is out of scope of this thesis due to the high complexity this would add to the approach. While the schedulability analysis of an existing configuration could be handled in principle, the calculation of a valid schedule is known to be NP-hard. An approach for FlexRay has been proposed in [Luk+09]. The linear program presented in this paper could be integrated in the approach for global analysis described in Chapter 3 but only at the cost of a huge increase in complexity.

2.2.7. Scheduling and Schedulability of Priority-Based Buses

The focus of this thesis is on priority-based buses such as Controller Area Networks (CAN). Schedulability analysis for those buses is very similar to schedulability analysis of ECUs. The main difference for CAN buses is that sending a message on the bus is non-preemptive. While during the arbitration phase messages with higher priority are preferred over messages with lower priority thus delaying the transmission of the lower-priority messages, no higher-priority messages can preempt the actual transmission of a lower-priority message. This has two consequences: Firstly, during the calculation of the worst-case response time for a message, the message's own worst case transmission time is not part of the fixed-point equation (because preemptions are no longer possible during the transmission). Secondly, lower-priority message can block higher-priority messages for a certain amount of time called **blocking time**. Both effects have been considered in the response time analysis.

In [TB94], [THW94] and [TBW95] the authors presented their approach for CAN bus schedulability analysis, which later was revised in [Dav+07]. The worst case response time for messages on CAN-buses can be calculated with Equation (2.7) which refers to fixed-point Equation (2.8). Note that in this thesis it is assumed that no release jitter occurs. Therefore the jitter has been removed from those equations (compare to [Dav+07, p. 251]).

$$r_i = w_i + C_i, \text{ with} \tag{2.7}$$

$$w_i = B_i + \sum_{k \in \mathrm{hp}(i)} \left\lceil \frac{w_i + \tau_{bit}}{T_k} \right\rceil C_k \tag{2.8}$$

$$B_i = \max_{k \in \mathrm{lep}(i)} \{C_k\} \tag{2.9}$$

$$r_i \leq D_i \tag{2.10}$$

where

- r_i is the worst case response time of message m_i,
- w_i is called the busy period of that message,

- C_i is the worst case transmission time of m_i,
- B_i is the maximal blocking time,
- function hp assigns to each message index the set of indexes of all higher-priority messages, and
- function lep assigns to each message index the set of indexes of all messages with lower or equal priorities,
- τ_{bit} the time required for transmitting one bit (required to avoid the trivial but wrong solution $w_i = 0$),
- T_i is the activation period of message m_i, and
- D_i is the deadline of message m_i.

Similar to the schedulability test for tasks the busy period w_i can be calculated iteratively using Equation (2.11) as starting point and calculating iterations with Equation (2.12) until either a fixed-point has been found for which the schedulability condition in Equation (2.13) is satisfied, or if no such $n \in \mathbb{N}$ can be found. In this case the message is not schedulable.

$$w_i^0 = B_i + \sum_{k \in \mathrm{hp}(i)} \left\lceil \frac{\tau_{bit}}{T_k} \right\rceil C_k \tag{2.11}$$

$$w_i^{n+1} = B_i + \sum_{k \in \mathrm{hp}(i)} \left\lceil \frac{w_i^n + \tau_{bit}}{T_k} \right\rceil C_k \tag{2.12}$$

$$\exists n \in \mathbb{N} : w_i^{n+1} = w_i^n \ \wedge \ w_i^n \leq (D_i - C_i) \tag{2.13}$$

2.3. Optimization Problems

The problems considered in this thesis are optimization problems. An optimization problem consists of a number of decision variables, a number of constraints describing valid valuations of the decision variables and one (sometimes even multiple) objective functions (also called cost functions) to be either minimized of maximized. A solution for an optimization problem is valuation of all decision variables satisfying all constraints. An optimal solution for a single objective problem is a solution having the best possible value for the objective function. Note that for a given optimization problem there may be no solution at all. The objective function might also be unbounded which means that there is no maximum/minimum. While for some optimization problems with multiple objectives those objectives can be aggregated to form a single objective function, there exists problems for which it is not possible. For such problems alternative notions for what an optimal solution is have to be used, e.g. Pareto optimality.

The formal definition of **optimization problem** given by Korte and Vygen in [KV10, p. 384] additionally states that the test whether or not a given solution is a valid solution (satisfying all constraints) for a given optimization problem must have polynomial-time complexity. The cost function used in an optimization problem must have polynomial-time complexity as well.

Of course the complexity of optimization problems can vary. For the simplest class of problems very efficient (deterministic) exact algorithms exist whose runtime can be bound by a polynomial in the number of decision variables. They are said to be in the class of deterministic polynomial-time problems P. However most optimization problems considered in this work belong to much harder-to-handle problem classes, mainly to the class of non-deterministic polynomial-time problems NP. For those problems no efficient exact algorithms are known. But several approaches exist which behave very well on most instances of real-world problems. On the one hand we have optimal algorithms and on the other hand heuristics.

One can distinguish between exact and heuristic approaches for solving optimization problems. Exact algorithms are guaranteed to find the best possible solution, but do not scale very well with increasing problem sizes for complex optimization problems, e.g. NP-hard optimization problems. Heuristic algorithms are not guaranteed to find the best possible solution but are usually significantly faster than exact algorithms.

2.3.1. Exact Optimization Methods

In this work, two different technologies have been applied for encoding the proposed exact optimization methods. Initially, SAT Modulo Theories solvers have been used. For the later works, mixed integer linear programming was applied.

2.3.1.1. SAT Modulo Theories (SMT)

SMT solvers combine the strengths of techniques for deciding satisfiability problems (SAT checking) with techniques specific to a theory module. In this work, a theory module providing linear arithmetic has been used. SMT solvers can decide whether or not a given SMT problem is feasible and can provide a solution if existing. Using SMT solvers for optimization can be done by iteratively running an SMT solver on the same SMT problem while tightening the constraints and thus performing a binary search. Because SMT solvers have only been evaluated initially in this work, foundations of SMT are not further discussed here. For detailed information about SMT in general and the SMT solver HySAT in particular see [FH07], [Her09] and [Her12].

2.3.1.2. Linear Programming

Linear programming is a technique for solving optimization problems consisting of real-valued variables and constraints given as linear inequations.

In [KV10, p. 49] the **linear programming** problem is defined as follows: A linear-programming instance is defined by specifying a matrix $A \in \mathbb{R}^{m \times n}$ and column vectors

$b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$. The task is to find a column vector $x \in \mathbb{R}^n$ such that $Ax \leq b$ and cx is maximum, decide that $\{x \in \mathbb{R}^n : Ax \leq b\}$ is empty, or decide that for all $\alpha \in \mathbb{R}$ there is an $x \in \mathbb{R}^n$ with $Ax \leq b$ and $cx > \alpha$.

Linear programming has been developed in the early 1940s by Leonid Vitalyevich Kantorovich but kept secret until first published by George Bernard Dantzig in 1947. Dantzig invented the Simplex algorithm which is still one of the most powerful approaches for solving linear programs and is implemented in all free and commercial linear-programming solvers used in this thesis (see [Dan63]). Simplex is not a polynomial-time algorithm for solving linear programs. While Simplex works pretty well for most real-world problems there exist problems for which the algorithm degrades. In 1979 Leonid Khachiyan proposed his Ellipsoid method and proofed that linear-programming problems can be solved in polynomial-time (see [Kha79]). However the practical relevance of the Ellipsoid method is negligible. The interior-point method (also called barrier methods) proposed by Narendra Karmarkar in 1984 has been shown to have polynomial-time, too (see [Kar84]). The interior-point algorithm is the most important alternative to Simplex and is also implemented by all free and commercial linear-programming solvers used in this thesis.

All free variables in a **linear program** have to be reals. In an **integer program** all variables x are required to be integers. In a **mixed integer linear program (MILP)** only some x_i are required to be integers while others are reals.

The following example demonstrates how mixed integer linear programming is used in the following chapters.

A factory has one machine which can produce one product out of two different product types each hour. A product of the first type requires 23 hours of production time and can be sold for € 10000. A product of the second type requires 42 hours of production and can be sold for € 15000. The machine is available for maximal 200 hours per month. Furthermore it is a requirement to produce at least one product of each type per month. What is the ideal mix of products for maximizing the total profit?

Encoding the example results in the following model:

$$\text{maximize } 10000x_1 + 15000x_2 \tag{2.14}$$

subject to:

$$23x_1 + 42x_2 \leq 200 \tag{2.15}$$

$$x_i \geq 1 \qquad \forall x_i \in \{x_1, x_2\} \tag{2.16}$$

$$x_1, x_2 \in \mathbb{N} \tag{2.17}$$

In Equation (2.17) two variables are declared, both natural numbers (and therefore integers). x_1 represents the number of products to produce of the first product type, x_2 the number of products to produce of the second product type. In the following chapters, all variables declared in this manner are free variables for which the linear-programming solver has to find a valid valuation. Note that in this example all free variables are integers which means that the formulation is an integer program. Each free variable corresponds to exactly one column in the matrix A (see above).

Each of the following constraints corresponds to one row in the matrix A. Equation (2.14) specifies the objective function which in this case is the sum of the number of products to be produced of each product type multiplied with the respective price. Equation (2.15) expresses the constraint that only a maximal 200 hours of production are available per month. In Equation (2.16) the constraint that at least one product has to be produced for each of the product types. This constraint has been added to the example to show how to describe many similar rows in a short hand notation by specifying on the right side of the equation that such a row has to be added to the linear program for each x_i.

A formulation of the example as a MathProg model can be found in the appendix in Section B.1 on page 193. Solving the example with a linear-programming solver (or by hand) gives the result that the maximal profit is € 80000 per month which can be achieved by producing 5 products of type 1 and 2 products of type 2 per month.

3. Problem Definition and Two-Tier Optimization Approach

This chapter provides a formal definition of the system model used throughout the remainder of this thesis and describes the two-tier optimization approach capable of solving the characterized problem first published in [Bük+11a] and [Bük+11b].

3.1. Formal System Model

In this section the hardware and the software design spaces are formally defined, followed by the definitions of a DSE problem, including constraint type classes. A formal definition of a system configuration for a given DSE problem is completed by several predicates formalizing constraints for the validity of configurations regarding the communication structure.

3.1.1. Formalization: Hardware Design Space

The hardware design space is defined by specifying a hardware architectural pattern (see Definition 3.2 on page 30). Before introducing the hardware design space notion, the concepts ECU, ECU-type, bus, bus-type, and the concept of a **hardware architecture tree** are formally defined.

3.1.1.1. Formalization: Logical ECU and ECUType

A (logical) ECU represents a computing resource in an architectural pattern (see also Section 2.1.7 on page 7). It has no further properties but can be typed by an ECU-type (see also Section 2.1.8 on page 7). The universe of ECU-types is denoted by $\mathbb{E}^\star$. The universe of ECUs is denoted by $E^\star$.

The size of the available memory (in kilobytes) of ECU-types is denoted by function

$$\text{mem}\colon \mathbb{E}^\star \to \mathbb{N}$$

and their hardware costs by

$$\text{cost}\colon \mathbb{E}^\star \to \mathbb{N}$$

For each ECU a set of allowed ECU-types is specified as part of the specification of the hardware architectural pattern (see Definition 3.2) thus restricting the choice during the optimization process to only those types. If the special ECU-type $\perp$ is assigned to an ECU then that ECU is not instantiated when creating an instance of the hardware architectural pattern. As a consequence, no software tasks may be allocated to that ECU. The hardware cost for ECU-type $\perp$ is defined as cost($\perp$):=0 and the memory size as mem($\perp$):=0.

3.1.1.2. Formalization: Logical Bus and Bus Type

A (logical) bus represents a communication component in a hardware architectural pattern (see also Section 2.1.9 on page 7). A bus itself has no further attributes but can be typed by a bus-type (see also Section 2.1.10 on page 7).

The universe of bus types is denoted by $\mathbb{B}^\star$. The universe of buses is denoted by $B^\star$. The bus class of bus types is denoted by function

$$\text{class}\colon \mathbb{B}^\star \to \{\texttt{PB}, \texttt{TB}\}$$

where PB stands for priority-based and TB stands for TDMA-based. The bandwidth in bytes per second is denoted by function

$$\text{bandwidth}\colon \mathbb{B}^\star \to \mathbb{N}$$

Additionally, a TDMA bus type consists of a list of **bus slots**. The set of all bus slots is denoted by *Slots**. The finite ordered set of bus slots for each bus is defined by function

$$\text{slots}\colon B^\star \to \mathcal{P}(\mathit{Slots}^\star)$$

The bus slot function is defined such that for each priority-based bus the set of assigned bus slots is empty and such that every bus slot belongs only to one bus (bus slots are never shared by multiple buses). Each bus slot has a defined **bus slot length**. The bus slot length (in seconds) of a given bus slot *slot* is denoted by

$$\text{length}\colon \mathit{Slots}^\star \to \mathbb{N}$$

The interconnection structure of an architectural pattern is specified as a **hardware architecture tree**. For reference, a definition of a "tree" can be found in [KV10, p. 7]: "Let G be some undirected graph. G is called connected if there is a v-w-path for all $v, w \in \mathrm{V}(G)$ [...]. An undirected graph without a circuit (as a subgraph) is called a forest. A connected forest is a tree." See [KV10] for details.

Definition 3.1 (Hardware Architecture Tree)
A hardware architecture tree is a tree

$$G^{\texttt{hw}} = \left(E \mathbin{\dot\cup} B, \mathit{Edges}^{\texttt{hw}}\right)$$

with set of vertices $E \mathbin{\dot\cup} B$ *and set of edges* $\mathit{Edges}^{\texttt{hw}}$ *where*

- $E \subset E^\star$ *is a finite set of ECUs*
- $B \subset B^\star$ *is a finite set of buses*

- *Edges*$^{\texttt{hw}} \subseteq \{\{e, b\} \mid e \in E,\, b \in B\}$ *is a set of undirected edges*
- *each ECU vertex has a degree of either* 1 *or* 2*, formally*

$$\forall e \in E : \deg(e) \in \{1, 2\}$$

  where function deg *maps to each vertex the number of connected edges*
- *each bus vertex has at least a degree of* 2*, formally*

$$\forall b \in B : \deg(b) \geq 2$$

- *there exists one "global" bus* $b^{\texttt{global}} \in B$ *to which every ECU with degree* 2 *is connected* □

The universe of hardware architecture trees is denoted by $G^{\texttt{hw}\star}$. Note that the definition of the edge set implies that every hardware architecture tree is a bipartite graph with respect to its ECU set E and bus set B.

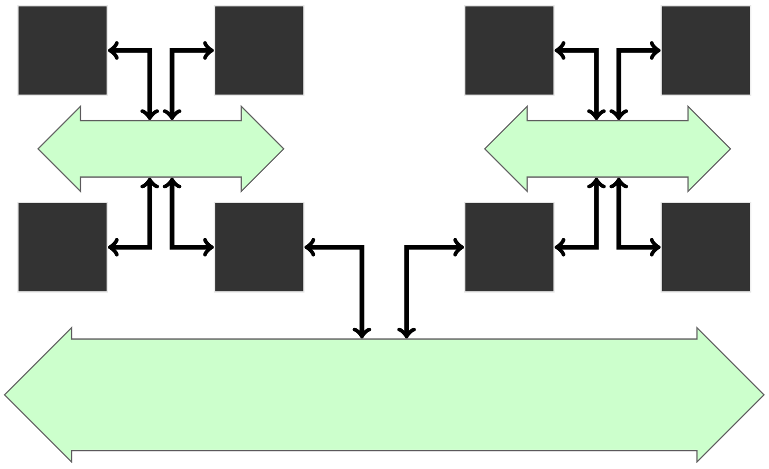


Figure 3.1. – Example: Hardware Architecture Tree

In Figure 3.1 a typical hardware architecture tree is shown. The large arrow-like vertices represent logical buses and the box-like vertices logical ECUs. Edges are represented by small black double headed arrows.

Let $G^{\texttt{hw}}$ be a hardware architecture tree with global bus $b^{\texttt{global}}$. Then every tree in the forest obtained by removing vertex $b^{\texttt{global}}$ from $G^{\texttt{hw}}$ is called a **hardware subsystem**. The set of subsystems of a hardware architecture tree $G^{\texttt{hw}}$ is denoted by subsysset($G^{\texttt{hw}}$). For convenience the function subsys($G^{\texttt{hw}}, v$) maps a given vertex (either an ECU or a bus except for the global bus) to the subsystem it is allocated to.

The function

$$\text{ecus}\colon G^{\texttt{hw}\star} \to \mathcal{P}(E^\star)$$

is defined such that it maps any given hardware architecture tree to the set of ECUs it contains. The function

$$\text{buses}\colon G^{\mathtt{hw}\star} \to \mathcal{P}(B^\star)$$

is defined such that it maps any given hardware architecture tree to the set of buses it contains.

It is required that the bus type used for the global bus connecting all hardware subsystems is TDMA-based (class is `TB`, e.g. FlexRay). Subsystems usually contain more than one ECU. A special case is supported where a subsystem contains no local bus and exactly one ECU which is directly connected to the global bus. One assumption of this work is that an existing embedded system is extended by adding new functionality. For the communication of the existing software functions over the global bus it is necessary to preserve the structure of the TDMA cycle, defined by the number, order and size of the bus slots. Therefore it is assumed here that the set of global bus slots $\text{slots}(b^{\mathtt{global}})$ is fixedly defined and remains unchanged during the optimization.

Following from Definition 3.1, all subsystems with more than one ECU contain exactly one local bus; each bus except for the global bus is part of exactly one subsystem. For simplicity, each local bus is required to use a priority-based bus type (class is `PB`, e.g. CAN).

Each ECU belongs to exactly one subsystem and is connected to the subsystem's local bus. In each subsystem the ECU which is connected to both the local and the global bus has the role of a **gateway ECU**. Gateway ECUs can be used for task allocation like every ECU in the system. But only software tasks directly allocated to a gateway ECU can directly send messages on the global bus. For software tasks not allocated to a gateway ECU, a relay service is used on the gateway ECU of their subsystem. Buses in subsystems are in the following also called (subsystem-) local buses.

The following definition formalizes the notion of hardware architecture used as input for the optimization process.

Definition 3.2 (Hardware Architectural Pattern)
A hardware architectural pattern is a tuple

$$hw = (G^{\mathtt{hw}}, \mathbb{E}, \mathbb{B}, \text{allowed}^E, \text{allowed}^B, slots^{\mathtt{avail}})$$

consisting of

- *a hardware architecture tree* $G^{\mathtt{hw}} = \left(E \mathbin{\dot\cup} B, Edges^{\mathtt{hw}}\right)$
- *a finite set of ECU-types* $\mathbb{E} \subset \mathbb{E}^\star$
- *a finite set of bus types* $\mathbb{B} \subset \mathbb{B}^\star$
- *a function assigning the set of allowed ECU-types to each ECU:*

$$\text{allowed}^E\colon \text{ecus}(G^{\mathtt{hw}}) \to \mathcal{P}(\mathbb{E})$$

- *a function assigning the set of allowed bus types to each bus:*

$$\text{allowed}^{B} : \text{buses}(G^{\texttt{hw}}) \to \mathcal{P}(\mathbb{B})$$

- *a finite set of available bus slots on the global bus* $b^{\texttt{global}}$

$$\mathit{slots}^{\texttt{avail}} \subseteq \text{slots}(b^{\texttt{global}})$$ □

Note that by excluding certain bus slots of the global bus from the subset of available bus slots $\mathit{slots}^{\texttt{avail}}$ on can ensure that those bus slots will not be used in any of the solutions of the optimization process. Those slots can thus be reserved, e.g. for extensions of the resulting system in the future.

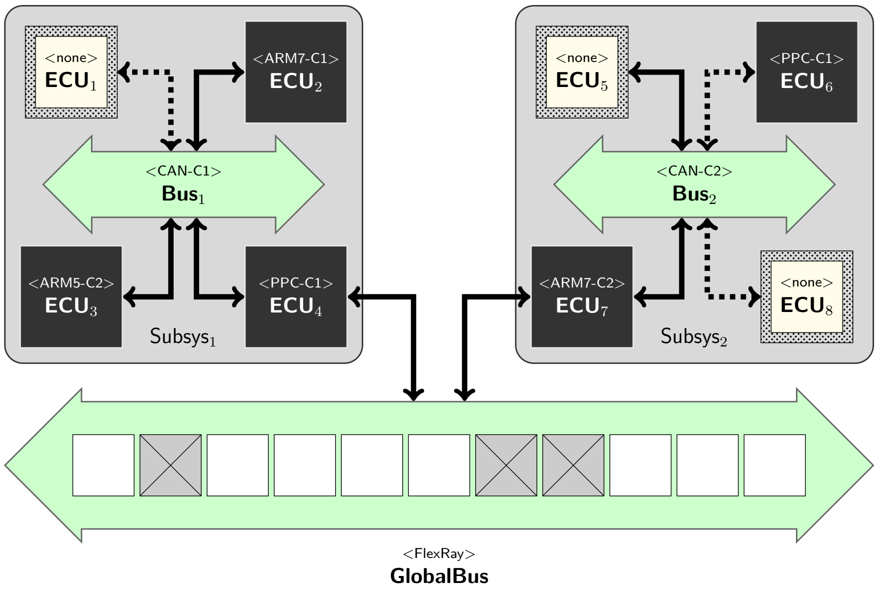


Figure 3.2. – Example Hardware Architectural Pattern (including initial ECU-type Function)

Figure 3.2 shows an example of a hardware architecture tree. Two hardware subsystems are connected via one global bus *GlobalBus* which is of type *FlexRay*. The global bus consists of eleven bus slots of which three are unavailable for messages. The global bus is connected to the gateway ECU of each of the subsystems. Each subsystem consists of four ECUs connected via one local bus. Note that two different bus types are used for the local buses, namely *CAN-C1* and *CAN-C2*. Both are priority-based buses but their respective bandwidth might differ. In the current configuration not every ECU is actually typed by one of the ECU-types, namely ECU_1, ECU_5 and ECU_8 are typed by $\perp$ (written as <*none*>) and will therefore not be instantiated when hardware architectural pattern is instantiating. All other ECUs are typed by existing ECU-types, among them two different configurations based on ARM7 processors *ARM7-C1* and *ARM7-C2*. Physical links between buses and ECUs are represented by the edges, where a solid line means that the link would be instantiated while a dashed line means that this link will not be instantiated. A link is instantiated only if the corresponding ECU is also instantiated.

3.1.2. Formalization: Task Network

3.1.2.1. Formalization: Software Task

In the context of this thesis a software task (denoted by τ) represents a functionality which is implemented by providing a source code implementation for one or multiple ECU-types which then is compiled specifically for this type (see also Section 2.1.12 on page 8). Software tasks appear as parts of a task network.

The universe of software tasks is denoted by $\mathcal{T}^\star$. The following task properties are explicitly used during the optimization process: The period is denoted by function

$$\text{period}^{\texttt{task}} \colon \mathcal{T}^\star \to \mathbb{N}$$

A local deadline is denoted by function

$$\text{deadline}^{\texttt{task}} \colon \mathcal{T}^\star \to \mathbb{N} \cup \{\infty\}$$

where $\text{deadline}^{\texttt{task}}(\tau) = \infty$ means that no deadline is specified for task τ (yet). The specification of a deadline is mandatory if a task is not part of task chain with an end-to-end deadline which could be used to synthesize a local deadline. Note that this local deadline is a property of the task in contrary to synthesized local deadlines for tasks which are calculated as part of a solution for a specific problem instance. Two limitations apply currently: Firstly, it is required that the deadline of each task is smaller or equal to its activation period. Secondly, it is required that only binary implementations are selected whose worst case execution time is smaller or equal to the task's activation period.

The universe of binary executables is denoted by $Bin^\star$. Each binary might be executable on several different ECU-types, depending on the instruction set of the used processors. For each ECU-type where a binary executable is executable, the worst case execution time and the maximal memory consumption could be determined.

The WCET of binaries on given ECU-types is denoted by partial function

$$\text{wcet}^{Bin^\star} : Bin^\star \times \mathbb{E}^\star \nrightarrow \mathbb{N}$$

and the memory consumption by partial function

$$\text{memreq}^{Bin^\star} : Bin^\star \times \mathbb{E}^\star \nrightarrow \mathbb{N}$$

3.1.2.2. Formalization: Signal

Data flow between software tasks is modeled explicitly using the concept of **signal** (see also Section 2.1.13 on page 11). While in general data flow between tasks could induce control flow as well, e.g. if a task is activated by the reception of a signal, this is not supported in this thesis. In this thesis it is assumed that each signal has exactly one **sender task** which is the source of the data flow and one or more **receiver tasks** which are the targets of the data flow.

The universe of signals is denoted by $S^\star$. The **size** in bytes of the data flow is denote by

$$\text{bytes}\colon S^\star \to \mathbb{N}$$

A signal inherits the activation period of its sender task. The (inherited) activation period is denoted by

$$\text{period}^{\texttt{sig}}\colon S^\star \to \mathbb{N}$$

A **deadline** can be specified optionally, denoted by

$$\text{deadline}^{\texttt{signal}}\colon S^\star \to \mathbb{N} \cup \{\infty\}$$

where $\text{deadline}^{\texttt{signal}}(s) = \infty$ means that no deadline has been specified for signal s. The specification of a deadline is mandatory if a signal is not part of task chain with an end-to-end deadline which could be used to synthesize a signal deadline. Note that this local deadline is a property of the signal in contrary to synthesized local deadlines for signals which are calculated as part of a solution for a specific problem instance.

For any given bus type $\mathfrak{b}$ the **worst case transmission time** (WCTT) of any signal s specified in seconds, is defined by function

$$\text{wctt}(s, \mathfrak{b}) := \left\lceil \frac{\text{bytes}(s)}{\text{bandwidth}(\mathfrak{b})} \right\rceil$$

The ceiling function is used to ensure that the transmission time is always a natural number.

3.1.2.3. Formalization: Message

A message is used whenever a signal has to be sent over a bus between multiple ECU (see also Section 2.1.14 on page 11). The universe of messages is denoted by $M^\star$.

Each message is allocated to exactly one bus either with a fixed priority (for CAN buses) or to one or more bus slots (for TDMA buses).

Each message belongs to exactly one signal. The relation is established as part of an allocation (see Definition 3.6 on page 38). The activation period of a given message m is derived from its signal s and denoted by $\text{period}^{\texttt{msg}}(m)$.

For any given bus type $\mathfrak{b}$ the **worst case transmission time** (WCTT) (unit: seconds) of any message m is equal to the

$$\text{wctt}(m, \mathfrak{b}) := \text{wctt}(s, \mathfrak{b})$$

where s is the signal to which the message m belongs.

Each message optionally has a **message deadline** denoted as

$$\text{deadline}^{\texttt{msg}}\colon M^\star \to \mathbb{N} \cup \{\infty\}$$

A value of $\text{deadline}^{\texttt{msg}}() = \infty$ means that no deadline has been specified for message m. Note that this local deadline is a property of the message in contrary to synthesized local deadlines for messages which are specified as part of a solution for a specific problem instance. If no deadline has been specified initially than an artificial deadline has to be synthesized during the optimization process for newly created messages.

The following example demonstrates the different situations with respect to signals and messages which might occur when allocating software tasks to ECUs. Let $\tau_1, \tau_2 \in \mathcal{T}^\star, \tau_1 \neq \tau_2$ be tasks and $s_1 \in S$ be a signal sent by task τ_1 and received only by τ_2 (not by any other task).

In the simplest case both tasks are allocated to the same ECU. Then the signal can be "sent" using operating system specific mechanisms, e.g. by writing the signal's data into a shared memory location and notifying the receiver task.

A more complex case occurs if the tasks are allocated to different ECUs in the same hardware subsystem. Then the signal has to be transmitted over the subsystem-local bus (which has to be priority-based in this thesis). A message is associated to the signal. The message's deadline is used to determine the message priority (following the deadline-monotonic paradigm). Assuming that the priority-based bus is a CAN-bus, a unique CAN object identifier (which is used to identify a message and for arbitration on CAN buses) is created for the message based on its priority and the priority of the other messages on that bus.

In another even more complex scenario the tasks are allocated to different ECUs which are in different hardware subsystems. In this scenario it is required to send a message over the global bus. The message is associated to the signal and assigned to one or more bus slots on the global bus. If the sender task is allocated to the gateway ECU of its subsystem, no additional steps are required in there. If it is allocated to a non-gateway ECU, a message on the local bus is required. This message is associated to the signal (in addition to the already associated message on the global bus). The message relay service of the subsystem gateway ECU is configured accordingly. For simplicity it is assumed that this service is implemented as part of the operating system on the gateway ECUs and has a negligible execution time (one which is very small compared to the usual execution times of tasks). For the receiver task the procedure is analogical. If the receiver task is allocated to the gateway ECU no additional actions are required. Otherwise a message on the local bus is necessary and a relay task on the gateway ECU.

Model for signal/message transmission Each task can send signals at the end of its execution. Each signal is sent using an operating system call which

- puts the signal in a shared memory location and notifies the local receiver tasks (if there are receiver tasks on the same ECU)

- encodes the corresponding message for the global or local bus and adds it to the transceivers send queue (if there is there are receiver tasks on different ECU)

If an ECU receives a message, it is copied to a buffer and all receiver tasks on that ECU are informed. However as in the context of this work all tasks are activated strictly periodically, the receiver tasks are not run immediately. At their next regular activation they receive the message by fetching it from the input buffer. Therefore a receiver task might read a message after it has been received by the ECU with a time delay potentially as large as the task's period, e.g. if a message is entering the receive buffer immediately after a task has (unsuccessfully) checked that buffer for new messages. In this case the message is read by the next occurrence of that task causing a message receive delay of a whole task period.

For each task all the execution delays due to operating system calls for receiving and sending messages are subsumed in the task's worst case execution time.

Software tasks and signals together form a forest of directed graphs. In this work it is mandatory that all of these directed task/signal graphs are cycle-free and therefore form trees called **task trees** (see Figure 3.3 for an example).

Definition 3.3 (Task Tree)
A task tree is a directed tree

$$G^{\mathtt{sw}} = (\mathcal{T} \,\dot{\cup}\, S, \mathit{Edges}^{\mathtt{sw}})$$

with a finite set of vertices $\mathcal{T} \,\dot{\cup}\, S$ and a finite set of edges $\mathit{Edges}^{\mathtt{sw}}$ with the following properties:

- *$\mathcal{T} \subset \mathcal{T}^{\star}$ is a finite set of tasks*
- *$S \subset S^{\star}$ is a finite set of signals*
- *$\mathit{Edges}^{\mathtt{sw}} \subseteq (\mathcal{T} \times S) \cup (S \times \mathcal{T})$ is a set of directed edges*
- *the root vertex (denoted by τ_{root}) is a task*
- *every leaf vertex is a task* □

Note that the way the edge set of a task tree is defined, implies that every task tree is bipartite with respect to its task set and its signal set. Note furthermore that since a task tree is a tree in the mathematical sense, the in-degree of the root vertex is 0 and the in-degree of every other vertex is 1. The universe of task trees is denoted by $\mathit{TREE}^{\star}$.

For a given task tree $\mathit{tree} \in \mathit{TREE}^{\star}$ the set of all paths of a given task tree which start at the root vertex and and end at one of its leaf vertices is denoted by

$$\begin{aligned}\text{paths}(\mathit{tree}) := \Big\{ (v_i, v_j) \;|\; & v_i, v_j \in \text{vertices}(\mathit{tree}) \\ & \wedge\ \deg^{-}(v_i) = 0 \\ & \wedge\ \deg^{+}(v_j) = 0 \Big\}\end{aligned}$$

where $\deg^{-}(v)$ denotes the in-degree of a vertex v of a directed tree (the number of incoming edges) and $\deg^{+}(v)$ the out-degree (the number of outgoing edges). Remark: For referring to an existing path of a task tree it is sufficient to specify the first and the last vertex due to the tree properties.

The function "paths" may also be applied on a set of task trees *Trees* in which case it is defined to be the union of all the maximal paths of task trees in the set, formally:

$$\text{paths}(\mathit{Trees}) := \bigcup_{\mathit{tree} \in \mathit{Trees}} \text{paths}(\mathit{tree})$$

The formal definition of task network as used in the context of this thesis is given in Definition 3.4 (see also Section 2.1.15 on page 11).

Definition 3.4 (Task Network)
A task network is a tuple

$$\mathit{tn} = (\mathit{Trees}, \text{choice}^{\texttt{exec}}, \text{deadline}^{\texttt{e2e}})$$

consisting of

- *a finite set of pairwise disjoint task trees Trees*
- *for E being the finite set of all ECUs in Trees, a partial function for choosing for each task one binary executable for some of the ECU-types:*

$$\text{choice}^{\texttt{exec}} : \mathcal{T} \times \mathbb{E}^{\star} \nrightarrow \mathit{Bin}^{\star}$$

- *a (partial) function for assigning end-to-end deadlines to some of the paths appearing in the set of task trees:*

$$\text{deadline}^{\texttt{e2e}} : \text{paths}(\mathit{Trees}) \nrightarrow \mathbb{N}$$ □

The universe of task networks is denoted by $\mathit{TN}^{\star}$. The set of tasks of a given task network tn is denoted by $\text{tasks}(\mathit{tn})$ or simply $\mathcal{T}$ if the corresponding task network is unambiguous. Analogously, the set of signals is denoted by $\text{signals}(\mathit{tn})$ or simply S.

Two functions are defined for convenience. For any given task network tn with task set $\mathcal{T}$, signal set S and edge set $\mathit{Edges}^{\texttt{sw}}$ function

$$\text{sender} : \mathit{TN}^{\star} \times S^{\star} \rightarrow \mathcal{T}^{\star} \text{, with}$$
$$\text{sender}(\mathit{tn}, s) := \begin{cases} \tau, & \text{if } s \in S \wedge \exists \tau \in \mathcal{T} \text{ such that } (s, \tau) \in \mathit{Edges}^{\texttt{sw}} \\ \perp, & \text{otherwise} \end{cases}$$

maps the sender task to each signal and function

$$\mathrm{recv}\colon TN^\star \times S^\star \to \mathcal{P}(\mathcal{T}^\star)$$
$$\mathrm{recv}(tn, s) := \{\tau \mid \tau \in \mathcal{T},\, (\tau, s) \in \mathit{Edges}^{\texttt{sw}}\}$$

maps to each signal the set of its receiver tasks.

Depending on the used optimization algorithms, certain restrictions on the acceptable input models are necessary. One restriction used throughout this work is that the activation periods for every task and every signal in the same task tree have to be equal, formally: Let *tree* be a task tree with set of ECUs E and set of signals S. *tree* is called **equal-period** if the activation periods of all tasks and signals are equal, formally:

$$\exists p \in \mathbb{N} : \forall e \in E : \mathrm{period}^{\texttt{task}}(e) = p \wedge \forall s \in S : \mathrm{period}^{\texttt{sig}}(s) = p$$

If for a task, signal or message the local deadline is missing, a deadline synthesis step is required. In such a step the end-to-end deadlines are split up into local deadlines for all path members which do not already have a local deadline assigned. During deadline synthesis the available fraction of the overall time budget (the time which is not already consumed by local deadlines of elements along the path) of each end-to-end deadline is distributed among the tasks/signals without a local deadline.

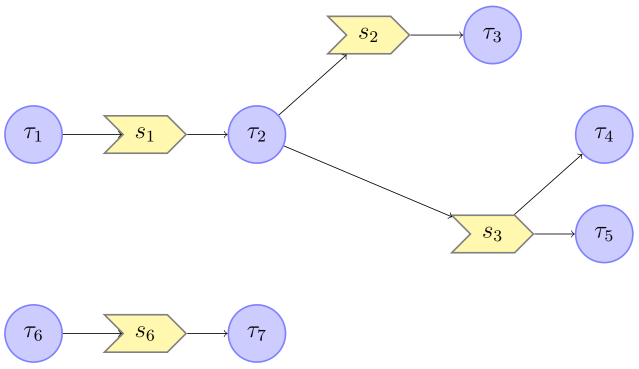


Figure 3.3. – Example Task Network with two Task Trees (5 Tasks/3 Signals and 2 Tasks/1 Signals, respectively)

3.1.3. ECU Type Function and Allocation

ECU-types are assigned to ECU by specifying an ECU-type function:

Definition 3.5 (ECU Type Function)
Let

$$hw = (G^{\texttt{hw}}, \mathbb{E}, \mathbb{B}, \mathrm{allowed}^E, \mathrm{allowed}^B, \mathit{slots}^{\texttt{avail}})$$

be an architectural pattern with set of ECUs E. An ECU-type function assigns a type to *each ECU, formally:*

$$\text{type}\colon E \to \mathbb{E} \cup \{\bot\} \qquad \square$$

where E is the set of ECUs as specified in the hardware architecture tree $G^{\mathtt{hw}}$.

The concept of **allocation** unites the allocation decisions for tasks, signals and messages in one tuple (see also Section 2.1.16 on page 11).

Definition 3.6 (Allocation)
Let tn be a task network consisting of the finite set of tasks $\mathcal{T}$, the finite set of signals S. Let M denote a finite set of messages. Let hw be an architectural pattern consisting of the finite set of ECUs E and the finite set of buses B. An allocation is a tuple

$$\mathcal{A} = (\text{taskalloc}, \text{msg}, \text{msgalloc})$$

consisting of

- *a (partial or total) function for allocating tasks to ECUs*

$$\text{taskalloc}\colon \mathcal{T} \nrightarrow E$$

- *a (total) function for allocating a (possibly empty) set of messages to each signal*

$$\text{msg}\colon S \to \mathcal{P}(M)$$

- *a (total) function for allocating each message to one bus*

$$\text{msgalloc}\colon M \to B \qquad \square$$

An allocation is called a **partial allocation** if some tasks are not allocated to ECUs and/or some signals which have to be sent over one or more buses due to the allocation of their sender/receiver tasks do not have message on these buses yet. Note that it is required that all messages which are relevant for the current DSE problem are allocated to buses (the message allocation function is a total function). If it is still undecided whether or not a given signal has to be transmitted via a bus or not, then that signal must have no message assigned (the set of assigned messages is empty). Whenever a message is assigned to a signal that message has to be allocated to a bus.

Figure 3.4 shows an allocation of tasks to ECUs and an ECU-type function. Signal s_6 is transmitted locally only (inside of one ECU). To all other signals one or more messages are assigned, which are allocated to buses of the hardware architecture. Message $m_{3,2}$ is transmitted on the global bus where it has been assigned to multiple bus slots.

3.1.4. Constraints

It is possible to specify constraints restricting the solution space.

Constraints are used to restrict how tasks may be allocated. For a set of tasks, it can be specified that all of the tasks have to be allocated to the same ECU, or in contrary that

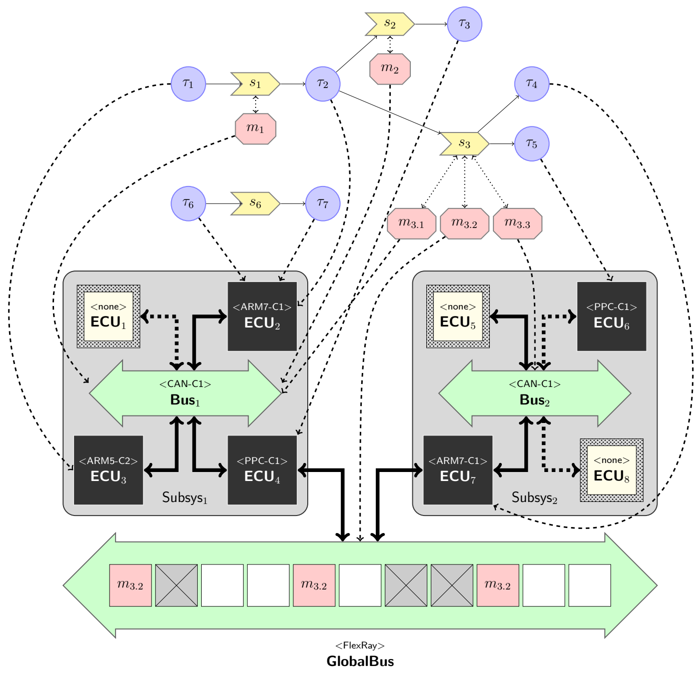


Figure 3.4. – Example Allocation

all of them have to be allocated to different ECUs. Such constraint types are necessary, if e.g. safety considerations require certain redundancy concepts to be applied.

Note that all constraints are formulated problem specific. That means they relate to a specific task network and a specific hardware architectural pattern. For simplicity it is assumed in the next paragraphs that such a specific problem has been defined and that all used sets (e.g. the set of ECUs E) belong to that problem specification.

3.1.4.1. Constraint-Type: Allowed ECUs per Task

For some tasks it might be necessary to restrict to which ECUs they may be allocated, e.g. if a task reads a certain sensor value of a sensor connected to a specific ECU.

Definition 3.7 (Allowed-ECUs-per-Task Constraint)
Let $\mathcal{T}$ be the set of tasks of a given task network and E the set of ECUs of a given hardware architectural pattern. A function

$$\psi^{\mathsf{ECUs}} : \mathcal{T} \to \mathcal{P}(E)$$

specifies for each task the set of ECUs to which the task may be allocated. A (partial) task allocation function taskalloc *of a given allocation $\mathcal{A}$ satisfies such a constraint if*

$$\forall \tau \in \mathrm{dom}(\mathrm{taskalloc}) : \mathrm{taskalloc}(\tau) \in \psi^{\mathsf{ECUs}}(\tau) \qquad \square$$

3.1.4.2. Constraint-Type: Allowed ECU Types per Task

For some tasks it might be necessary to restrict to which ECU-types they may be allocated.

Definition 3.8 (Allowed-ECU-Types-per-Task Constraint)
Let $\mathcal{T}$ be the set of tasks of a given task network and $\mathbb{E}$ the set of ECU-types of a given hardware architectural pattern. A function

$$\psi^{\mathtt{types}} : \mathcal{T} \to \mathcal{P}(\mathbb{E})$$

specifies for each task the set of ECU-types on which the task may be allocated. A (partial) task allocation function taskalloc *and an ECU-type function* type *satisfy such a constraint if*

$$\forall \tau \in \mathrm{dom}(\mathrm{taskalloc}) : \mathrm{type}(\mathrm{taskalloc}(\tau)) \in \psi^{\mathtt{types}}(\tau) \qquad \square$$

This constraint is also used in cases where no information is available regarding the WCET or memory consumption for a task on an ECU-type. In this thesis it is assumed that in this case the respective ECU-type is not part of the set of allowed ECU-types specified for that task.

3.1.4.3. Constraint Type: Never on same ECU

Especially for establishing redundancy requirements, e.g. due to safety considerations, this constraint type can be used to specify a set of tasks of which every task has to be allocated to a separate ECU.

Definition 3.9 (Never-On-Same-ECU Constraint)
Let $\mathcal{T}$ be the set of tasks of a given task network. Then this constraint is defined by specifying a finite set

$$\psi^{\mathtt{never}} \subseteq \mathcal{P}(\mathcal{T})$$
$$\forall X \in \psi^{\mathtt{never}} : |X| \geq 2$$

where each element is a set of tasks of which every task has to be allocated to separate ECU. For a given allocation $\mathcal{A}$, a (partial) task allocation function taskalloc *satisfies such a constraint if*

$$\forall X \in \psi^{\mathtt{never}}, \forall \tau_i, \tau_j \in (X \cap \mathrm{dom}(\mathrm{taskalloc})), \tau_i \neq \tau_j : \\ \mathrm{taskalloc}(\tau_i) \neq \mathrm{taskalloc}(\tau_j) \qquad \square$$

3.1.4.4. Constraint Type: Always on same ECU

Sometimes it can be helpful to force a set of tasks to be allocated onto the same ECU. The following constraint type can be used for specifying sets of tasks which always must be allocated to the same ECU.

Definition 3.10 (Constraint Type: Always-On-Same-ECU)
Let $\mathcal{T}$ be the set of tasks of a given task network. Then this constraint is defined by specifying a finite set

$$\psi^{\texttt{always}} \subseteq \mathcal{P}(\mathcal{T})$$
$$\forall X \in \psi^{\texttt{always}} : |X| \geq 2$$

where each element is a set of tasks which must be allocated to the same ECU. For a given allocation $\mathcal{A}$, a given (partial) task allocation function taskalloc *satisfies such a constraint if*

$$\begin{aligned}&\forall X \in \psi^{\texttt{always}} : \exists e \in E \textit{ such that}\\ &\ \forall \tau \in (X \cap \mathrm{dom}(\mathrm{taskalloc})) : \mathrm{taskalloc}(\tau) = e\end{aligned} \qquad \square$$

Note that the satisfaction condition explicitly ignores tasks which have not been allocated yet. Otherwise an initial incomplete allocation could possibly violate the constraints which is undesirable during the optimization process.

3.1.4.5. Constraint Type: Allowed Subsystems

It may also be required to specify for some tasks a set of allowed subsystems.

Definition 3.11 (Constraint Type: Allowed-Subsystems)
Let $\mathcal{T}$ be the set of tasks of a given task network, $G^{\texttt{hw}}$ a hardware architecture graph consisting of a set of hardware subsystems Sub. Then this constraint is defined by specifying a function

$$\psi^{\texttt{allowedSubsys}} : \mathcal{T} \to \mathcal{P}(Sub)$$

where to each task a set of allowed subsystems is assigned. For a given allocation $\mathcal{A}$, a given (partial) task allocation function taskalloc *satisfies such a constraint if*

$$\begin{aligned}&\forall \tau \in \mathrm{dom}(\mathrm{taskalloc}) :\\ &\mathrm{taskalloc}(\tau) \in \Big\{ e \,|\, e \in E : \mathrm{subsys}(G^{\texttt{hw}}, e) \in \psi^{\texttt{allowedSubsys}}(\tau) \Big\}\end{aligned} \qquad \square$$

3.1.4.6. Constraint Type: Always in Same Subsystem

Sometimes it can be helpful to force a set of tasks to be allocated onto the same hardware subsystems. The following constraint type can be used for that.

Definition 3.12 (Constraint Type: Always-In-Same-Subsystem)
Let $\mathcal{T}$ be the set of tasks of a given task network, $G^{\texttt{hw}}$ a hardware architecture graph consisting of a set of hardware subsystems Sub. Then this constraint is defined by specifying a finite set

$$\psi^{\texttt{sameSubsys}} \subseteq \mathcal{P}(\mathcal{T})$$
$$\forall X \in \psi^{\texttt{sameSubsys}} : |X| \geq 2$$

where each element is a set of tasks which must be allocated to the same subsystem. The (partial) task allocation function taskalloc *of a given allocation $\mathcal{A}$ satisfies such a constraint if*

$$\begin{aligned}&\forall X \in \psi^{\texttt{always}} : \exists sub \in \mathrm{subsysset}(G^{\texttt{hw}}) \textit{ such that}\\ &\quad \forall \tau \in (X \cap \mathrm{dom}(\mathrm{taskalloc})) : \mathrm{subsys}(G^{\texttt{hw}}, \mathrm{taskalloc}(\tau)) = sub\end{aligned} \qquad \square$$

3.1.4.7. Constraint Type: Never in Same Subsystem

Sometimes it can be helpful to forbid that certain tasks are allocated to the same hardware subsystem. The following constraint type can be used for that.

Definition 3.13 (Constraint Type: Never-In-Same-Subsystem)
Let $\mathcal{T}$ be the set of tasks of a given task network, $G^{\texttt{hw}}$ a hardware architecture graph consisting of a set of hardware subsystems Sub. Then this constraint is defined by specifying a finite set

$$\begin{aligned}\psi^{\texttt{diffSubsys}} &\subseteq \mathcal{P}(\mathcal{T}) \textit{ with}\\ \forall X \in \psi^{\texttt{diffSubsys}} &: |X| \geq 2\end{aligned}$$

where each element is a set of tasks with the following semantics: All tasks have to be allocated in such a way that every two (unequal) tasks are on different subsystems. E.g. for a set of three tasks three different subsystems are required, one for each task. The (partial) task allocation function taskalloc *of a given allocation $\mathcal{A}$ satisfies such a constraint if*

$$\begin{aligned}&\forall X \in \psi^{\texttt{always}} :\\ &\quad \forall \tau_i, \tau_j \in (X \cap \mathrm{dom}(\mathrm{taskalloc})), \tau_i \neq \tau_j :\\ &\quad \mathrm{subsys}(G^{\texttt{hw}}, \mathrm{taskalloc}(\tau_i)) \neq \mathrm{subsys}(G^{\texttt{hw}}, \mathrm{taskalloc}(\tau_j))\end{aligned} \qquad \square$$

3.1.4.8. Constraint Type: Always on Bus

Some signals might be required to be allocated to one or more buses (global or local). The following constraint type allows one to specify for each bus the set of signals which always must be allocated to that bus.

Definition 3.14 (Constraint Type: Always-on-Bus)
Let $S \subset S^\star$ be the finite set of signals of a given task network and $B \subset B^\star$ be the finite set of buses of a given hardware architectural pattern. Then this constraint is defined by specifying a function

$$\psi^{\texttt{onBus}} : B \to \mathcal{P}(S)$$

where each bus is mapped to the set of signals which must be allocated onto it in any case. For a given allocation $\mathcal{A}$ with a given signal to message allocation function msg *and a* *corresponding message allocation function* msgalloc *satisfy such a constraint if*

$$\forall b \in B, \forall s \in \psi^{\texttt{onBus}}(b) : \exists m \in \text{msg}(s) \textit{ with } \text{msgalloc}(m) = b \qquad \square$$

All supported constraint types are combined to form a constraint specification.

Definition 3.15 (Constraint Specification)
A constraint specification is a tuple

$$constr = (\psi^{\texttt{ECUs}}, \psi^{\texttt{types}}, \psi^{\texttt{never}}.\psi^{\texttt{always}}, \psi^{\texttt{allowedSubsys}}, \psi^{\texttt{sameSubsys}}, \psi^{\texttt{diffSubsys}}, \psi^{\texttt{onBus}})$$

where

- *function* $\psi^{\texttt{ECUs}}$ *specifies the allowed ECUs for each task*
- *function* $\psi^{\texttt{types}}$ *specified the allowed ECU-types for each task*
- *the finite set* $\psi^{\texttt{never}}$ *specifies tasks which may never be allocated to the same ECU*
- *the finite set* $\psi^{\texttt{always}}$ *specifies tasks which always have to be allocated to the same ECU*
- *function* $\psi^{\texttt{allowedSubsys}}$ *specifies for each task the set of allowed subsystems*
- *the finite set* $\psi^{\texttt{sameSubsys}}$ *specifies sets of tasks which always must be allocated to the same subsystem*
- *the finite set* $\psi^{\texttt{diffSubsys}}$ *specifies sets of tasks which must never be allocated to the same subsystem*
- *function* $\psi^{\texttt{onBus}}$ *specifies for each bus a set of signals which must be allocated via messages to that bus in any case* $\square$

3.1.5. DSE Problem

The term **design space exploration problem** is used in this thesis to describe one problem instance.

Definition 3.16 (DSE Problem)
A design space exploration (DSE) problem is a tuple

$$p = (tn, hw, alloc^{\texttt{initial}}, constr)$$

where

- *tn is a task network*
- *hw is a hardware architectural pattern*

- *$alloc^{\texttt{initial}}$ is an initial (partial) allocation*
- *$constr$ is a constraint tuple.*

Initially all constraints must be satisfied. □

Let

$$p = (tn, hw, alloc^{\texttt{initial}}, constr)$$

be a DSE problem with a task network

$$tn = \left(\mathit{Trees}, \text{choice}^{\mathsf{exec}}, \text{deadline}^{\mathsf{e2e}}\right)$$

with $\mathcal{T}$ being the finite set of tasks.

The following two derived functions map one value for the worst case execution time (resp. memory consumption) to each task for each of the ECU-types:

$$\begin{aligned}
&\forall \tau \in \mathcal{T}, \forall t \in \mathbb{E}^\star : \\
\text{wcet}_p(\tau, t) &:= \begin{cases} \text{wcet}^{Bin^\star}(\text{choice}^{\mathsf{exec}}(\tau, t), t), & \text{if choice}^{\mathsf{exec}}(\tau, t) \text{ is defined} \\ 0, & \text{otherwise} \end{cases} \\
\text{memreq}_p(\tau, t) &:= \begin{cases} \text{memreq}^{Bin^\star}(\text{choice}^{\mathsf{exec}}(\tau, t), t), & \text{if choice}^{\mathsf{exec}}(\tau, t) \text{ is defined} \\ 0, & \text{otherwise} \end{cases}
\end{aligned}$$

Note that the value of WCET function $\text{wcet}^{Bin^\star}(\text{choice}^{\mathsf{exec}}(\tau, t), t)$ is mapped to 0 if no information about the worst case execution time exists. This is done to make the handling with this function easier in the following chapters. It is required to add additional constraints for guaranteeing that a task is never allocated to an ECU which has been assigned an ECU-type for which no worst case execution time is known for that task.

3.1.6. Configuration and Solution

The optimization process can modify different aspects of the model which has to be optimized. Firstly, it can choose the used ECU-type for each ECU which directly affects the optimization objective which is defined as the sum of ECU-type costs.

Secondly all initially unallocated software tasks have to be allocated to an ECU. Finally — depending on the allocation of the tasks —for some of the signals representing the communication between software tasks message have to be created which are then allocated to communication buses.

In this thesis a **configuration** denotes one intermediate or final solution of a given DSE Problem.

Definition 3.17 (Configuration)
Let $p = (tn, hw, alloc^{\texttt{initial}}, constr)$ be a DSE problem with task network

$$tn = (\mathit{Trees}, \text{choice}^{\mathsf{exec}}, \text{deadline}^{\mathsf{e2e}})$$

and hardware architectural pattern

$$hw = (G^{\texttt{hw}}, \mathbb{E}, \mathbb{B}, \mathrm{allowed}^{E}, \mathrm{allowed}^{B}, \mathit{slots}^{\texttt{avail}})$$

A configuration for DSE problem p is a tuple

$$\mathit{conf} = (\mathrm{type}, \mathcal{A}, \mathrm{deadline}^{\texttt{task}}, \mathrm{deadline}^{\texttt{signal}}, \mathrm{deadline}^{\texttt{msg}})$$

where

- type *is an ECU-type function*
- $\mathcal{A} = (\mathrm{taskalloc}, \mathrm{msg}, \mathrm{msgalloc})$ *is an allocation*
- *(partial) function* $\mathrm{deadline}^{\texttt{task}}$ *assigns local deadlines to tasks (Definition 3.4)*
- *(partial) function* $\mathrm{deadline}^{\texttt{signal}}$ *assigns local deadlines to signals (Definition 3.4)*
- *(partial) function* $\mathrm{deadline}^{\texttt{msg}}$ *assigns local deadlines to messages (Definition 3.4)*$\square$

The function

$$\mathrm{subsys}^{\mathcal{T}} \colon (\mathcal{P}^{\star} \times \mathit{CONF}^{\star} \times \mathcal{T}^{\star}) \nrightarrow S^{\star}$$

maps each for a given DSE problem every task allocated by a given configuration to the subsystem it has been allocated to.

Not every configuration is valid with respect to the implementability of the system. Several requirements have to be satisfied which are mentioned in the definition of validity of configurations (see Definition 3.18 on page 50).

In order to formalize the conditions under which a configuration satisfies all implicit requirements regarding the communication via signals/messages, two predicates are defined.

3.1.6.1. Message Predicates

For the definition of the following predicates let p be a DSE problem and *conf* a configuration for that problem.

Messages required for any two tasks Let $\tau_1, \tau_2 \in \mathcal{T}, \tau_1 \neq \tau_2$ be tasks and s_1 be signal sent from τ_1 to τ_2. The allocation of the tasks determines how many messages are required for implementing the data flow. The following cases are to be distinguished:

1. τ_1 and τ_2 are allocated to the same ECU: No message is required
2. τ_1 and τ_2 are allocated to different ECUs located in the same subsystem: One message on the local bus of that subsystem is required
3. τ_1 is on the gateway ECU of a subsystem sub_1 and τ_2 is on the gateway ECU of a subsystem sub_2 and $\mathit{sub}_1 \neq \mathit{sub}_2$: Only one message on the global bus is required

4. τ_1 is not on the gateway ECU of a subsystem sub_1 and τ_2 is on the gateway ECU of a subsystem sub_2 and $sub_1 \neq sub_2$: One message on the global bus and one message on the local bus of subsystem sub_1 are required
5. τ_1 is on the gateway ECU of a subsystem sub_1 and τ_2 is not on the gateway ECU of a subsystem sub_2 and $sub_1 \neq sub_2$: One message on the global bus and one message on the local bus of subsystem sub_2 are required
6. τ_1 is not on the gateway ECU of a subsystem sub_1 and τ_2 is not on the gateway ECU of a subsystem sub_2 and $sub_1 \neq sub_2$: One message on the global bus and one message on each of the local buses of subsystem sub_1 and sub_2 are required.

Task	**Same Subsystem**		**Different Subsystems**			
Allocation	**Same ECU**	**Different ECUs**	**Sender Task on GW**		**Receiver Task on GW**	
			Yes	**No**	**Yes**	**No**
Global bus	–	–	X	X	X	X
Bus in subsystem of sender task	–	X	–	X		
Bus in subsystem of receiver task	–	X			–	X

Table 3.1. – Message Requirements for Tasks

Based on these considerations, appropriate predicates have been formulated.

Global Message A given signal must have a message on the global bus

a) if the user specified a constraint to enforce this, or

b) if the user specified a constraint to enforce that signal to be allocated to a local bus in a subsystem different from the sender task's subsystem, or

c) if at least one receiver task is allocated to a different subsystem than the sender task

These conditions are formalized in predicate needsGlobal which evaluates to true for a signal iff a corresponding message is required on the global bus. Let p be a DSE problem, *conf* a configuration for that DSE problem as defined above, s be a signal used in that DSE problem.

A given configuration *conf* for a DSE problem p with an allocation $\mathcal{A}$ complies to the global message condition iff

$$\forall s \in S : (\text{needsGlobal}(p, conf, s) = \texttt{true}) \Leftrightarrow$$
$$\exists m \in \text{msg}_{\mathcal{A}}(s) : \text{msgalloc}_{\mathcal{A}}(m) = b^{\texttt{global}}$$

where $\text{msg}_{\mathcal{A}}$ is the signal allocation function of the configuration *conf* and $b^{\texttt{global}}$ is the global bus. The predicate needsGlobal$(p, conf, s)$ is defined as follows:

$$\text{needsGlobal}(p, \mathit{conf}, s) \ := \tag{3.1}$$
$$\begin{cases} \texttt{true}, & \text{if } s \in \psi^{\texttt{onBus}}(b^{\texttt{global}}) \\ \texttt{true}, & \text{or if } \exists b \in B \text{ with } \text{subsys}(G^{\texttt{hw}}, b) \neq \mathit{sub}_{\tau^{\texttt{send}}} \wedge s \in \psi^{\texttt{onBus}}(b) \\ \texttt{true}, & \text{or if } \exists \tau^{\texttt{recv}} \in \text{recv}(tn, s) : \text{subsys}^{\mathcal{T}}(p, \mathit{conf}, \tau^{\texttt{recv}}) \neq \\ & \qquad \mathit{sub}_{\tau^{\texttt{send}}} \\ \texttt{false}, & \text{otherwise} \end{cases}$$

where $G^{\texttt{hw}}$ is a given hardware architecture graph, $\tau^{\texttt{recv}}$ denotes a receiver task, $\tau^{\texttt{send}}$ the sender task, and $\mathit{sub}_{\tau^{\texttt{send}}}$ denotes the subsystem of the ECU to which the sender task is allocated (assuming that the task is allocated).

Let S denote the set of signals of a given DSE problem p and a given configuration *conf* for that problem. Using the (partial) allocation which is part of the configuration, this set can be divided into three disjoint partitions:

$$S = S^{\texttt{always}}_{\mathit{conf}} \mathbin{\dot\cup} S^{\texttt{never}}_{\mathit{conf}} \mathbin{\dot\cup} S^{\texttt{maybe}}_{\mathit{conf}}$$

The set of signals for which a message has to be allocated onto the global bus in any case, as required by the predicate in Definition (3.1) is defined as:

$$S^{\texttt{always}}_{\mathit{conf}} := \{s \mid s \in S : \text{needsGlobal}(p, \mathit{conf}, s) = \texttt{true}\} \tag{3.2}$$

The set of signals where for each signal sender task and all receiver tasks are already allocated in such a way that no message on the global bus is required (all tasks are inside of the same subsystem) is defined as:

$$\begin{aligned} S^{\texttt{never}}_{\mathit{conf}} := \{ & s \mid s \in S : \text{needsGlobal}(p, \mathit{conf}, s) = \texttt{false} \wedge \\ & \text{sender}(tn, s) \in \text{dom}(\text{taskalloc}) \wedge \\ & \text{recv}(tn, s) \subseteq \text{dom}(\text{taskalloc}) \end{aligned} \tag{3.3}$$

All the signals not belonging to one of the aforementioned partitions are in the set of signals for which no definitive decision has been taken yet. For them there is no need to acquire a global message for the given partial configuration. But that need may arise during the following analysis steps, e.g. if the sender task or at least of the receiver tasks of a signal is currently unallocated and is then allocated to a different subsystem than the other tasks sending/receiving that signal. This set is defined as:

$$S^{\texttt{maybe}}_{\mathit{conf}} := S \setminus \left(S^{\texttt{always}}_{\mathit{conf}} \cup S^{\texttt{never}}_{\mathit{conf}} \right) \tag{3.4}$$

Local Message In Subsystem A signal must correspond to a message in a given subsystem *sub* (but only if that subsystem has a local bus)

a) by request of the user, or

b) if the sender and at least one receiver are on different ECUs in that subsystem

c) if there is a global message for that signal and either the sender task or one receiver tasks are on a non-gateway ECU in subsystem *sub*

The conditions explicitly exclude the situation where there is a message on the global bus such that all tasks involved in either receiving or sending the signal in the given subsystem are on gateway ECUs. In this special case the tasks have direct access to the global bus and no communication on the subsystem-local bus is required.

The conditions can be formalized as follows with p being a DSE problem, $conf$ a configuration with allocation $\mathcal{A}$, s a signal contained in the set of signals of the DSE problem and a subsystem sub of the DSE problem:

$$\text{needsLocal}(p, conf, s, sub) \coloneqq \tag{3.5}$$
$$\begin{cases} \texttt{true}, & \text{if } s \in \psi^{\text{onBus}}(b_{sub}) \\ \texttt{true}, & \text{or if } \exists \tau \in (\text{recv}(tn, s) \cup \{\text{sender}(tn, s)\}) : \\ & sub_\tau = sub \ \wedge \ \neg\text{isGw}(e_\tau) \wedge \\ & \text{needsGlobal}(p, conf, s) = \texttt{true} \\ \texttt{true}, & \text{or if } \exists \tau^{recv} \in \text{recv}(tn, s) : \\ & \quad \text{subsys}^{\mathcal{T}}(p, conf, \tau^{send}) = \text{subsys}^{\mathcal{T}}(p, conf, \tau^{recv}) \wedge \\ & \quad \text{taskalloc}(\tau^{send}) \neq \text{taskalloc}(\tau^{recv}) \\ \texttt{false}, & \text{otherwise} \end{cases}$$

where b_{sub} is the local bus in subsystem sub, τ^{send} is the (unique) sender task of signal s, and e_τ denotes the ECU to which the task τ is allocated without loss of generality.

A given configuration $conf$ with allocation $\mathcal{A}$ complies to the local message condition iff

$$\forall s \in S, \forall sub \in Sub : (\text{needsLocal}(p, conf, s, sub) = \texttt{true}) \Leftrightarrow$$
$$\exists m \in \text{msg}_{\mathcal{A}}(s) : \text{msgalloc}_{\mathcal{A}}(m) = b_{sub}$$

where b_{sub} is the local bus of subsystem sub.

3.1.6.2. Messages Between Two Tasks

Each signal can be received by multiple tasks. Therefore, while a whole set of messages might be associated to a given signal, only a subset of them might be required to transmit the data from the signal's sender task to one of its receiver tasks.

For example, one receiver of a given signal might be allocated to the same ECU as the sender and therefore does not require any message. But another receiver of the same signal might be allocated to a different ECU inside of the same subsystem. For this second task, a message on the subsystem-local bus is required.

Let $p = (tn, hw, alloc^{\texttt{initial}}, constr)$ be a DSE problem with set of tasks $\mathcal{T}$, set of signals S and set of messages M. Let furthermore be

$$conf = (\text{type}, \mathcal{A}, \text{deadline}^{\texttt{task}}_{conf}, \text{deadline}^{\texttt{signal}}_{conf}, \text{deadline}^{\texttt{msg}}_{conf})$$

be a configuration for that problem with allocation $\mathcal{A} = (\text{taskalloc}, \text{msg}_{\mathcal{A}}, \text{msgalloc}_{\mathcal{A}})$. Let $G^{\texttt{hw}}$ be the corresponding architecture graph of the hardware architectural pattern.

Let $\tau^{\texttt{send}}_s \in \mathcal{T}$ be a task sending signal $s \in S$ and $\tau^{\texttt{recv}} \in \mathcal{T}$ a task receiving that signal. For better readability of the following function, some abbreviations are used for substituting the respective functions. Let

- $e^{\texttt{send}}{:=}\text{taskalloc}(\tau^{\texttt{send}}_s)$ be that ECU to which task $\tau^{\texttt{send}}_s$ is allocated.
- $sub^{\texttt{send}}{:=}\text{subsys}(G^{\texttt{hw}}, e^{\texttt{send}})$ the subsystem of ECU $e^{\texttt{send}}$
- $b^{\texttt{send}} \in B$ be the local bus connected to $e^{\texttt{send}}$
- $e^{\texttt{recv}}{:=}\text{taskalloc}_{G^{\texttt{hw}}}(\tau^{\texttt{recv}})$ be the ECU to which task $\tau^{\texttt{recv}}$ is allocated.
- $b^{\texttt{recv}}$ be the local bus connected to ECU $e^{\texttt{recv}}$
- $sub^{\texttt{recv}}$ be the subsystem containing ECU $e^{\texttt{recv}}$ and bus $b^{\texttt{recv}}$

Then the set of messages involved in transmitting that signal from the sender task to the receiver task is defined as:

$$\text{reqmsg}^{\texttt{t2t}}(conf, \tau^{\texttt{send}}_s, \tau^{\texttt{recv}}, s){:=}\{m \mid m \in \text{msg}_{\mathcal{A}}(s) \text{ such that} \tag{3.6}$$

$$\Big(\text{msgalloc}_{\mathcal{A}}(m) = b^{\texttt{send}} \ \wedge\ sub^{\texttt{send}} = sub^{\texttt{recv}} \ \wedge\ e^{\texttt{send}} \neq e^{\texttt{recv}}\Big) \tag{3.7}$$

$$\vee\ \Big(\text{msgalloc}_{\mathcal{A}}(m) = b^{\texttt{global}} \ \wedge\ sub^{\texttt{send}} \neq sub^{\texttt{recv}}\Big) \tag{3.8}$$

$$\vee\ \Big(\text{msgalloc}_{\mathcal{A}}(m) = b^{\texttt{send}} \ \wedge\ sub^{\texttt{send}} \neq sub^{\texttt{recv}} \ \wedge\ \neg\text{isGw}(e^{\texttt{send}})\Big) \tag{3.9}$$

$$\vee\ \Big(\text{msgalloc}_{\mathcal{A}}(m) = b^{\texttt{recv}} \ \wedge\ sub^{\texttt{send}} \neq sub^{\texttt{recv}} \ \wedge\ \neg\text{isGw}(e^{\texttt{recv}})\Big)\Big\} \tag{3.10}$$

In the above equation the term (3.7) covers the situation where sender and receiver tasks are in the same subsystem but on different ECUs. The term (3.8) covers the global message required whenever the tasks are in different subsystems. Finally, the terms (3.9) and (3.10) handle messages required on the local bus in the sender resp. receiver subsystem.

3.1.6.3. Classification of Configurations

Configurations for DSE problems can be further classified.

Definition 3.18 (Valid Configuration)
A configuration conf for a given DSE problem p with

$$conf = (\text{type}, , \text{deadline}_{\mathcal{A}}^{\texttt{task}}, \text{deadline}^{\texttt{signal}}, \text{deadline}^{\texttt{msg}})$$

and allocation function

$$\mathcal{A} = (\text{taskalloc}, \text{msg}, \text{msgalloc})$$

is called valid if the following conditions are satisfied:

1. *no tasks must be allocated to ECUs with ECU-type ⊥, formally:*

$$\forall e \in \text{ran}(\text{taskalloc}) : \text{type}(e) \neq \perp$$

2. *All constraints have to be satisfied*
3. *for every allocated task/signal/message a local deadline is available (either initially specified or synthesized during the optimization process).*
4. *for each end-to-end deadline the sum of the (possibly synthesized) local deadlines of all the corresponding elements of the respective task tree is not larger than the end-to-end-deadline (local deadlines are sound with respect to end-to-end deadlines)*
5. *for any two communicating tasks which are allocated on different ECUs one or more messages are associated to the corresponding signal and allocated to the appropriate buses depending on the location of the two ECUs in the hardware architecture* □

Definition 3.19 (Complete Configuration)
A configuration conf is complete if

- *it is valid according to Definition 3.18*
- *each task of the task network is allocated to an ECU* □

Definition 3.20 (Schedulable Configuration)
A configuration conf is called schedulable *if*

1. *Each allocated task is schedulable, according to the schedulability analysis of tasks (see Section 2.2.5.3 on page 18)*
2. *All allocated messages on local buses are schedulable, according to the schedulability analysis of signals on CAN buses (see Section 2.7 on page 21)*
3. *All message allocated to the global bus have been allocated to at least as many bus slots as minimal required.* □

Note that the last condition is weaker than the other schedulability conditions. This is because in this work the (NP-hard) problem of finding valid schedules on TDMA-buses could not be addressed.

Definition 3.21 (Solution)
A solution for a DSE problem is a complete and schedulable configuration. □

3.1.7. Optimization Objective

In this thesis the only optimization objective used is the sum of hardware costs, which has to be minimized.

Definition 3.22 (Optimization Objective: Minimal Hardware Cost)
For a DSE problem p with a hardware architecture hw and a configuration conf for that DSE problem, the sum of hardware costs is defined as

$$cost^{\mathtt{all}}(conf) := \sum_{e \in E} \mathrm{cost}(\mathrm{type}(e))$$

where E is the set of ECUs of the corresponding hardware architectural pattern and type *is the ECU-type function specified as part of the configuration.* □

With the formalization of the optimization objective it is now possible to compare the quality of solutions.

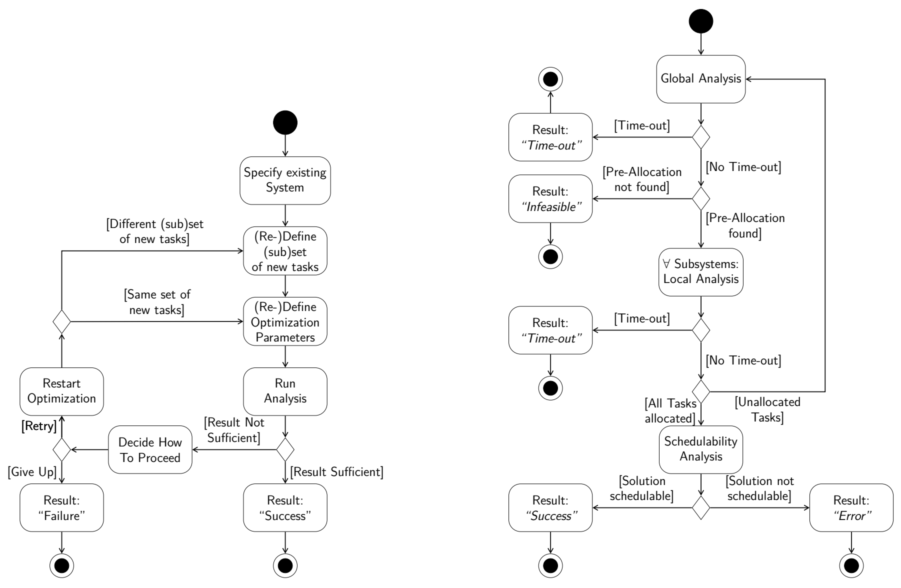
The quality of a given solution is defined to be the value resulting from the chosen optimization objective. If the objective is to minimize (maximize) the objective function, a solution Sol_1 is better than (worse than) a Sol_2 if the cost c_1 of Sol_1 are less than the cost c_2 of Sol_2. A solution for a DSE problem is optimal if there exists no better solution with respect to the chosen optimization objective.

3.2. Semi-Automatic Design Space Exploration

The mathematically model presented so far is specifically designed for enabling the decoupling the hardware subsystems of the whole embedded system from each other thus facilitating a divide-and-conquer strategy for optimization. This section contains an informal introduction to the different approaches developed as part of this PhD thesis.

Figure 3.5a depicts the DSE optimization process as seen by the user. First, the user specifies the existing system including all information already known about it. This includes the existing task network, the hardware architecture including a set of allowed ECU-types for each of the ECUs, a feasible allocation of some of the software tasks to ECUs, and an allocation of signals to messages and messages to buses, as far as the communication structure is already fixed.

Then the set of new software tasks which have to be allocated to the hardware architecture is specified. The approach explicitly supports to incrementally allocate subsets of the new software tasks one by one, thus reducing the design space handled in each iteration in order to gain a significant speedup. The drawback of using such an incremental approach is, that even exact optimization methods will produce only suboptimal solutions, due to the missing knowledge about the subsets of new tasks handled in later iterations. The quality of such solutions can be increased by choosing the incremental subsets of the new tasks wisely. By minimizing the dependencies between tasks of different subsets the additional communication in each incremental steps is kept minimal. This in turn reduces the negative effects of previous allocation decisions on the current DSE problem.



(a) Semi-Automatic Optimization   **(b)** Global and Local Analysis Process

Figure 3.5. – DSE Processes

The next step for the user is to specify parameters for the DSE process including which methods should be used. Each method can have one or more parameters, such as which backend solvers to use, time-outs for the optimization phase, etc. After choosing those parameters the DSE analysis is started. The result of the DSE analysis step is either a feasible result model, or an error condition. Error conditions include, for example the occurrence of time-out, that no feasible solution could be found, or that the analysis implementation encountered an exception (e.g. caused by the violation of preconditions in the input model, numerical problems in the back-end solvers, or — in some rare cases — programming bugs in the analysis method's implementation.

While it is clear that the user has to take action if no result has been returned, even the result of a successful optimization might not be sufficient from the point of view of the user. Insufficient solutions can be the result of erroneous input, missing constraints, or inadequate parameters for the analysis methods. Some additional constraints even may not be expressible directly for a given analysis approach (e.g. some complex allocation constraints) but could instead be incorporated by iteratively specifying additional constraints with other constraint types.

In all cases it is up to the user to decide how to proceed, for example by changing some parameters for the DSE optimization phase or by redefining the (sub)set of new tasks to be allocated. Modifying the models or DSE parameters is a manually job for the user, thus explaining the "semi-automatic" nature of this work mentioned prominently in the thesis' title. From an academic point of view, a fully automated DSE process which always finds optimal results might probably be the "Holy Grail". But it can be expected that from a practical point of view a highly configurable user-driven optimization process is much more likely to be accepted in an industrial context, where “the user” usually is a professional engineer with years of work experience.

3.2.1. Two-Tier Optimization Approach

The two-tier optimization approach presented in this thesis is based on the observation that by restricting the hardware architecture class as stated before a separation of concerns can be achieved which considerably reduces the complexity of the optimization problem. Separation of concerns in this context means that by choosing appropriate levels of abstraction many details of the involved optimization problem can be hidden. Two levels of abstractions are used in the following.

The **global analysis** is responsible for managing the spare capacity on the global bus connecting subsystems while abstracting from many subsystem internals. The global analysis calculates a so-called **pre-allocation** which assigns all unallocated tasks to subsystems while taking account of the communication via the global bus. Subsystems are abstractly represented as resources for performing computations. The approach presented in Chapter 4 explicitly represents for each subsystem the groups of ECUs of same ECU-type.

A pre-allocation is used as input for the **local analysis** which is performed separately for each subsystem. Contrary to the global analysis the local analysis explicitly incorporates all details known about the subsystem which is optimized and ignores most information about the global communication and the other subsystems.

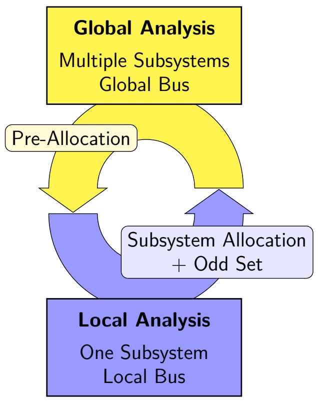


Figure 3.6. – Global and Local Analysis

As depicted in Figure 3.6, global and local analyses are running iteratively starting with the global analysis. The pre-allocation calculated in the first and each successive iteration of the global analysis is then used as input for the local analysis which is performed sequentially for each of the subsystems. In each of the local analysis steps the chosen local optimization method tries to allocate all pre-allocated tasks to ECUs of the subsystem including local communication, such that a cost-limit specified by the global analysis for the hardware architecture of the subsystem is satisfied. The cost limit must not be less than the cost for the hardware subsystem used as input for the local analysis. Depending on its implementation, the local analysis module might additionally try to minimize the hardware cost.

The result of a local analysis step for a given subsystem is a feasible allocation which might allocate some of the pre-allocated tasks to ECUs of the subsystem. Tasks that have been pre-allocated to the subsystems, but could not be allocated form the so-called **odd set** which is returned to the global analysis.

Figure 3.5b depicts the analysis process. Note that the “Error” state should not be reachable as long as the local analysis is implemented correctly and the particular preconditions of the chosen local analysis method are satisfied.

The following chapters describe in detail a global analysis approach (see Chapter 4) and a local analysis approach (see Chapter 5). Each of the two chapters contains a short overview on the features of the proposed approaches in comparison with those of alternative approaches. In Chapter 6 an evaluation of all described approaches is presented. For this the different methods developed are tested with the help of a set of benchmarks. The main indicators used for comparing the results are the runtime of the particular methods and the quality of their results, simply measured by calculating the costs of the resulting hardware architectures.

4. Global Analysis

In this chapter, the concept of **global analysis** is introduced and an approach for global analysis formulated as mathematical model is proposed. As depicted in Figure 4.1 the global analysis represents the upper tier of the two-tier optimization approach described in this thesis. The key concept behind the global analysis is the use of abstraction to improve the performance and — even more important — the scalability of the optimization approach. The hardware architecture model proposed in the preceding chapter has already been defined considering those general objectives.

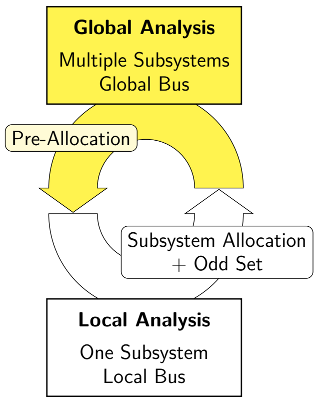


Figure 4.1. – Optimization Process Cycle: Global Analysis

Firstly, the hierarchical definition of the hardware architecture class is used to separate concerns: The global analysis is responsible for distributing functionality among the available hardware subsystems, while abstracting from the details of the single subsystems. The local analysis has full knowledge about the hardware subsystem to be analyzed including all parts of the task network already allocated to ECUs and buses inside of that subsystem, while abstracting from most details about other subsystems and communication on the global bus.

Secondly, while in a final solution of the whole optimization process the whole task network has to be schedulable (see 2.2.5.3 for the definition of schedulability) which can be interpreted as a capacity measure for real-time, a simplified notion of real-time capacity is used during global analysis. Here, the total utilization of every single ECU caused by tasks allocated to that ECU is required not to exceed a given maximal utilization which naturally is smaller or equal to 100%. It is left up to the local analysis to obtain solutions satisfying the (tighter) constraints posed by the requirement of full schedulability. The approach presented in the following sections has been published in [CST11].

4.1. Global Analysis with ECU-Type Bins

The goal of the DSE optimization is to find a cost-efficient hardware architecture and an allocation of all unallocated software tasks to ECUs such that all user-supplied constraints (e.g. that two tasks must not be allocated to the same ECU) and inherent constraints (e.g. all ECUs must be schedulable) are satisfied. The ECU-types of ECUs can be modified to gain additional capacity which increases the hardware cost of the hardware architecture.

The global analysis is not directly responsible for allocating tasks to ECUs but calculates only a **pre-allocation** of all unallocated tasks to hardware subsystems. Of course, a good prediction of the available capacity for each of the subsystems is required to find good pre-allocations. Underestimating the available capacity for a subsystem potentially wastes already existing capacity while overestimating the available capacity leads to huge odd sets containing all tasks that could not be allocated by the local analysis step.

Some important properties of a task depend on the ECU-types of the ECU the task has been allocated to, namely its worst case execution time and its memory consumption. One hypothesis of this work is, that pre-allocating tasks to subsystems as a whole without considering the ECU-types used for the subsystem's ECUs does not facilitate a very accurate estimation of the capacity required by those tasks on the one hand and the capacity provided by the subsystems on the other hand. This could of course be solved by allocating tasks not to subsystems but directly to ECUs. This would, however, be contradictory to the two-tier approach where the local analysis is responsible for actually allocating tasks to ECUs and would heavily increase the complexity of the optimization. An intermediate approach between these two extreme approaches would be advisable.

The approach presented in this chapter is based on the observation that for determining the properties of software tasks only the ECU-types to which they are allocated have to be known. It is not important to know to which ECUs the tasks actually are allocated. Therefore in addition to pre-allocating each initially unallocated task to one subsystem as a whole, an ECU-type is chosen for the task as well. For calculating the hardware cost, all ECUs with the same ECU-type are grouped to-so called **ECU-type Bins** separately for each subsystem. The capacity of an ECU-type Bin of a specific subsystem is calculated by multiplying the number of ECUs in the group with the upper bound for the utilization per ECU as specified by the user (e.g. 100%).

Figure 4.2 shows an example of a pre-allocation based on ECU-type Bins. Subsystem 1 contains two ECU-type Bins, one with one ECU of type *ARM5-C2* the other with two ECUs of type *ARM7-C1*. The remaining empty ECU socket is not considered. Subsystem 2 contains only one ECU-type Bin of type *ARM7-C1* with two ECUs. Tasks are not directly allocated to ECUs but are pre-allocated to the ECU-type Bins of a specific subsystem. Using the pre-allocation, their worst case execution times can be determined and the utilization they would cause on the ECU-type Bin is calculated.

In Figure 4.3 the whole process behind the global analysis approach is depicted. All sub-activities in the *Optimization* activity are running in parallel. If the process is successful, local analysis runs can be started, one for each hardware subsystem.

In the following sections a formal definition of the informal described approach is given.

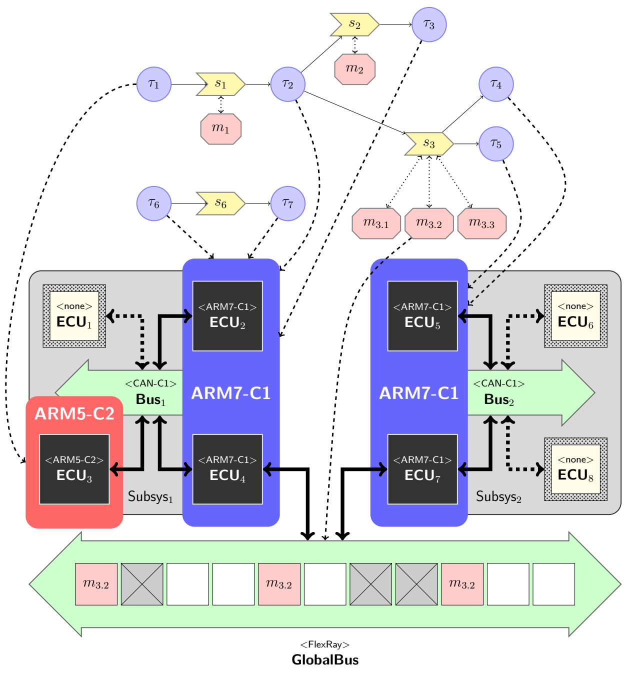


Figure 4.2. – Example Pre-Allocation

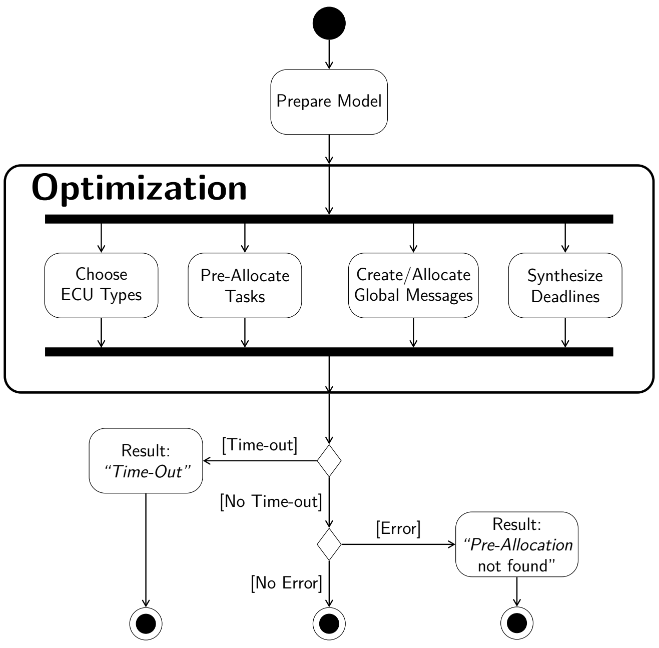


Figure 4.3. – Global Analysis Process

4.2. System Model

The mathematical model defined in Chapter 3 is the basis for the global analysis approach presented in this chapter. It is extended by a few additional concepts which are specific to the chosen approach. The terminology invented in the next sections frequently refers to properties of a given DSE problem p. Additionally, an initial (user-specified) or intermediate (created during the optimization phase) configuration *conf* is defined. All mathematical elements which are not newly invented in this chapter are taken from either the given DSE problem p or the configuration *conf* without further qualification, e.g. wherever the set of ECUs E is used in this chapter, the set of ECUs defined in the DSE problem p is meant.

Let p be a DSE problem without loss of generality with

$$p = (\mathit{tn}, \mathit{hw}, \mathit{alloc}^{\texttt{initial}}, \mathit{constr})$$

consisting of a task network

$$\mathit{tn} = (\mathit{Trees}, \mathrm{choice}^{\texttt{exec}}, \mathrm{deadline}^{\texttt{e2e}})$$

and a hardware architectural pattern

$$\mathit{hw} = (G^{\texttt{hw}}, \mathbb{E}, \mathbb{B}, \mathrm{allowed}^{E}, \mathrm{allowed}^{B}, \mathit{slots}^{\texttt{avail}})$$

The initial (user-specified) or intermediate (created during the optimization phase) configuration is defined as

$$\mathit{conf} = (\mathrm{type}, \mathcal{A}, \mathrm{deadline}^{\texttt{task}}, \mathrm{deadline}^{\texttt{signal}}, \mathrm{deadline}^{\texttt{msg}})$$

with allocation

$$\mathcal{A} = (\mathrm{taskalloc}, \mathrm{msg}, \mathrm{msgalloc})$$

and constraint specification

$$\mathit{constr} = (\psi^{\texttt{ECUs}}, \psi^{\texttt{types}}, \psi^{\texttt{never}}.\psi^{\texttt{always}}, \psi^{\texttt{allowedSubsys}}, \psi^{\texttt{sameSubsys}}, \psi^{\texttt{diffSubsys}}, \psi^{\texttt{onBus}})$$

Let $\mathcal{T}^{\texttt{unalloc}}_{\mathit{conf}}$ denote the set of unallocated tasks according to configuration *conf*.

4.2.1. Global Analysis Problem and Pre-Allocation

Each global optimization step is defined by a **global analysis problem**:

Definition 4.1 (Global Analysis Problem)
A global analysis problem is a tuple $p^{\texttt{glob}} = (p, \mathit{conf}, \mathit{util}^{\texttt{max}})$ *consisting of*

- *a DSE problem p*
- *a (initial or intermediate) configuration conf*

- *a variable $util^{\texttt{max}} \in \mathbb{N}$ defined by the user for limiting the maximal utilization available per ECU* $\square$

In the first iteration of the optimization process the global analysis problem consists of an initial DSE problem as defined in the last chapter, a configuration identical to the initial configuration specified for that DSE problem and a valuation for the maximal utilization of ECUs given by the user. For a given DSE problem

$$p_{init} = (tn, hw, alloc^{\texttt{initial}}, constr)$$

and a upper limit for the maximal acceptable utilization on every ECU $util^{\texttt{max}} \in \mathbb{N}$, the initial global analysis problem is defined as

$$\begin{aligned} p_0^{\texttt{glob}} &= (p_{init}, conf, util^{\texttt{max}}) \text{ with} \\ conf &= (\text{type}, alloc^{\texttt{initial}}, \text{deadline}^{\texttt{task}}, \\ &\qquad \text{deadline}^{\texttt{signal}}_{conf}, \text{deadline}^{\texttt{msg}}_{conf}) \end{aligned}$$

During the optimization process the concept of **pre-allocation** is used to describe intermediate partial solutions used as the input for the local analysis steps. A pre-allocation is a preliminary allocation of tasks to subsystems without a full schedulability check. All subsystem ECUs with the same ECU-type form an ECU-type bin specific to that subsystem. The computation and memory capacity of all ECUs in an ECU-types bin is subsumed to form a virtual ECU-type bin capacity.

Instead of allocating tasks to ECUs of a subsystem each task is allocated to a subsystem and an ECU-type bin. As all platform-specific properties of software tasks are specified relative to ECU-types, choosing an ECU-type bin for a task effectively determines those properties. This has the advantage to significantly reduce the number of possible allocations, compared to allocating tasks directly to ECUs.

In addition to pre-allocating tasks, the global analysis approach chooses ECU-types for ECUs, which determines the size of the individual ECU-type bins and the global hardware cost, too.

Definition 4.2 (Pre-Allocation)
For the given DSE problem p a pre-allocation is a tuple

$$conf^{\texttt{pre}} = (conf, \text{tasksubsys}, \text{tasktype}, \text{type})$$

consisting of

- *a valid configuration conf*
- *a function for pre-allocating unallocated tasks to subsystems*

$$\text{tasksubsys}\colon \mathcal{T}^{\texttt{unalloc}}_{conf} \to Sub$$

- *a function for pre-allocating unallocated tasks to ECU-types*

$$\text{tasktype}\colon \mathcal{T}_{conf}^{\texttt{unalloc}} \to \mathbb{E}$$

- *an ECU-type function* type

where $\mathcal{T}_{conf}^{\texttt{unalloc}}$ denotes the set of unallocated tasks according to configuration conf. □

4.2.2. Limitations of the Current Approach

Currently the exploration of the hardware architecture design space is limited to finding ECU-types for ECUs. The pre-defined types of buses remain unchanged. Therefore it is assumed that for each DSE problem which has to be solved by using the two-tier approach defined in this thesis, for the global and each local bus the respective set of allowed bus types contains exactly one element (which of course does not mean that all buses have to have the same bus type). Furthermore, the global bus has to use a TDMA-based bus type and every local bus a priority-based bus type, see Chapter 3.

Let p denote the DSE problem of the global analysis problem $p^{\texttt{glob}}$. For simplification, all bus slots of the global TMDA-based bus $b^{\texttt{global}}$ are required to have the same predefined length denoted as $size_{p^{\texttt{glob}}}^{\texttt{global}}$. The length is defined as part of the specification of the global analysis problem. It is required that the predefined length is large enough to hold any message. The sequence of all bus slots is called a *TDMA round.* The length of the TDMA round on the global bus in seconds is defined as

$$\lambda_p^{\texttt{TDMA}} := \left|\text{slots}(b^{\texttt{global}})\right| size_{p^{\texttt{glob}}}^{\texttt{global}}$$

Assuming that all bus slots are large enough to hold any message, a check whether or not a given message fits into a given bus slot is not required during the optimization process.

4.2.3. Measuring Capacity

The constraints in this section describe the inherent limitations of the used hardware elements, such as computation capacity and available memory for ECUs, and the number of available slots on the global bus. The number of ECUs with a given ECU-type $t \in \mathbb{E}^\star$ on a subsystem $sub \in S^\star$ is calculated by function

$$\begin{aligned}
&\text{typecount}\colon CONF^\star \times S^\star \times \mathbb{E}^\star \to \mathbb{N}\\
&\text{typecount}(conf, sub, t) := |E_{sub,t}| \text{ with}\\
&E_{sub,t} := \left\{e \mid e \in E \ \land\ \text{subsys}(G^{\texttt{hw}}, e) = sub \ \land\ \text{type}(e) = t\right\}
\end{aligned}$$

The allocation of tasks to ECUs must not exceed the available computation capacity. The following constraints ensure that the load imposed by the additional tasks remains below the ECU capacity limit. Let $\mathcal{T}$ denote the set of tasks of the DSE problem p.

The (worst case) utilization caused by a given task is as usually defined as the worst case execution time divided by the period (see Section 2.2.5.2 on page 17).

In this chapter, the utilization induced by a task τ allocated to an ECU with ECU-type is defined as:

$$\text{util}(p, \tau, t) := \begin{cases} \frac{\text{wcet}_p(\tau, t)}{\text{period}^{\text{task}}(\tau)} & \text{if } \tau \in \mathcal{T} \wedge \text{choice}^{\texttt{exec}}(\tau, t) \text{ is defined} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

with $\mathcal{T}$ being the set of tasks and $\mathbb{E}$ being the set of ECUs.

Note that the WCET function wcet is also based on the binary choice function $\text{choice}^{\texttt{exec}}$ of the DSE problem because that function is required to obtain unique WCET values for each task with respect to each ECU-type, see Definition 3.17 (page 44) for details.

The already allocated tasks are causing a **base load** on the ECUs to which they are allocated, for both the ECU's computation and its memory capacity. The **utilization base load** for a given ECU e is calculated by subsuming the utilization requirements of all tasks allocated to that ECU. As the utilization requirement of a task depends on its WCET for the chosen ECU-type, the base load depends on the ECU-type, formally:

$$\text{ubl}(\mathit{conf}, e, t) := \sum_{\tau \in \mathcal{T}_e} \text{util}(p, \tau, t) \text{ with}$$
$$\mathcal{T}_e = \Big\{ \tau \mid \tau \in \text{dom}(\text{taskalloc}) \wedge \text{taskalloc}(\tau) = e \wedge \text{util}(p, \tau, t) \neq \texttt{undef} \Big\}$$

Note that $\mathcal{T}_e$ denotes the set of tasks allocated to ECU e.

The utilization base load of the ECU-type bin for a given ECU-type t in a subsystem sub is defined as follows:

$$\text{ubl}^{\mathit{Sub},\mathbb{E}}(\mathit{conf}, \mathit{sub}, t) := \sum_{e \in E_{\mathit{sub},t}} \text{ubl}(\mathit{conf}, e, t) \text{ with}$$
$$E_{\mathit{sub},t} := \Big\{ e \mid e \in E \wedge \text{subsys}(G^{\texttt{hw}}, e) = \mathit{sub} \wedge \text{type}(e) = t \Big\}$$

where $G^{\texttt{hw}}$ is the architecture tree of DSE problem p with the set of ECUs E.

The memory base load is calculated similarly:

$$\text{mbl}(\mathit{conf}, e, t) := \sum_{\tau \in \mathcal{T}_e} \text{memreq}_p(\tau, t) \text{ with}$$
$$\mathcal{T}_e := \Big\{ \tau \mid \tau \in \text{dom}(\text{taskalloc}) \wedge \text{taskalloc}(\tau) = e \wedge \text{memreq}_p(\tau, t) \text{ is defined} \Big\}$$

The memory base load of a given ECU-type t on a subsystem sub is defined as:

$$\mathrm{mbl}^{Sub,\mathbb{E}}(conf, sub, t) := \sum_{e \in E_{sub,t}} \mathrm{mbl}(conf, e, t) \text{ with}$$

$$E_{sub,t} = \Big\{ e \mid e \in E \ \wedge \ \mathrm{subsys}(G^{\mathtt{hw}}, e) = sub \ \wedge \ \mathrm{type}(e) = t \Big\}$$

for the architecture tree $G^{\mathtt{hw}}$.

4.2.3.1. Utilization Constraints

One parameter for our optimization algorithm is the maximal available utilization on one ECU-type, defined as $util^{\mathtt{max}} \in [0, 1]$. In theory, each ECU can be utilized up to 100%. From [LL73] we know that for fixed-priority preemptive systems with strictly periodically tasks (deadlines equal to periods) there is no guarantee for schedulability if the ECU utilization is above approximately 69%.

The maximal available utilization on each subsystem per ECU-type depends on the number of ECUs to which that ECU-type has been assigned. Let

$$conf_p^{\mathtt{pre}} = \Big(conf, \mathrm{tasksubsys}_{conf_p^{\mathtt{pre}}}, \mathrm{tasktype}_{conf_p^{\mathtt{pre}}}, \mathrm{type}_{conf_p^{\mathtt{pre}}} \Big)$$

be a pre-allocation. The available utilization per ECU is calculated by subtracting the utilization already used by allocated tasks from the maximal acceptable utilization:

$$\mathrm{utilavail}^{E,\mathbb{E}}(conf, e, t) := \max \left\{ 0, util^{\mathtt{max}} - \mathrm{ubl}(conf, e, t) \right\}$$

where $\mathcal{T}_{e,conf}$ denotes the set of tasks allocated to ECU e by configuration $conf$.

The available utilization in subsystem sub for ECU-type t is determined by the number of ECUs in the subsystem with the given ECU-type multiplied by the user-defined maximal allowed utilization. The base load of the ECUs is subtracted from that value. Formally:

$$\begin{aligned} &\mathrm{utilavail}^{Sub,\mathbb{E}}(conf_p^{\mathtt{pre}}, sub, t) := \\ &\qquad \mathrm{typecount}(conf_p^{\mathtt{pre}}, sub, t) \cdot util^{\mathtt{max}} - \mathrm{ubl}(conf, sub, t) \end{aligned}$$

This accumulated maximal available utilization limits the number of tasks which can be deployed to each ECU-type per subsystem. The utilization requirement of all tasks allocated to a subsystem sub with respect to an ECU-type t is:

$$\mathrm{utilreq}^{Sub,\mathbb{E}}(conf_p^{\mathtt{pre}}, sub, t) := \sum_{\tau \in \mathcal{T}^{\mathtt{pre}}{}_{sub,t}} \mathrm{util}(p, \tau, t) \text{ with}$$

$$\mathcal{T}^{\texttt{pre}}sub, t := \Big\{\tau \mid \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf^{\texttt{pre}}_p} \land \\ \text{tasksubsys}_{conf^{\texttt{pre}}_p}(\tau) = sub \land \text{tasktype}_{conf^{\texttt{pre}}_p}(\tau) = t\Big\}$$

The optimization process has to guarantee that the required utilization per subsystem is not larger than the available utilization:

$$\forall sub \in Sub, \forall t \in \mathbb{E} : \\ \text{utilavail}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) \ \geq \ \text{utilreq}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) \tag{4.1}$$

with *Sub* being the set of subsystems and $\mathbb{E}$ being the set of ECU-types being used in DSE problem p.

4.2.3.2. Memory Constraints

Memory constraints are handled analogously to utilization constraints.

The available memory on any ECU is calculated by taking the memory provided by the ECU's ECU-type and subtracting the amount of memory already used by tasks allocated to that ECU, formally:

$$\text{memavail}^{E,\mathbb{E}}(conf, e, t) := \max\{0, \text{mem}(t) - \text{mbl}(conf, e, t)\}$$

The accumulated maximal available memory for any given subsystem *sub* and ECU-type t is defined as:

$$\text{memavail}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) := \\ \text{typecount}(conf^{\texttt{pre}}_p, sub, t) \cdot \text{mem}(t) - \text{mbl}(conf^{\texttt{pre}}_p, sub, t)$$

The memory required by all tasks on subsystem *sub* and ECU-type t is defined as:

$$\text{memreq}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) := \sum_{\tau \in \mathcal{T}^{\texttt{pre}}sub,t} \text{memreq}_p(\tau, t)$$

$$\mathcal{T}^{\texttt{pre}}sub, t := \Big\{\tau \mid \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \land \text{tasksubsys}_{conf^{\texttt{pre}}_p}(\tau) = sub \land \\ \text{tasktype}_{conf^{\texttt{pre}}_p}(\tau) = t\Big\}$$

Hence, every pre-allocation has to satisfy the requirement that enough memory is available on each subsystem for all ECU-type bins:

$$\forall sub \in Sub, \forall t \in \mathbb{E} : \\ \text{memavail}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) \geq \text{memreq}^{Sub,\mathbb{E}}(conf^{\texttt{pre}}_p, sub, t) \tag{4.2}$$

4.2.3.3. Global Bus Utilization

In this thesis a necessary but not sufficient condition is used to characterize the capacity of the global bus. The condition ensures that to each message on the global bus a sufficient number of bus slots is allocated.

First a predicate is defined which evaluates to true whenever a given signal has to be allocated on the global bus (similar to the predicates specified in the previous chapter):

$$\text{needsGlobal}^{\texttt{GA}}(p^{\texttt{glob}}, \mathit{conf}_p^{\texttt{pre}}, s) := \begin{cases} \texttt{true} & \text{if needsGlobal}(p, \mathit{conf}, s) = \texttt{true} \\ \texttt{true} & \text{or if } |\mathit{Sub}_s| \geq 2 \\ \texttt{false} & \text{otherwise} \end{cases} \tag{4.3}$$

where Sub_s is for any signal s the set of subsystems where at least one of the sender or receiver tasks is allocated either by the configuration specified for the global analysis or by the pre-allocation

In this thesis the signal period is by definition equal to the period of its sender task. For a signal s with sender task τ the number of required bus slots for its message on the global bus is defined as:

$$\text{slotsreq}(p, s) := \begin{cases} \left\lceil \frac{\text{period}^{\texttt{task}}(\tau)}{\lambda_p^{\texttt{TDMA}}} \right\rceil & \text{if needsGlobal}^{\texttt{GA}}(p^{\texttt{glob}}, \mathit{conf}_p^{\texttt{pre}}, s) = \texttt{true} \\ 0 & \text{Otherwise} \end{cases}$$

As described in the last chapter in Section 3.1.6.1, there exists a subset of signals for which it is already known that they must be allocated to the global bus. For another subset it is known those signals will never be allocated to the global bus. Only signals for which no decision has been taken (because that decision depends on tasks not yet allocated by the given configuration) have to be considered during the global analysis optimization process.

The signals which must be allocated to the global bus consume bus slots, thus reducing the number of available bus slots on the global bus.

Let $S_{\mathit{conf}}^{\texttt{always}}$ denote the set of signals which must have a message on the global bus. The total number of available bus slots on that bus is reduced by the number of bus slots required by signals in that set, formally:

$$\mathit{slots}_{\mathit{conf}}^{\texttt{remain}} := \left|\mathit{slots}_p^{\texttt{avail}}\right| - \sum_{s \in S_{\mathit{conf}}^{\texttt{always}}} \text{slotsreq}(p, s) \tag{4.4}$$

where $\mathit{slots}_p^{\texttt{avail}}$ is the set of available bus slots on the global bus as specified in the DSE problem. Note that in the DSE problem some bus slots of the global bus might be excluded from the set of available slots.

A valid pre-allocation may not use more bus slots than are available on the global bus:

$$\sum_{s \in S} \text{slotsreq}(conf_p^{\texttt{pre}}, s) \leq slots_{conf}^{\texttt{remain}} \tag{4.5}$$

Note that the actual slot assignment for messages on the global bus is not determined by this analysis approach. The problem of finding a feasible schedule on a time-triggered bus itself is a complicated optimization problem, see e.g. [Luk+09] where an approach for calculating valid schedules for the static segment of a FlexRay bus is presented.

4.2.4. Constraints for Task Allocation

Most of the user-defined constraints defined in Section 3.1.4 have to be considered in the global analysis. Some other constraints cannot be considered directly, but have to be handled by the local analysis.

If allowed/forbidden subsystems are specified for some or all tasks, these constraints can be handled directly by limiting the pre-allocation of those tasks to the respective subsystems. In contrary constraints specifying that a set of tasks has to be allocated to the same ECU cannot be analyzed by the global analysis.

But for that kind of constraint derived constraints can be formulated stating that those tasks must be allocated to the same subsystem (otherwise the local analysis would obviously not be able to satisfy that constraint). Constraints forbidding certain tasks to be allocated to the same ECU can be handled by the global analysis only in very special cases, e.g. if all ECUs of a subsystem are forbidden for a given task, then that subsystem can be added to the lists of forbidden subsystem for that tasks. In all other cases those constraint types are used solely during the local optimization phase.

The same is true for all constraints enforcing that a given signal has to be allocated to a given local bus, as local buses are not considered in the global analysis process. Table 4.1 gives an overview on the constraint types defined in the last chapter, and how and where they can/could be incorporated in the two-tier optimization approach.

Constraint Type	**Global Analysis**	**Local Analysis**
Allowed ECUs per Task	○	●
Allowed ECU Types per Task	●	●
Never on same ECU	○	●
Always on same ECU	○	●
Allowed ECUs	○	●
Forbidden ECUs	○	●
Allowed Subsystems	●	–
Forbidden Subsystems	●	–
Always on Bus	○ (only global bus)	○ (only local bus)

Table 4.1. – Support for Constraint Types on different Levels: (●) fully supported, (○) partially supported, (–) unsupported

4.2.5. Approaches for Deadline Synthesis

The problem formalization given in Chapter 3 explicitly allows the specification of tasks and signals which initially do not have local deadlines, as long as they are subject to at least one end-to-end deadline. Such an end-to-end deadline can then be used to synthesize local deadlines during the optimization process. Note that multiple end-to-end-deadlines may be effective for a task/signal due to the definition of such end-to-end-deadlines on paths of task trees.

Synthesizing local deadlines for tasks and signals is a very complex process with many implications on both the system and the subsystem level. For example, choosing a small deadline for a task/signal will lead to a higher priority compared to the other tasks/signals in a subsystem, which in turn influences the chances to find a feasible allocation for that task/signal during the subsystem-local analysis. But all other tasks and signals in the same subsystem will potentially be influenced by this decision, and potentially rendering them unschedulable. If a large deadline is chosen for a task/signal, then it will get a low priority instead. The task/signal will then influence only a small number of other tasks/signals, but in turn is influenced by many higher priority task/signals, which makes it harder to allocate that task/signal successfully. On a global analysis scope based on utilization, it is not possible to deduce optimal local deadlines due to the lack of detailed information.

Based on these considerations and the fact that deadline synthesis is a smaller subtask of this PhD thesis, a basic heuristic approach for deadline synthesis is presented and implemented. The approach aims for achieving synthesized deadlines which are considered likely to ease the subsystem-local analyses. The approach synthesizes deadlines for tasks and signals for which no initial local deadline has been specified, while leaving all initially specified deadlines untouched. It retains the priorities of all allocated tasks and messages on their respective ECUs and buses, and ensures that they comply with the deadline-monotonic priority assignment paradigm.

The following steps are used to perform the deadline synthesis:

1. Perform a schedulability analysis for all already allocated tasks and signals/messages
2. For all tasks without fixed deadlines: determine upper and lower bounds for their local deadlines. Those bounds can also be symbolic in the sense that they depend on other still undefined local deadlines.
3. For all ECUs which are considered *full*: change all synthesized deadlines to the lowest possible values which still allow each task to finish its execution without violating its particular local deadline. Initially defined local deadlines are not changed.
4. for all signals estimate the required local deadlines
5. Estimate how much slack is available for each of the end-to-end deadlines considering all already known local deadlines and the estimates for the synthesized local signal deadlines

6. Calculate a target local deadline for each task without a user-defined local deadline by equally distributing the slack among all tasks

7. Calculate the difference of that hypothetical optimal local deadline and the synthesized local deadline and sum up the positive differences (where the synthesized local deadline is smaller than the hypothetical value) as part of the objective function

Note that there might be different definitions of what it means if an ECU is *full*. The simplest definition is to mark an ECU as being full if none of the still unallocated tasks can be allocated to that ECU no matter what ECU-type is used. An alternative is to define a minimal required remaining utilization and mark all ECUs as being *full* which remaining utilization is smaller than that threshold.

Note that by adding the positive difference to the objective function to be minimized, those differences are minimized, too. This way it can be ensured, that the available slack of the end-to-end-deadlines is equally distributed among the task/signals.

The following paragraphs explain in detail how the single steps can be defined.

Deadline Bounds for Unallocated Software Tasks For an unallocated task without a user-defined local deadline only few information is already usable. One reliable property is the activation period which does not change during the whole optimization process. The effective worst case execution time is unknown until the task is allocated to an ECU which has a defined ECU-type.

While the local analysis is responsible for this final allocation decision, the global analysis pre-allocates each task to a subsystem and an ECU-type. Even though the local analysis may allocate a task to an ECU with a different ECU-type, the pre-allocation is useful because it allows a prediction of the effective worst case execution time.

Let $wcet_{\tau}^{\texttt{eff}}$ denote the effective worst case execution time of an unallocated task τ as determined by the pre-allocation. The synthesized deadline d_{τ} has to be larger or equal to the effective worst case execution time otherwise that task would not be schedulable. In the same time the synthesized deadline has to be smaller or equal to the task's activation period as stated by the premises of this thesis. This leaves us with the following bounds:

$$\begin{gathered} d_{\tau} \in \mathbb{N} \text{ such that} \\ wcet_{\tau}^{\texttt{eff}} \leq d_{\tau} \leq \text{period}^{\texttt{task}}(\tau) \end{gathered}$$

Deadline Bounds for Allocated Software Tasks For every already allocated task, lots of useful information is available, namely its activation period, its current (synthesized) local deadline (e.g. stemming from a previous global/local analysis iteration) and its worst case execution time, determined by the currently chosen ECU-type of the ECU the task is allocated to. Even better, the worst case response time of the task has been calculated via schedulability analysis.

Let τ be a task allocated to an ECU e with synthesized local deadline d_{τ}.

In case there are additional tasks allocated to the same ECU, let $\tau^{\texttt{higher}}$ denote the task having the next higher priority than τ (if such a task exists), and let denote $\tau^{\texttt{lower}}$ the task having the next lower priority than τ (if such a task exists). Let r_τ denote the calculated worst case response time of task τ and $\text{period}^{\texttt{task}}(\tau)$ its activation period (as defined in Chapter 3).

For ensuring the schedulability of τ while in the same time retaining all the current task priorities on the ECU in compliance with the deadline-monotonic priority assignment paradigm, changes to the synthesized deadline can only be done in a very limited fashion.

On the one side the updated synthesized deadline has to be larger or equal to the worst case response time (otherwise the task would be unschedulable). Additionally, it has to be larger or equal to the deadline $d_{\tau^{\texttt{higher}}}$ of the task which has the next higher priority on that ECU (but only if τ has not the highest priority itself). This lower bound prohibits changes to task priorities in order to keep the allocation compliant with the deadline-monotonic paradigm.

On the other side, the updated synthesized deadline has to be smaller or equal to the tasks period (due to the requirement to use only tasks whose local deadlines are smaller or equal to their period). Additionally, it has to be smaller than the deadline of the task with the next lower priority (if such a task exists on the ECU), which is itself required to retain the current task priorities. The synthesized deadline d'_τ therefore has to satisfy the following condition:

$$\begin{aligned} d'_\tau &\in \mathbb{N} \text{ such that} \\ \max(r_\tau, d_{\tau^{\texttt{higher}}}) \leq & d'_\tau \leq \min(\text{period}^{\texttt{task}}(\tau), d_{\tau^{\texttt{lower}}}) \end{aligned}$$

Tightening Bounds on ECUs marked as *full* For every task without an initially defined local deadline that is allocated to an ECU marked as *full*, the following procedure is used to tighten the local deadlines: First the local deadline for the highest priority task is changed to the lower bound. Then the bounds are recalculated and the procedure proceeds with the task with the next lower priority.

Deadline Synthesis for Signals The synthesis of local deadlines for signals is even more complicated than the synthesis of task deadlines. The reason for this is that while local task deadlines influence only exactly one ECU, local signal deadlines influence either a whole subsystem if the signal has to be allocated to a local bus, or even the whole system if it is allocated to the global bus.

If for example the deadline for a signal's message allocated to the priority-based local bus of any subsystem is tightened too much, this message can block the whole local bus even if it is the only message allocated to that bus, and even if the message itself causes only a very low utilization on the bus. This has to do with the fact that the message transmission over priority-based buses considered, cannot be preempted, not even by higher priority message. If now only one message is allocated to such a bus, then no blocking by lower priority messages has to be taken in consideration during schedulability analysis. If the synthesized deadline is set/changed to be equal to the calculated worst case response time of the message, not a single additional message can be allocated to that bus. Messages with a higher priority cannot be allocated to the bus, because there simply is no slack available for preemptions by higher priority messages. In the same time even messages with lower priority could not be allocated. They would impose a blocking time on the already allocated message which would increase the worst case transmission time of this message such that it exceeds the deadline.

On the global TDMA-based bus, decreasing the deadline of any message allocated to one or more slots will increase the number of required bus slots.

For the above reasons a simplified method for synthesizing local signal/message deadlines is chosen in this thesis. To each message that does not have an initially specified deadline, a local deadline equal to the activation period of the corresponding signal is assigned. For every signal, the following situations can be distinguished:

- The signal is transmitted only locally. Than the local deadline can be 0
- The sender task and at least one receiver tasks are allocated to different ECUs in the same subsystem. In this case only one message on the local bus is required and the signal's deadline can be set to its activation period.
- The sender task is allocated to a gateway ECU and one or more receiver tasks are allocated to gateway ECUs of different subsystems. Than only one message on the global bus is required, which is similar to the previous case
- A sender task is allocated to any non-gateway ECU and at least one receiver task is allocated to a different subsystem. Than at least two messages are required, one on local bus of the sender task's subsystem and one on the global bus
- One additional message is required if the receiver tasks is also allocated to a non-gateway ECU

As a result a signal's deadline is defined as a multiple of the signal's activation period where the allocation of the sender and receiver tasks determines the correct factor between 0 and 3 is used.

However as the global analysis does not allocate tasks to ECUs directly and therefore does not know whether or not a transmission on any of the subsystem local buses is required, the above cases can be condensed to only two cases relevant for global analysis: For each signal for which at least one sender/receiver task is still unallocated the local deadline is synthesized depending on the choice whether or not it has to be transmitted on the global bus. If the signal is not transmitted on the global bus (all the sender/receiver tasks are allocated and/or pre-allocated to the same subsystem), its synthesized deadline is set to its activation period, otherwise it is set to its activation period multiplied by 3.

4.2.5.1. Synthesizing Target Deadlines

A simple heuristic is used to equally distribute the calculated remaining fraction of each end-to-end deadline among the particular affected tasks for which no fixed local deadline has been specified. For each task a (hypothetical) target deadline is estimated based on the task parameters, the size of the remaining slackness in the deadline and the total number of participating tasks involved in the end-to-end deadline. This value is hypothetical in the sense that during the global analysis the real remaining slackness of any end-to-end deadline might be smaller than estimated. The target deadlines may then be too large to be achieved during the optimization process. Therefore the target values are not formulated as constraints of the optimization problem but instead are incorporated into the objective function. The analysis back-end is driven to choosing synthesized deadlines which are close to the target deadlines by adding a penalty to the optimization objective. That penalty is proportional to the difference between the synthesized deadlines and the estimated target deadline.

Using this construction the distribution of the available end-to-end deadline slack is somewhat balanced between the involved tasks. Solutions are avoided where e.g. all the slackness of an end-to-end deadline is used in only one task's local deadline while all other tasks affected by that end-to-end deadline do not get a share of that slack.

First the remaining part of the end-to-end deadline is estimated. For the given configuration and one of the given end-to-end deadlines d let $\mathcal{T}_d$ be the set involved tasks with

$$\mathcal{T}_d := \mathcal{T}_d^{\texttt{defined}} \dot{\cup} \mathcal{T}_d^{\texttt{alloc}} \dot{\cup} \mathcal{T}_d^{\texttt{unalloc}}$$

where $\mathcal{T}_d^{\texttt{defined}}$ is the set subset of tasks with initially defined deadlines, $\mathcal{T}_d^{\texttt{alloc}}$ is the subset of allocated tasks without such a deadline and $\mathcal{T}_d^{\texttt{unalloc}}$ is the subset of unallocated tasks also without such a deadline.

Let furthermore S_d be the set of signals which are affected by the end-to-end deadline with

$$S_d := S_d^{\texttt{defined}} \dot{\cup} S_d^{\texttt{undef}}$$

where $S_d^{\texttt{defined}}$ is that subset of signals for which a local deadline has been initially defined, and $S_d^{\texttt{undef}}$ is the set of signals without such a local deadline.

The remaining fraction of the end-to-end deadline d is calculated by subtracting all already known local deadlines:

$$d_d^{\texttt{remain}} := d - \sum_{\tau \in \mathcal{T}_d^{\texttt{defined}}} \text{deadline}^{\texttt{task}}(\tau) - \sum_{s \in S_d^{\texttt{defined}}} \text{deadline}^{\texttt{signal}}(s)$$

This remaining fraction is then used for synthesizing local deadlines for all tasks and signals affected by that end-to-end deadline which do not have initial local deadline. The signal's deadline is synthesized to be the equal to the period , if it is known already that the signal will not be transmitted on the global bus. Otherwise the signal's deadline is synthesized to be activation period multiplied by three (see above). Let $d_d^{\texttt{remain,signal}}$ denote the fraction of the remaining slack assigned to the signal. Then the remaining fraction available for the tasks is defined as:

$$d_d^{\texttt{remain,task}} := d_d^{\texttt{remain}} - d_d^{\texttt{remain,signal}}$$

For each of the tasks without initial deadlines multiple end-to-end deadlines might apply in parallel, depending on their location in their task trees (e.g. the root node is by definition affected by every end-to-end deadline defined for the tree). Let $D_\tau^{\texttt{e2e}}$ denote the set of all end-to-end deadlines which are affecting task τ. The target deadline for that task is then defined as follows:

$$d_\tau^{\texttt{target}} := \min\left\{\text{period}^{\texttt{task}}(\tau), wcet_\tau^{\texttt{max}} + \min\left\{\frac{d_d^{\texttt{remain,task}}}{|\mathcal{T}_d^{\texttt{alloc}} \cup \mathcal{T}_d^{\texttt{unalloc}}|} \;\middle|\; d \in D_\tau^{\texttt{e2e}}\right\}\right\}$$

where $wcet_\tau^{\texttt{max}}$ denotes the maximal worst case execution time defined for τ for the set of ECU-types used in the current DSE problem. In the above equation this maximal worst case execution time has been chosen, such that synthesized task deadlines are always greater than the greatest WCET. The minimal slack divided by the number of tasks to which that slack has to be distributed is then added to the maximal WCET, considering all end-to-end deadlines which are relevant for task τ. Because the resulting sum might very well exceed the tasks period, the minimum of the result and the task's period is used to define the hypothetical optimal task deadline.

4.2.6. Optimization Objective: Minimize Target Deadline Differences

Based on the defined target deadlines for tasks for which a local deadline has to be synthesized, a secondary optimization can be formulated for minimizing the differences between the synthesized deadlines and the target deadlines.

$$penalty^{\texttt{deadsynth}} := \sum_{d \in \text{deadline}^{\texttt{e2e}}} \; \sum_{\tau \in \mathcal{T}_d^{\texttt{alloc}} \cup \mathcal{T}_d^{\texttt{unalloc}}} \max\left\{0, d_\tau^{\texttt{target}} - d_\tau\right\}$$

where $d_\tau^{\texttt{target}}$ is the target deadline of task τ and d_τ is the deadline synthesized during the global optimization phase. Note that intentionally only deadlines that are too small are penalized (by using the min operator).

4.2.7. Optimization Objective: Minimize Number of Used Bus Slots

$$penalty^{\texttt{slot}} := \sum_{s \in M} \text{slotsreq}(conf_p^{\texttt{pre}}, s)$$

4.2.8. Optimization Objective: Minimize Hardware Cost

For a given pre-allocation

$$conf_p^{\mathtt{pre}} = \left(conf, \text{tasksubsys}_{conf_p^{\mathtt{pre}}}, \text{tasktype}_{conf_p^{\mathtt{pre}}}, \text{type}_{conf_p^{\mathtt{pre}}}\right)$$

the total cost are defined as follows:

$$cost^{\mathtt{all}}(conf_p^{\mathtt{pre}}) := \sum_{e \in E} \text{cost}(\text{type}(e))$$

The optimization objective is to find a pre-allocation that satisfies all constraints and has minimal overall cost. As stated in the cost function, the cost only depend on the choice of the ECU-types. The other components of a configuration restrict the solution space. For more expensive ECU-types, it can be expected that tasks have lower WCETs, which reduces their utilization on those ECU-types. On such ECU-types, more tasks can be deployed, which could allow the optimization process to leave one or more ECUs empty, which in turn reduces the overall cost. A second advantage of more expensive ECU-types is that they usually provide more memory, which is also required for allocating more tasks to them.

Note that in industrial applications, software tasks may use optimized code for certain ECU-types, which would make those types the preferable choice. Our approach can handle this situation well, because for each task the input models contain a separate WCET for each ECU-type.

4.2.9. Secondary Optimization Objectives

Several optional optimization objectives could be useful for certain problem instances. However, as no multi objective optimization is used in this thesis, secondary optimization objectives have to be incorporated into the main objective function. This can be done via penalty functions where decreasing the quality of a solution with respect to a secondary objective leads to an increased (if the optimization objective has to be minimized) penalty added to the objective value.

4.2.10. Approaches for Guaranteeing Termination

Several means are applied for guaranteeing the termination of the global analysis optimization.

4.2.10.1. Tabu List

If a task could not be allocated to a subsystem in any of the global iterations (which means that the task was contained in the odd set for that subsystem after the local analysis) that subsystem is added to the **tabu list** of that task. The global analysis is not allowed to pre-allocate a task to one the subsystems contained in that task's tabu list. Tabu lists are realized by temporarily adding the forbidden subsystems to the task-specific sets of forbidden subsystem specified by the user.

Of course, if the tabu list handling is too strict, this might lead to situations where DSE problems are reported to be infeasible spuriously. For example if a task could not be allocated to a subsystem because of very tight restrictions for the maximal hardware cost in one global iteration, it might very well be possible to allocate that task to that subsystem in one of the succeeding global iterations if the cost constraints are relaxed.

Currently, situations like that are avoided by resetting the tabu list several times before giving up. For this, the global iterations are organized in *rounds* where for each round different strategies apply. Each round consists of maximal n global analysis iterations (where n is the number of subsystems). In the first round, the global analysis calculates tight cost constraints for the subsystem-specific local analyses. The tabu lists are growing as described above. They are reseted only if an infeasible constellation is detected, e.g. if for a task the only subsystem which not in the tabu list is forbidden due to user-specified constraints. If no complete solution can be found in the first n iterations the tabu lists of all still unallocated tasks are reseted. In the following n global iterations they are used again as described. But this time all cost restrictions are relaxed yet not fully disabled. Again the tabu lists are reseted if no solution can be found during these iterations. In the last round no cost restrictions are applied at all, and the tabu lists are enabled again. If still no solution can be found the global analysis gives up reporting "Problem infeasible".

If the backend reports during one of the rounds that the MILP problem is infeasible, the next round is started immediately and all relaxations take effect.

4.2.11. Complexity Considerations

Some consideration about the complexity of the problem are advisable:

Theorem 4.3
The global analysis decision problem described in this thesis is NP-hard. □

Proof The bin packing problem (BPP) which is known to be NP-hard (a compact proof of this property can be found in [KV10]) can be polynomial-time reduced to our problem formulation.

First, we assume to have only exactly one ECU-type with cost 1 and with enough memory such that the whole set of tasks could be allocated to a single ECU. Furthermore, we assume that this ECU-type allows a utilization of up to 100% by choosing $util^{\mathtt{max}} = 1$. Next, we restrict each subsystem to contain exactly one ECU, which can either be typed by that ECU-type or remain untyped. We choose the global bus to be powerful enough to handle any possible message communication load.

Now each bin of the BPP corresponds to one ECU and each item to one task. By choosing the size of the available memory of the ECU-type large enough to hold the whole task set, only the constraints for the maximal utilization are still effective. They correspond directly to the size constraints of the BPP bins. ■

Theorem 4.4
The global analysis optimization problem described in this thesis is an NP-hard optimization problem. □

PROOF Minimizing the number of required bins in BPP is an NP-hard optimization problem. The same reduction as already presented in the proof of Theorem 4.3 is used. Each ECU to which ECU-type has been assigned gets the cost 1 (as assigned to the ECU-type), all other ECUs cost 0. Then minimizing the hardware cost effectively means minimizing the number of ECUs used. ■

4.3. Optimizing with HySAT

Encoding the mathematical equations given in section 4.2 for the SMT solver HySAT is straightforward. Background information about HySAT can be found in [Her10].

4.3.1. Encoding the Problem

For each of the sets used, unique identifiers are assigned to all its elements (e.g. each ECU in E gets a unique sequential number). Functions are then encoded in one of the following ways (element identifiers are used where applicable):

1. If a function is part of the user input, (e.g. the function which assigns ECUs to subsystems), a constant for each element of its domain is used, e.g.
   `define PROC0ToSubsystem=1;`
   which means that ECU e_0 belongs to subsystem sub_1

2. Non-boolean functions which are part of the pre-allocation are modeled as integer variables, e.g. for the function assigning ECU-types to ECUs (see Section 4.2.8), the following variable is declared
   `int [0,3] PROC0ToType;`
   which allows any ECU-type $\{t_0, t_1, t_2, t_3\}$ to be assigned to ECU e_0

3. Boolean functions which are part of the pre-allocation are encoded as one boolean variable for each element of the function's domain, e.g. for the boolean function which models whether or not a message has to be on the bus (see Section 3.1.4.8), variables are defined, such as
   `boole message0HasToBeOnBus;`

The HySAT definitions and variables are then used in several HySAT expressions:

1. Limitations for the values a function can get, e.g. if only ECU-type t_0 or t_1 are acceptable as the type of ECU e_0 (see Definition 3.17), the following expression would be generated:
   `(PROC0ToType = 0) or (PROC0ToType = 1);`

2. Implications are used to describe the dependencies between decision variables and intermediate variables, e.g. the cost of choosing ECU-type t_0 for ECU e_0 can be encoded like this:
   `(PROC0ToType=0) -> (PROC0Cost=19);`

3. Linear (integer) expressions are used to formulate e.g. the objective function:
   `Cost = (PROC0Cost) + (PROC1Cost)`

4.4. Optimization based on Linear Programming

4.4.1. Helpful Definitions

For a given DSE problem p with set of ECUs E and a configuration *conf* with task allocation function taskalloc_{conf}, the set of empty (aka unused) ECUs is defined as:

$$E^{\texttt{empty}}_{conf} := E \setminus \text{dom}(\text{taskalloc}_{conf})$$

The set of ECUs onto which at least one task is allocated is defined as:

$$E^{\texttt{NotEmpty}}_{conf} := \text{dom}(\text{taskalloc}_{conf})$$

4.4.2. Encoding the Problem

In the following subsections the complete encoding of the global analysis approach as mixed integer linear program is given. First all constraints are specified, then the objective function with its different parts is described.

4.4.2.1. ECU Types

$$\sum_{t \in \mathbb{E}} x^{\texttt{type}}_{e,t} \leq 1 \qquad \forall e \in E \tag{4.6}$$

$$\sum_{t \in \mathbb{E}} x^{\texttt{type}}_{e,t} \geq 0 \qquad \forall e \in E^{\texttt{empty}}_{conf} \tag{4.7}$$

$$\sum_{t \in \mathbb{E}} x^{\texttt{type}}_{e,t} \geq 1 \qquad \forall e \in E^{\texttt{NotEmpty}}_{conf} \tag{4.8}$$

$$x^{\texttt{type}}_{e,t} \in \{0,1\} \qquad e \in E, t \in \mathbb{E} \tag{4.9}$$

The binary variable $x^{\texttt{type}}_{e,t}$ is 1 if ECU-type t is assigned to ECU e, 0 otherwise. Equation (4.6) make sure that maximal one ECU-type is chosen per ECU. Equation (4.7) ensures that to all non-empty ECUs an ECU-type is assigned, while Equation (4.8) allows empty ECUs to have no ECU-type.

4.4.2.2. Task Allocation

$$\sum_{sub \in Sub} y^{\texttt{subsys}}_{\tau,sub} = 1 \qquad \forall \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \tag{4.10}$$

$$\sum_{t \in \mathbb{E}} y^{\texttt{type}}_{\tau,t} = 1 \qquad \forall \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \tag{4.11}$$

$$y^{\texttt{subsys}}_{\tau,sub} \in \{0,1\} \qquad \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}, sub \in Sub \tag{4.12}$$

$$y^{\texttt{type}}_{\tau,t} \in \{0,1\} \qquad \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}, t \in \mathbb{E} \tag{4.13}$$

The binary variable $y^{\texttt{subsys}}_{\tau,sub}$ is 1 if task τ is allocated to subsystem sub, 0 otherwise. Analogously the binary variable $y^{\texttt{type}}_{\tau,t}$ is 1 if task τ is assigned ECU-type t, 0 otherwise. By Equation (4.10) it is enforced that every unallocated task is allocated to exactly one subsystem and by Equation (4.11) that exactly one ECU-type is chosen for each unallocated task.

4.4.2.3. Constraints for Task Allocation

Let

$$constr = (\psi^{\texttt{ECUs}}, \psi^{\texttt{types}}, \psi^{\texttt{never}}.\psi^{\texttt{always}}, \psi^{\texttt{allowedSubsys}}, \psi^{\texttt{sameSubsys}}, \psi^{\texttt{diffSubsys}}, \psi^{\texttt{onBus}})$$

denote the constraint specification tuple of DSE problem p. As stated before, the constraints specifying the sets of allowed ECUs per task cannot be encoded due to the chosen level of abstraction of the global analysis approach. The same is true for constraints enforcing tasks not to be on the same ECU.

$$\sum_{t \in \psi^{\texttt{types}}(\tau)} y^{\texttt{type}}_{\tau,t} = 1 \qquad \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \tag{4.14}$$

$$\sum_{sub \in \psi^{\texttt{allowedSubsys}}(\tau)} y^{\texttt{subsys}}_{\tau,sub} = 1 \qquad \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \tag{4.15}$$

$$\sum_{\tau \in X} y^{\texttt{subsys}}_{\tau,sub} \leq 1 \qquad \forall X \in \psi^{\texttt{diffSubsys}}, sub \in Sub \tag{4.16}$$

$$y^{\texttt{subsys}}_{\tau,sub} \cdot (|X| - 1) - \sum_{\tau \in X \setminus \{\tau_0\}} y^{\texttt{subsys}}_{\tau,sub} = 0 \qquad \forall sub \in Sub, \quad \forall X = \{\tau_0, \ldots, \tau_n\} \in \psi^{\texttt{sameSubsys}} \tag{4.17}$$

Enforced by Equation (4.14), for each unallocated task exactly one ECU-type contained in the set of allowed ECU-types is chosen. In combination with the constraints applied for the ECU-type choice for unallocated tasks this constraint excludes all ECU-types not contained in the set of allowed types from being chosen. In a similar manner, the Equation (4.15) enforces each unallocated tasks to be allocated to one of the allowed subsystems specified for that task. In Equation (4.16) it is encoded that of each set of tasks which must not be allocated to the same subsystem, maximal one task is allocated to each of the subsystems. The case that certain tasks always have to be allocated to the same subsystem is handled by Equation (4.17). This equation is satisfied if either no task of such a set is allocated to a given subsystem or all of them.

4.4.2.4. Utilization

$$z_{\tau,sub,t} \geq y^{\texttt{subsys}}_{\tau,sub} + y^{\texttt{type}}_{\tau,t} - 1 \qquad \forall \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}, sub \in Sub, t \in \mathbb{E} \tag{4.18}$$

$$z_{\tau,sub,t} \leq y^{\texttt{subsys}}_{\tau,sub} - y^{\texttt{type}}_{\tau,t} + 1 \qquad \forall \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}, sub \in Sub, t \in \mathbb{E} \tag{4.19}$$

$$z_{\tau,sub,t} \leq y^{\texttt{type}}_{\tau,t} - y^{\texttt{subsys}}_{\tau,sub} + 1 \qquad \forall \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}, sub \in Sub, t \in \mathbb{E} \tag{4.20}$$

$$\sum_{e \in E_{sub}} \mathrm{utilavail}^{E,\mathbb{E}}(conf, e, t)\, x^{\texttt{type}}_{e,t} - \sum_{\tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}} \mathrm{util}(p, \tau, t) \cdot z_{\tau,sub,t} \geq 0 \qquad \forall sub \in Sub, t \in \mathbb{E} \tag{4.21}$$

$$z_{\tau,sub,t} \in \{0, 1\} \qquad \forall sub \in Sub, t \in \mathbb{E} \tag{4.22}$$

The binary variable $z_{\tau,sub,t}$ is 1 if a task τ is allocated to subsystem sub and to ECU-type t (without loss of generality), 0 otherwise, as encoded by Equations (4.18)–(4.22).

4.4.2.5. Memory

The handling of the memory constraints for ECUs is very similar to that of task utilization requirements.

$$\sum_{e \in E_{sub}} \mathrm{memavail}^{E,\mathbb{E}}(conf, e, t)\, x^{\texttt{type}}_{e,t} - \sum_{\tau \in \mathcal{T}^{\texttt{unalloc}}_{conf}} \mathrm{memreq}_p(\tau, t) \cdot z_{\tau,sub,t} \geq 0 \qquad \forall sub \in Sub, t \in \mathbb{E} \tag{4.23}$$

Equation (4.23) ensures, that enough memory is available in each subsystem and each ECU-type bin.

4.4.2.6. Global Communication

Encoding the constraints for global communication is slightly more complicated as many different cases have to be handled. Fortunately based on the formalization in

Section 4.2.3.3 it is sufficient to focus on those signals where it is still undecided whether or not they have to be sent over the global bus.

Let $S^{\texttt{maybe}}_{conf}$ be the set of signals for which it is still undecided whether or not they must have messages on the global bus. A given signal $s \in S^{\texttt{maybe}}_{conf}$ has to be transmitted over the global bus without loss of generality, iff

1. the sender task and at least one receiver task are pre-allocated to different subsystems
2. its sender task is already allocated to a subsystem and one the receiver tasks is pre-allocated to a different subsystem
3. one receiver task is already allocated to a subsystem and the sender task is pre-allocated to a different subsystem

While the first case is obvious, the other two cases stem from the decision not to include already allocated tasks into the MILP encoding directly. The knowledge about already allocated tasks which are involved in the sending/receiving of signal s is therefore included into the MILP model by directly encoding the cases as MILP formulas, such as "if the sender task or one receive tasks of signal s is allocated to a different subsystem than subsystem sub_1, the signal has to be transmitted over the global bus".

Let therefore

$$\mathcal{T}^{\texttt{recv}}_{tn,s} := \text{recv}(tn, s)$$

denote the set of tasks for a given task network tn receiving signal s and

$$\mathcal{T}^{\texttt{unalloc,recv}}_{conf,s} := \mathcal{T}_{tn}, s \cap \mathcal{T}^{\texttt{unalloc}}_{conf}$$

the set of unallocated receiver tasks of signal s where tn is the task network of DSE problem p. Furthermore let

$$\mathcal{T}^{\texttt{unalloc}}_{conf,s} := \left(\mathcal{T}_{tn}, s \cup \{\tau^{\texttt{send}}_{tn} s\}\right) \cap \mathcal{T}^{\texttt{unalloc}}_{conf}$$

be the set of all unallocated tasks sending or receiving signal s.

Firstly, new binary variables are introduced, one for each undecided signal:

$$b_s \in \{0, 1\} \qquad \forall s \in S^{\texttt{maybe}}_{conf} \tag{4.24}$$

If for a signal s the corresponding variable $b_s = 1$, then that signal has to be allocated onto the global bus, otherwise it must not be allocated to the global bus.

Two cases have to be distinguished: Let $S^{\texttt{none}}_{conf} \subseteq S^{\texttt{maybe}}_{conf}$ be the subset of signals where the sender and all receiver tasks are still unallocated. For all those signals the following constraints are added to the problem:

$$\left|\mathcal{T}^{\texttt{unalloc,recv}}_{conf,s}\right| \cdot b_s \geq \left|\mathcal{T}^{\texttt{unalloc,recv}}_{conf,s}\right| \cdot y^{\texttt{subsys}}_{\mathcal{T}^{\texttt{send}}_s,sub} - \sum_{\forall \tau \in \mathcal{T}^{\texttt{unalloc,recv}}_{conf,s}} y^{\texttt{subsys}}_{\tau,sub} \qquad \forall s \in S^{\texttt{none}}_{conf}, \forall sub \in Sub \quad (4.25)$$

$$b_s \leq 1 + \left|\mathcal{T}^{\texttt{unalloc,recv}}_{conf,s}\right| - y^{\texttt{subsys}}_{\mathcal{T}^{\texttt{send}}_s,sub} - \sum_{\tau \in \mathcal{T}^{\texttt{unalloc,recv}}_{conf,s}} y^{\texttt{subsys}}_{\tau,sub} \forall s \qquad \forall sub \in Sub \in S^{\texttt{none}}_{conf} \quad (4.26)$$

Equation (4.25) forces b_s to 1 if there is at least one receiver task is in a different subsystem than the sender task. In this case the right hand side would be larger than 0 for the subsystem where the sender task is pre-allocated to (and at least one receiver task is not). Equation (4.26) forces b_s to 0 exactly if there is one subsystem where the sender and all receiver tasks are allocated (only then the right hand side evaluates to 0).

Let $S^{\texttt{some}}_{conf} \subseteq S^{\texttt{maybe}}_{conf}$ be the subset of signals which sender task and/or some receiver tasks are allocated to a subsystem denoted by $sub^{\texttt{used}}_s$.

$$\left|\mathcal{T}^{\texttt{unalloc}}_{conf,s}\right| \cdot b_s \geq \sum_{sub \in Sub \setminus \left\{sub^{\texttt{used}}_s\right\}} \sum_{\tau \in \mathcal{T}^{\texttt{unalloc}}_{conf,s}} y^{\texttt{subsys}}_{\tau,sub} \qquad \forall s \in S^{\texttt{some}}_{conf} \quad (4.27)$$

$$\left|\mathcal{T}^{\texttt{unalloc}}_{conf,s}\right| \cdot b_s \leq \sum_{sub \in Sub \setminus \left\{sub^{\texttt{used}}_s\right\}} \sum_{\tau \in \mathcal{T}^{\texttt{unalloc}}_{conf,s}} y^{\texttt{subsys}}_{\tau,sub} + \left|\mathcal{T}^{\texttt{unalloc}}_{conf,s}\right| - 1 \qquad \forall s \in S^{\texttt{some}}_{conf} \quad (4.28)$$

Equation (4.27) forces b_s to 1 if at least task (receiver or sender) is in a different subsystem than $sub^{\texttt{used}}_s$. Equation (4.28) forces the value to be 0 if no other subsystem is used by any of the tasks (receiver or sender).

Based on the decision whether or not a signal has to have a message on the global bus the main constraint for the global bus capacity can be formulated:

$$slots^{\texttt{remain}}_{conf} \geq \sum_{s \in S^{\texttt{maybe}}_{conf}} b_s \cdot \text{slotsreq}(p, s) \quad (4.29)$$

The sum of all bus slots required for every signal for which the pre-allocation has decided that that signal has to have a message on the global bus, has to be smaller or equal to the number of still available bus slots on the global bus.

4.4.2.7. Deadline Synthesis

Several cases have to be handled separately for each end-to-end deadline and depending on whether or not the considered task is already allocated to an ECU.

Allocated Tasks Let $\mathcal{T}^{\texttt{alloc}}$ denote the set of tasks which are allocated by the current configuration but do not have initial deadlines. For task $\tau \in \mathcal{T}^{\texttt{alloc}}$ bounds for the synthesized deadline have to be established. The deadline of τ is denoted by $d_\tau^{\texttt{synth}}$ in this section.

In any case, the synthesized deadline has to be equal or larger than the chosen WCET, determined by the chosen ECU-type for that ECU and smaller or equal to the task's period:

$$d_\tau^{\texttt{synth}} + (1 - x_{e,t}^{\texttt{type}}) \cdot \text{period}^{\texttt{task}}(\tau) \geq \text{wcet}_p(\tau, t) \qquad \forall t \in \mathbb{E} \tag{4.30}$$

$$d_\tau^{\texttt{synth}} \leq \text{period}^{\texttt{task}}(\tau) \tag{4.31}$$

where e denotes the ECU where τ is allocated. Note that in Equation (4.30) the task's period is on the left hand side added for every ECU-type which has not been chosen for the ECU thus trivially satisfying the equation (as all WCETs are smaller or equal to the activation period).

If there is a task with a higher priority on the same ECU — denoted by $\tau^{\texttt{hp}}$ — than the deadline of this task — denoted by $d_{\tau^{\texttt{hp}}}$ — is the lower bound. The following additional constraint is added:

$$d_\tau^{\texttt{synth}} \geq d_{\tau^{\texttt{hp}}} \tag{4.32}$$

Note that while specifying the actual input for the MIP solver, one has to distinguish the case where $d_{\tau^{\texttt{hp}}}$ is fixed (in this case a constant is used), and the case where it is synthesized as well (in this case there is a variable for that deadline which has to be used in the constraint specification).

If there is a task with a lower priority on the same ECU — denoted by $\tau^{\texttt{lp}}$ — then its deadline — denoted by $d_{\tau^{\texttt{lp}}}$ — is the upper bound for $d_\tau^{\texttt{synth}}$. If the deadline of the lower priority task is synthesized, than no constraint has to be added because than the relation between the two deadlines is specified when $\tau^{\texttt{lp}}$ is handled (see above case). If the deadline is fixed, the following constraint is added:

$$d_\tau^{\texttt{synth}} \leq d_{\tau^{\texttt{lp}}} \tag{4.33}$$

Unallocated Tasks Let $\mathcal{T}^{\texttt{unalloc}}$ denote the set of tasks which are unallocated in the current configuration and do not have initial deadlines. For task $\tau \in \mathcal{T}^{\texttt{unalloc}}$ bounds for the synthesized deadline have to be established. The deadline of τ is denoted by $d_\tau^{\texttt{synth}}$ in this section.

The bounds for unallocated tasks are calculated very similar to those of allocated tasks with the obvious differences that no higher or lower priority tasks have to be considered. Additionally the WCET is determined for unallocated tasks based on their pre-allocation.

$$d_\tau^{\mathtt{synth}} + (1 - y_{\tau,t}^{\mathtt{type}}) \cdot \mathrm{period}^{\mathtt{task}}(\tau) \geq \mathrm{wcet}_p(\tau, t) \qquad \forall t \in \mathbb{E} \tag{4.34}$$

$$d_\tau^{\mathtt{synth}} \leq \mathrm{period}^{\mathtt{task}}(\tau) \tag{4.35}$$

Signals Let $S^{\mathtt{undef}}$ denote the set of all signals without fixed deadlines for the given DSE problem. For any signal s let $d_s^{\mathtt{synth}}$ denote its synthesized signal.

Again, several cases have to be distinguished: For every signal which is already allocated the global bus, the deadline is set to its period multiplied by three. For every signal which is known to be subsystem-local, the deadline is set to the signal's period. For every signal which is local to one ECU the deadline is set to 0. It remains the set of signals which might be on the global bus $S^{\mathtt{maybe}}$. The following constraints are added:

$$d_s^{\mathtt{synth}} = \mathrm{period}^{\mathtt{sig}}(s) + 2 \cdot b_s \cdot \mathrm{period}^{\mathtt{sig}}(s) \qquad \forall s \in S_{conf}^{\mathtt{maybe}} \tag{4.36}$$

Each signal has at least a synthesized deadline equal to its period. In case it has to be allocated to the global bus, the period multiplied by two is added.

Enforcing End-to-end Deadlines Let (again) denote $\mathcal{T}_d^{\mathtt{undef}}$ the set of all tasks without fixed deadlines affected by end-to-end deadline d. Furthermore let $S_d^{\mathtt{undef}}$ denote the set of signals without a fixed deadline affected by the end-to-end deadline.

$$d_d^{\mathtt{remain}} \geq \sum_{\tau \in \mathcal{T}_d^{\mathtt{undef}}} d_\tau^{\mathtt{synth}} + \sum_{s \in S_d^{\mathtt{undef}}} d_s^{\mathtt{synth}} \qquad \forall d \in \mathrm{dom}(\mathrm{deadline}^{\mathsf{e2e}}) \tag{4.37}$$

where $d_d^{\mathtt{remain}}$ denotes the remaining fraction of the end-to-end deadline (not consumed by already fixed local deadlines) as in Equation (4.2.5.1) on page 70.

Task Penalty Variables For each task without a fixed deadline a target deadline can be calculated as defined in Section 4.2.5.1 on page 69 before running the MIP optimization. For each of those tasks a penalty variable $p_\tau^{\mathtt{td}}$ is defined:

$$p_\tau^{\mathtt{td}} \geq d_\tau^{\mathtt{target}} - d_\tau^{\mathtt{synth}} \qquad \forall \tau \in \mathcal{T}^{\mathtt{undef}} \tag{4.38}$$

$$p_\tau^{\mathtt{td}} \in \mathbb{N} \qquad \forall \tau \in \mathcal{T}^{\mathtt{undef}} \tag{4.39}$$

Note that Equation (4.38) gives only a lower bound for penalty variables. An upper bound is not required because the variables appear in the objective function. As the objective function is minimized, each penalty variable is assigned the smallest possible value anyway.

4.4.2.8. Optimization Objective

The objective function is the sum of one or more of the following sub-objectives. The sum is minimized by the MILP solver using the standard Simplex method and a branch and bound method for integer variables. The minimization of the total hardware cost defined as the sum of the used ECU-types is seen as the primary objective which is mandatory. Other objectives are optional, e.g. if the minimization of the number of used bus slots on the global bus is not required, this part of the objective function can simply be deactivated. The additional cost introduced by secondary objectives are also called penalties.

To achieve a ranking between the parts of the objective function which prioritizes the primary objective over the secondary objectives, a constant offset is added to the cost specified for each single ECU-types. This offset is an upper bound for the sum of all penalties of secondary objectives. Thus solutions with lower hardware cost always supersede solutions with higher hardware cost but lower global bus usage.

Penalties used in Deadline Synthesis

$$\sum_{\tau \in \mathcal{T}^{\texttt{undef}}} p_\tau^{\texttt{td}} \tag{4.40}$$

In Equation (4.40) the sum of all penalties caused by tasks having a too small synthesized deadline compared to their target deadline is calculated. An approximating upper bound for the sum of these penalties can be calculated as the sum of all activation periods of all affected tasks.

Minimal Number of Bus Slots

$$\sum_{s \in S_{conf}^{\texttt{maybe}}} b_s \cdot \text{slotsreq}(p, s) \tag{4.41}$$

In Equation (4.41) the sum of the number of required bus slots for each signal which is allocated to the bus due to the pre-allocation (not a-priori) is calculated. An upper bound for the sum of these penalties is simply the number of still available bus slots on the global bus.

Minimal Hardware Cost This is the primary objective function. Offset c represents the upper bound for all penalties by secondary objectives as stated above. It is added to the cost of each ECU-type thus ensuring that the primary objective dominates the secondary ones.

$$\sum_{e \in E} \sum_{t \in \mathbb{E}} (c + \text{cost}(t)) x_{e,t}^{\texttt{type}} \tag{4.42}$$

In Equation (4.42) the sum of the costs (including the offset mentioned above) for each ECU-type used in the pre-allocation is calculated.

4.5. Other Approaches

Currently there exists only one alternative approach for the global analysis problem.

4.5.1. Global Analysis via Graph Partitioning: The KL Approach

The alternative global analysis approach described in this section is based on graph partitioning. An overview and a comparison of the performance of that algorithm with other graph partitioning algorithms is published in [Bük+11a], [Bük+11b] and will be published in [Bük12], too.

The Kernighan-Lin Algorithm [KL70] has been combined with concepts taken from [Dut93] and further extended to support multiple partitions. The resulting algorithm is called *KL+*. Each hardware subsystem is represented by one partition containing software task nodes of the task network. Each partition initially contains all tasks allocated to its corresponding subsystem. All allocated tasks must remain in their partitions. The algorithm iteratively assigns/reassigns each unallocated task to one of the partitions until the minimal cut for the task network (graph) has been found. In contrary to the classical KL algorithm the objective function is not the sum of the weight of the edges (here: signals) but the sum of the costs for allocating the tasks to the partitions.

The cost for allocating a task to a partition is based on the communication cost and the capacity required by the task in conjunction with the remaining capacity on the corresponding hardware subsystem. The communication cost is calculated by looking at the communication between tasks and the allocation of those tasks: intra-partition communication (tasks are in the same partition) is for free, inter-partition communication is expensive (tasks are in different partitions) because the global bus has to be used for that. The way two tasks are communicating with each other can be characterized even more detailed by expressing how frequently the communication occurs and how much data is transferred. The capacity required for allocating a task onto an ECU depends on the type of that ECU. This information is not available to the KL+ algorithm because all modifications to the hardware architecture are solely realized on the local analysis tier and — in contrary to the global analysis approach presented in this thesis — *KL+* does not use a detailed prediction of the ECU-types the task might be running on. Instead, the algorithm determines separately for each hardware subsystem the best ECU-type available for that subsystem (constraints might forbid certain types per subsystem) and calculates the required capacity of all task on that subsystem (in that partition) assuming that the task will run on an ECU with that ECU-type.

Due to that abstraction of the real capacity (available and required) the approach often underestimates the available capacity and therefore provides very tight hardware cost limits to the subsystem-specific local analysis steps. But this is very well acceptable and even desired because it will prevent that the local analysis invests huge amounts of money on hardware modifications just to be able to allocate all the task pre-allocated to the respective subsystem. In an ideal global analysis run, all tasks would be allocated during the first iteration already. If this is impossible due to the tight cost restrictions, some tasks will remain unallocated after the first global iteration and are then pre-allocated to other subsystems during the second iteration, while in the same time the capacity calculations are revised and the cost restrictions are relaxed if necessary.

The KL approach is completed by an algorithm for calculating deadlines for tasks and signals from end-to-end deadlines and a heuristic for calculating static schedules for the global bus. For the evaluation in Chapter 6 a simplified heuristic has been implemented which does not calculate complete schedules but only estimates the minimal number of required bus slots for all messages on the global bus and checks this value against the number of still available bus slots. This simplified heuristic is used during the experiments to ensure that all approaches under evaluation solve the same problem. If using the original heuristic the KL approach would be handicapped (because it would calculate more detailed solutions than all the other approaches).

5. Local Analysis

All the approaches for global analysis described in chapter 4 use the quite abstract notion of **utilization** for characterizing the computation capacity provided by ECUs or even by whole subsystems. Moreover, local buses contained in the hardware subsystems are not considered during the global analysis at all. Handling those details intentionally is delegated to the subsystem-specific local analysis steps. The local analysis approaches are required to work on a sufficiently detailed level of abstraction such that they can guarantee the correctness of their local results with respect to all explicitly specified constraints and all implicit constraints such as schedulability of all ECUs in the subsystem. As depicted in Figure 5.1 the local analysis represents the lower tier of the two-tier optimization approach described in this thesis.

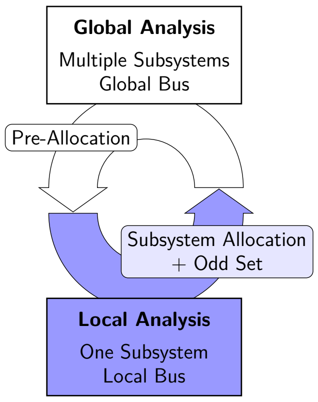


Figure 5.1. – Optimization Process Cycle: Local Analysis

A great advantage of the two-tier optimization approach is that the same global analysis approach can use different local analysis approaches depending on the technical realization in the target system. One local analysis approach might for example provide support for ECUs using time-triggered schedulers, a second local analysis approach might provide support for classical fixed-priority preemptive schedulers, and a third approach might incorporate the concepts of the other two by providing support for arbitrary combining ECUs with time-slice-based or FPS schedulers in one subsystem. Currently there is only support for homogeneous subsystems where all ECUs are FPS-scheduled and one priority-based subsystem-local bus (using CAN) exists.

This chapter first provides an informal definition of the local analysis problem, followed by a mathematical definition of the "interface" between the global and the local analysis. Then the local analysis problem is formally defined and finally features of the existing different local analysis approaches are compared. The emphasis of this chapter is on the description of the spare-time and MaxWCET approach for local analysis which has been developed as part of this thesis.

5.1. Problem Definition

The local analysis is responsible for optimizing one subsystem without considering any of the other subsystems. Therefore the local analysis works only with those ECUs which are part of that subsystem and with the subsystem-local communication bus.

At the beginning of each local analysis run a set of (currently unallocated) tasks which have been pre-allocated to the subsystem to be optimized is provided by the global analysis. Furthermore, a set of constraints (including an upper bound for the hardware cost of the subsystem) and the current global partial allocation of tasks and signals is provided (containing those tasks/signals which have been allocated initially or during earlier optimization steps). Note that each local analysis run has access to all information about the other subsystems. But this information is only used to assess the communication constraints, e.g. if a pre-allocated task sends a signal to a task allocated or pre-allocated to a different subsystem, allocating that task to the subsystem to be optimized implies that one or more messages on the global bus (and potentially on the local bus) are required.

The goal of the local analysis is to find a solution where as many pre-allocated tasks are allocated to the subsystem's ECUs as possible while in the same time assure that the hardware cost caused by the necessary hardware modifications (if any) are not exceeding the specified cost limit. Some approaches are even able to minimize these hardware cost.

The objective functions of all local analysis approaches have in common that pre-allocated tasks are only left unallocated if they cannot be allocated even when using the most expensive hardware configuration for the respective subsystem (while still satisfying the specified cost limit). Without this important property, the local analysis approaches would frequently leave software tasks unallocated in order to minimize the subsystem's hardware cost. Of course, this is not desired.

The result of every local analysis step is a feasible allocation of tasks to ECUs, an assignment of messages to signals, and an allocation of messages to the subsystem-local bus. In this context "feasible" means that all subsystem ECUs and the local bus are schedulable, that the memory consumption does not exceed the available memory, that all signals sent between ECUs of the subsystem have corresponding messages on the local bus and that all user-specified constraints are satisfied.

It is required that the allocation of all tasks and message to the subsystem's ECUs and its local bus as provided by the global analysis is not changed during the local analysis but solely extended (e.g. an already allocated task has to remain on its ECU, but a pre-allocated tasks can be allocated to any ECU of the respective subsystem). Currently this requirement does not allow backtracking in the sense that an already allocated task is removed from its ECU. Backtracking in this two-tier approach currently means that tasks which could not be allocated to a subsystem form the so-called odd set and are handed back to the global analysis for re-allocation. Another requirement is that the cost limit specified for the local hardware subsystem is not exceeded. Changes to other subsystems are strictly prohibited.

Several different algorithms have been developed for the local analysis optimization. All of them support at least the properties informally described in the last paragraphs. In the following sections a formal definition of the local analysis problem is provided starting with the interface between the global and the local analysis.

5.1.1. The Interface Between Global and Local Analysis

This section first introduces the formal definition of the local analysis problem. Then a subsystem-local view on the whole system is defined.

Definition 5.1 (Local Analysis Problem)
Let conf be a valid (see Definition 3.18) and schedulable (see Definition 3.20) configuration for a DSE problem p. A local analysis problem for a subsystem sub contained in the hardware architectural pattern of the DSE problem is defined as

$$p_{sub} = (\mathit{conf}, \mathcal{T}^{\mathtt{pre}}, \mathit{cost}^{\mathtt{max}}, \mathrm{penalty})$$

where

- $\mathcal{T}^{\mathtt{pre}} \subseteq \mathcal{T}$ *is a set of currently unallocated tasks which have been pre-allocated to the subsystem by the global analysis*
- $\mathit{cost}^{\mathtt{max}}$ *is an upper bound on the hardware architecture cost given by the global analysis.*
- *Function*
$$\mathrm{penalty}\colon \mathcal{T}^{\mathtt{pre}} \to \mathbb{N}$$
assigns a penalty cost to each of the pre-allocated tasks which incurs if the task remains unallocated after the local analysis step.

It is required that the current configuration given as input to the local analysis is a valid subsystem-local solution which does not exceed the specified cost limit. Therefore the trivial solution for a local analysis problem — which is to provide the unchanged input as result to the global analysis — is a valid solution for every local analysis problem. □

For describing the interface of the local analysis step a local view on the global problem and on solution candidates is required. In the following, definitions specific to local analysis are derived from the definitions for the overall DSE problem by projection on one subsystem.

Definition 5.2 (Hardware Subsystem Projection)
For a given architecture graph $G^{\mathtt{hw}} = \left(E \mathbin{\dot\cup} B, \mathit{Edges}^{\mathtt{hw}}\right)$ *(see Definition 3.2) the set of local ECUs* E_{sub} *for a given subsystem sub contained in the architecture graph is defined as*

$$E_{sub} := \left\{ e \mid e \in E \ \wedge\ \mathrm{subsys}(G^{\mathtt{hw}}, e) = \mathit{sub} \right\}$$

where subsys *is the subsystem allocation function.*

The set of local buses is defined as follows:

$$B_{sub} := \left\{ b \mid b \in B \ \wedge \ \text{subsys}(G^{\texttt{hw}}, b) = sub \right\}$$

where subsys *is the subsystem allocation function.*

A well-formed hardware architecture has only maximal one local bus b_{sub} per subsystem. The existence of a local bus is mandatory for all subsystems with more than one ECUs. If a local bus exists in a given subsystem sub, it is denoted by b_{sub} with

$$\begin{aligned} & b_{sub} \in B \textit{ is the (only) local bus of subsystem sub} \\ :\Longleftrightarrow \ & \text{subsys}(G^{\texttt{hw}}, b) = sub \ \wedge \\ & \forall b \in B : (\text{subsys}(G^{\texttt{hw}}, b) = sub) \Rightarrow (b = b_{sub}) \end{aligned}$$ □

Definition 5.3 (Subsystem Task Set)
For a given local analysis problem

$$p_{sub} = (conf, \mathcal{T}^{\texttt{pre}}, cost^{\texttt{max}}, \text{penalty})$$

the set of subsystem tasks is defined as

$$\mathcal{T}_{sub} := \mathcal{T}^{\texttt{pre}}_{sub} \ \cup \ \{\tau \mid \tau \in \text{dom}(\text{taskalloc}_{conf}) \ \wedge \ \text{taskalloc}_{conf}(\tau) \in E_{sub}\}$$

where $\mathcal{T}^{\texttt{pre}}_{sub}$ is the set of tasks pre-allocated to the subsystem, taskalloc_{conf} *is the current intermediate task allocation function and E_{sub} is the set of ECUs in subsystem sub.* □

Definition 5.4 (Subsystem Signal Set)
For a local analysis problem p_{sub} with task set $\mathcal{T}_{sub}$ the set of subsystem signals is defined as

$$S_{sub} := \{s \mid \text{sender}(tn, s) \in \mathcal{T}_{sub} \ \vee \ \text{recv}(tn, s) \subseteq \mathcal{T}_{sub}\}$$

where $\mathcal{T}_{sub}$ is the subsystem task set and sender *and* recv *are the sender and receiver functions, respectively.* □

Note that the subsystem signal set definition hides exactly those signals which are sent exclusively between tasks allocated or pre-allocated to other subsystems.

Definition 5.5 (Subsystem Configuration)
For any subsystem sub and a local analysis problem

$$p_{sub} = (conf, \mathcal{T}^{\texttt{pre}}, cost^{\texttt{max}}, \text{penalty})$$

a subsystem configuration is defined as

$$conf_{p,sub} = \left(\text{type}_{sub}, \mathcal{A}^{\texttt{subsys}}_{sub}, \mathcal{T}^{\texttt{odd}}_{sub}\right)$$

where type_{sub} *assigns ECU-types to some or all ECUs contained in the subsystem.* $\mathcal{A}_{sub}^{\mathsf{subsys}}$ *is an allocation of tasks, signals and messages as defined in Definition 3.6 but limited to hardware elements of the current subsystem. Note that an allocation resulting from a particular local analysis step contains all tasks, signals and message that have already been allocated to the subsystem prior to that local analysis step and additional tasks from the set of tasks* $\mathcal{T}_{sub}$ *which have been pre-allocated to the subsystem and their signals and potential new messages as long as they are required.*

The set $\mathcal{T}_{sub}^{\mathsf{odd}}$ *(called* ***odd set****) contains all pre-allocated tasks which could not be allocated during the local analysis step, formally:*

$$\mathcal{T}_{sub}^{\mathsf{odd}} := \mathcal{T}_{sub} \setminus \text{dom}(\text{taskalloc}_{sub})$$

where $\text{taskalloc}_{\mathcal{A}_{sub}^{\mathsf{subsys}}}$ *is the allocation of tasks onto subsystem ECUs result from the local analysis step.* □

Definition 5.6 (Subsystem Cost Function)
The subsystem cost function is equal to the global one given in Definition 3.22 with the only difference that the set of ECUs is limited to the subsystem.

$$cost(conf_{p,sub}) := \sum_{e \in E_{sub}} \text{cost}(\text{type}_{sub}(e))$$

for a given subsystem sub and a subsystem configuration $conf_{p,sub}$*.* □

Definition 5.7 (Subsystem Solution)
A subsystem solution is a schedulable subsystem configuration. As local analysis steps are usually intermediate steps towards a global solution the subsystem solution is not required to be complete in the sense that all pre-allocated tasks in $\mathcal{T}_{sub}$ *have to be allocated during the local analysis step.* □

5.2. Spare-Time and MaxWCET Analysis

The spare-time/MaxWCET analysis solves the Local Analysis Problem as defined in Section 5.1. In addition to satisfying the constraint that the subsystem hardware cost must be less than or equal to the specified cost limit, the spare-time/MaxWCET approach is capable of minimizing the subsystem hardware cost.

5.2.1. The Concept of Spare-Time/MaxWCET Analysis

Additional tasks which have been pre-allocated to a subsystem and the messages which might be required for their signals can be allocated to the subsystem only if there is enough free “computation capacity” and communication bandwidth. If there is not enough computation capacity for one or more of the pre-allocated tasks, it might be necessary to extend the system by modifying the ECU-types of ECUs, which might or might not already have an ECU-type assigned. Alternatively some or all of those tasks may also be put in the odd set and remain unallocated. If a signal does not fit onto the local bus, modifying the hardware currently is no option. Instead, the sender task and/or the receiver tasks of that signal have to be allocated onto the hardware architecture such that no local communication is required. This might even require to leave them unallocated thus handing them back to the global analysis by adding them to the odd set. The computation capacity of ECUs and the bandwidth of the local bus are shared among all tasks/signals allocated to them.

Many modeling concepts used in Chapter 4 (“Global Analysis”) can be adapted to fit the local analysis problem, e.g. the cost function and the assignment of ECU-types to ECUs. But one important concept is still missing: A notion of “capacity” regarding the schedulability of FPS-scheduled ECUs and CAN buses which facilitates the definition of a simple but sufficient schedulability test.

The spare-time/MaxWCET approach presented in this chapter provides simple and sufficient schedulability tests for FPS-scheduled ECUs with strictly periodic tasks and for CAN buses with strictly periodic signals/messages. Local deadlines are mandatory for all tasks and signals/messages. The deadlines have to be less than or equal to the periods of the tasks/signals.

The approach is based on the following observations. The characterization of remaining capacity has to cover two effects. Firstly, inserting a new task/signal with a higher priority than some existing tasks/signals on the same ECU will disturb those already allocated tasks/signals. When performing the schedulability analysis for each lower-priority task/signal, the additional interruptions caused by adding the new tasks/signals have to be considered.

Secondly, the additionally allocated tasks/signals are themselves interrupted by any previously allocated task/signals with higher priority on the same ECU/bus. Of course the worst case response time of those additionally allocated tasks/signals may not exceed their deadlines, too.

The first problem is tackled by **spare-time Analysis**. The **spare-time** characterizes the remaining computation capacity remaining on a given ECU for a given priority such that all previously allocated tasks/signals remain schedulable. The second problem is tackled **MaxWCET Analysis**. The **MaxWCET** value characterizes how much capacity is still available with respect to multiple specific combinations of task parameters (deadline, period, etc.). This enables the speculative pre-calculation of MaxWCET values without knowing yet which tasks/messages will be allocated to which ECUs/to the local bus in the following analysis phase.

The combination of spare-time and MaxWCET allows to predict whether or not a set (or subset) of currently unallocated tasks/message can be allocated to a hardware subsystem such that all tasks/messages are schedulable without performing the schedulability described in Chapter 2, Section 2.2.5.3 on page 18.

Figure5.2 depicts the process used for applying spare-time/MaxWCET analysis. An initial schedulability test ensures the precondition that the initial configuration is schedulable. Then spare-time values are calculated followed by a calculation of MaxWCET values. Based on the pre-calculations the optimization process is performed.

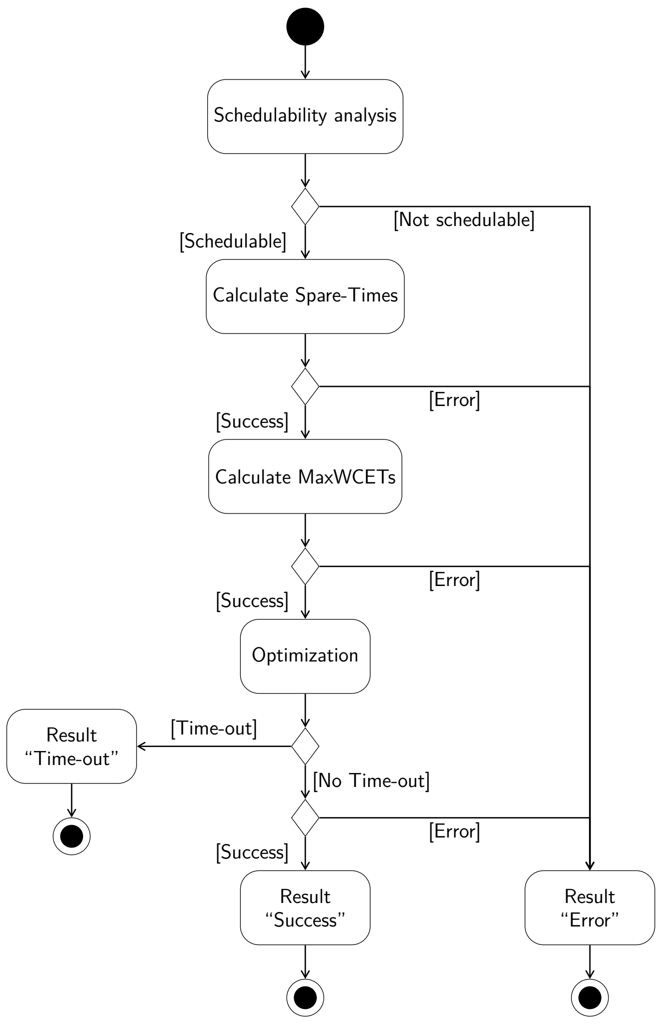


Figure 5.2. – Spare-Time/MaxWCET Local Analysis Process

5.2.2. Limitations and Preconditions of the Approach

There are some limitations of the current Spare-Time/MaxWCET approach. The notion of **limitation** is used to describe which invariant properties of the DSE problems are mandatory to be able to use the proposed approach. Example: The periods of tasks and signals remain unchanged during the whole optimization process.

Additional, DSE local analysis problems have to satisfy some preconditions for the spare-time / MaxWCET approach to be applicable on them. The notion of **precondition** is used to describe which variable properties the DSE problems and DSE local analysis problems must have to be compatible to the proposed methods. Examples: Not all local deadlines of tasks must be known initially. But it is a precondition of the spare-time/MaxWCET local analysis that all still missing local deadlines are synthesized by the global analysis step before running the spare-time/MaxWCET local analysis.

5.2.2.1. Limitation: Relay Tasks

If in a given task network two tasks communicate with each other via a signal, then their allocation to the hardware architecture has to guarantee that this communication is technically realizable. Now imagine that the first task is allocated to a non-gateway ECU of one hardware subsystem and the other task is allocated to a different subsystem. In this case using only one message for that signal is impossible because the signal has to be sent at least over the local bus of the subsystem where the first task is allocated and over the global bus. At least two messages are required. But what happens on the gateway ECU of the first task's subsystem? One possible solution is to use a relay task on the gateway ECU which relays that signal by (assuming the first task is the signal's sender) reacting on the corresponding message on the subsystem-local bus by sending the corresponding message on the global bus. Both the activation period and the deadline of such a relay task depend on the properties of the sender/receiver tasks and on the properties of the messages as well.

In this work I assume for simplicity, that signals/message are relayed not by special relay tasks but by the operating system on the gateway ECUs, transparently for the regular software tasks allocated to those gateway ECUs.

Originally, the idea was to explicitly model relay tasks running on each of the gateway ECUs with variable periods. Those periods would have to be adjusted during the optimization phase to the smallest possible period of all tasks which make use of the relay service. Unfortunately this concept has some serious implications: All local analysis approaches require that each hardware subsystem is schedulable initially. This includes the gateway ECUs with the relay tasks and additional tasks allocated to them. If a new task with a period smaller than all periods of the tasks already using the relay task is allocated to a subsystem, the period of the relay task would be reduced to that task's period. But reducing the period of the relay task might render tasks on the gateway ECU unschedulable which violates the local analysis precondition that the subsystem is schedulable. Therefore this concept has been dropped.

5.2.2.2. Limitation/Precondition: Deadlines Less or Equal Periods

For each task in the system the deadline has to be smaller or equal to its period. Also for each signal and each message the deadline has to be smaller than the period. Allowing deadlines larger than periods would lead to problems like overlapping instances of the same tasks/message which currently cannot be handled by the proposed approach.

5.2.2.3. Limitation: Strictly Periodic Activation of Tasks and Signals/Messages

Currently there is no support for release jitter in the approach. Supporting release jitter would require a holistic schedulability analysis (involving the whole system) which contradicts the concept of using pre-calculated spare-time / MaxWCET values which model the remaining capacity of each ECU and the local bus independently from other ECUs in the hardware subsystem.

Release jitter occurs if a task *arrives* (is able to run) but is not immediately *released* (placed in the set of runnable tasks by the operating system). Release jitter usually propagates along event-triggered task chains. Holistic schedulability analysis is required for handling systems with release jitter. *Holistic* schedulability analysis is used to calculate a fixed-point for the whole system because the ECUs cannot be analyzed independently from each other anymore. See [Tin96] for details.

5.2.2.4. Limitation/Precondition: Deadline Monotonic Priorities

For the Spare-Time/MaxWCET approach it is required, that the priorities of all tasks and messages are assigned following the deadline-monotonic paradigm (see also Section 2.2.5.1).

Furthermore, it is assumed that a total order on the set of tasks is given by the user by specifying a predicate:

$$\text{prio}^{>} : \mathcal{T} \times \mathcal{T} \rightarrow \{\texttt{false}, \texttt{true}\} \tag{5.1}$$

For any two tasks $\tau_i, \tau_j \in \mathcal{T}$ the predicate evaluating to `true` ($\text{prio}^{>}(\tau_i, \tau_j) = \texttt{true}$) implies that τ_i gets a higher priority than τ_j, but only if both tasks are allocated to the same ECU. The following properties have to hold for that predicate:

$$\forall \tau_1, \tau_2 \in \mathcal{T} : \left(\text{deadline}^{\texttt{task}}_{conf}(\tau_1) < \text{deadline}^{\texttt{task}}_{conf}(\tau_2)\right) \Rightarrow \\ \text{prio}^{>}(\tau_1, \tau_2) = \texttt{true} \tag{5.2}$$

$$\forall \tau \in \mathcal{T} : \text{prio}^{>}(\tau, \tau) = \texttt{false} \tag{5.3}$$

$$\forall \tau_1, \tau_2 \in \mathcal{T} : \text{prio}^{>}(\tau_1, \tau_2) = \texttt{true} \Rightarrow \text{prio}^{>}(\tau_2, \tau_1) = \texttt{false} \tag{5.4}$$

$$\forall \tau_1, \tau_2 \in \mathcal{T}, \tau_1 \neq \tau_2 : \text{prio}^{>}(\tau_1, \tau_2) = \texttt{true} \ \vee \ \text{prio}^{>}(\tau_2, \tau_1) = \texttt{true} \tag{5.5}$$

Equation (5.2) requires that for every two tasks with different deadlines the task with the lower deadline gets the higher priority. Equation (5.3) enforces the relation to be irreflexive and Equation (5.4) requires it to be asymmetric. Equation (5.5) states that if two tasks have equal deadlines than one of them has to be assigned a higher priority than the other. It is required to know the set of signals $S^{\texttt{alloc}}$ already allocated via messages to the local bus and the set of signals which might be allocated during the optimization step $\overline{S}$ before running the spare-time and MaxWCET pre-analysis steps. The reason is that because a higher priority message might be blocked by a lower priority message, upper bounds for those blocking times have to be calculated during the pre-analysis. As for tasks it is required to establish a total order with respect to the signal priorities for the set of signals $S = S^{\texttt{alloc}} \dot{\cup} \overline{S}$:

$$\text{prio}^{>} : S \times S \rightarrow \{\texttt{true}, \texttt{false}\} \tag{5.6}$$

Similar to the handling of priorities of tasks, if for any two signals $s_i, s_j \in S$ the predicate evaluates to true ($\text{prio}^{>}(s_i, s_j) = \texttt{true}$) then the message m_i corresponding to s_i would get a higher priority if allocated to the same (local) bus as message m_j corresponding to signal s_j. This additional implies that the deadline of m has to be smaller or equal to the deadline of message m_j because all priorities of local buses are defined deadline-monotonic.

A valid predicate definition has to satisfy the following properties:

$$\begin{aligned}
\forall s_1, s_2 \in S : \left(\text{deadline}^{\texttt{signal}}_{conf}(s_1) < \text{deadline}^{\texttt{signal}}_{conf}(s_2)\right) \Rightarrow& \\
\text{prio}^{>}(s_1, s_2) = \texttt{true}& \qquad (5.7)\\
\forall s \in S : \text{prio}^{>}(s, s) = \texttt{false}& \qquad (5.8)\\
\forall s_1, s_2 \in S : \text{prio}^{>}(s_1, s_2) = \texttt{true} \Rightarrow \text{prio}^{>}(s_2, s_1) = \texttt{false}& \qquad (5.9)\\
\forall s_1, s_2 \in S, s_1 \neq s_2 : \text{prio}^{>}(s_1, s_2) = \texttt{true} \lor \text{prio}^{>}(s_2, s_1) = \texttt{true}& \qquad (5.10)
\end{aligned}$$

In the following both predicates (for tasks and for signals) are assumed to be given.

5.2.3. Analysis of Electronic Control Units

First, the theory behind Spare-Time and MaxWCET analysis is presented with the focus on tasks and ECUs. In the next section the concepts are extended to signals/messages on CAN buses.

In the rest of this chapter the following notations are used without loss of generality: Let p be a DSE problem and *conf* a valid and feasible but incomplete configuration for that DSE problem with allocation $\mathcal{A}$. Furthermore let

$$p_{sub} = (conf, \mathcal{T}^{\texttt{pre}}, cost^{\texttt{max}}, \text{penalty})$$

be the local analysis problem to be analyzed.

5.2.3.1. Task Spare-Time Analysis

The spare-time is a notion for characterizing the remaining capacity of an ECU. Whenever a new task is allocated to an ECU with higher priority than an allocated task, that new task will cause additional preemptions for the allocated task. Therefore the worst case response time (WCRT) of the allocated task is calculated using fixed-point Equation (2.1) increases. The response time of the allocated tasks must not exceed that task's deadline, but it can be increased as long as it remains equal to or less than the task's deadline.

Let τ_i be a schedulable task allocated to an ECU e with ECU-type t. The maximum extent to which that task may be delayed during its execution by preemptions caused by additional tasks with higher priority on the same ECU without exceeding its deadline is called its Spare-Time $\mathfrak{s}_{\tau_i,t}$. Stated more formally, the Spare-Time is the greatest natural number for which there exists a fixed-point r_i for Equation (5.11) which is less than or equal to the task's deadline $\text{deadline}^{\texttt{task}}_{conf}(\tau_i)$.

Definition 5.8 (Task Spare-Time)
Let p be a DSE Problem and p_{sub} a local analysis problem as defined above. Let t be an ECU-type and e an ECU contained in the hardware architectural pattern of p. The spare-time $\mathfrak{s}_{\tau_i,t}$ of a task τ_i already allocated to that ECU is defined as

$$\begin{aligned}
&\mathfrak{s}_{\tau_i,t} \in \mathbb{N} \textit{ is the greatest natural number such that} \\
&\quad \exists r_i \in \mathbb{N} : \gamma_{p_{sub},\tau_i,t}(r_i) = r_i \wedge r_i \leq \text{deadline}^{\texttt{task}}_{conf}(\tau_i) \textit{ with} \\
\gamma_{p_{sub},\tau_i,t}(x) =& \text{wcet}_p(\tau_i, t) + \mathfrak{s}_{\tau_i,t} + \\
&\sum_{\tau_j \in \text{hp}_{\mathcal{A}}(\tau_i)} \left\lceil \frac{x}{\text{period}^{\texttt{task}}(\tau_j)} \right\rceil \text{wcet}_p(\tau_j, t)
\end{aligned} \tag{5.11}$$

where hp *denotes the function which for the given allocation assigns to any given task τ the set of all tasks which are allocated to the same ECU with a higher priority than τ.* □

Note that the existence of a Spare-Time $\mathfrak{s}_{\tau_i,t} \in \mathbb{N}$ follows from the precondition that all allocated tasks are schedulable.

Definition 5.9 (Family of Task Spare-Time Fixed-Point Equations)
By replacing the variable $\mathfrak{s}_{\tau_i,t}$ in Equation (5.11) by a parameter $z \in \{0, \dots, \mathfrak{s}_{\tau_i,t}\}$ a family of functions $\gamma^z_{p_{sub},\tau_i,t}$ is defined with:

$$\gamma^z_{p_{sub},\tau_i,t}(x) = \text{wcet}_p(\tau_i, t) + z + \sum_{\tau_j \in \text{hp}_{\mathcal{A}}(\tau_i)} \left\lceil \frac{x}{\text{period}^{\texttt{task}}(\tau_j)} \right\rceil \text{wcet}_p(\tau_j, t) \tag{5.12}$$

In the following it is shown that for every such function $\gamma^z_{p_{sub},\tau_i,t}$ a fixed-point exists. But first a sufficient condition for the existence of fixed-points in general is given in Lemma (5.10).

Lemma 5.10 (Sufficient Condition for the Existence of a Fixed-Point)
Let $f : \mathbb{N} \to \mathbb{N}$ be a monotonically increasing function. If for given $x, y \in \mathbb{N}, x < y$ holds that $f(x) > x$ and $f(y) < y$, then it follows that $x + 1 < y$ and there exists an intermediate value $z \in \mathbb{N}, x < z < y$ which is a fixed-point of f (namely $f(z) = z$).

Proof The statement $x + 1 < y$ holds because f is a monotonically increasing function defined on the natural numbers. Assume there exists no fixed-point of f between x and y, formally $\forall k \in \mathbb{N}, x < k < y : f(k) \neq k$. Then there has to exist a value $n \in \mathbb{N}, x < n < y$ such that $f(n) > n \wedge f(n+1) < (n+1)$ holds. As both the domain and the range of f is $\mathbb{N}$, it follows that $f(n) \geq (n+1)$ and $f(n+1) \leq n$. This contradicts the prerequisite that f is a monotonically increasing function. ■

Using that lemma the existence of a fixed-point for all members of the family of functions $\gamma^{z}_{p_{sub},\tau_i,t}$ is shown.

Theorem 5.11 (Fixed-Points in Task Spare-Time Functions)
Let τ_i be a schedulable task allocated to an ECU e with Spare-Time $\mathfrak{s}_{\tau_i,t}$. Let $\gamma^{z}_{p_{sub},\tau_i,t}$ with $w \in \{0, \ldots, \mathfrak{s}_{\tau_i,t}\}$ be the family of fixed-point functions as defined above. Then for function $\gamma^{w}_{p_{sub},\tau_i,t}$ there exists a fixed-point $r_i^w \in \mathbb{N}$ with $r_i^w \leq \mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)$. □

Proof The existence of a fixed-point r_i^0 for function $\gamma^{0}_{p_{sub},\tau_i,t}$ follows from the precondition that task τ_i is schedulable. The existence of a fixed-point $r_i^{\mathfrak{s}_{\tau_i,t}}$ for function $\gamma^{\mathfrak{s}_{\tau_i,t}}_{p_{sub},\tau_i,t}$ follows directly from the definition of spare-time, see Equation (5.11).

Without loss of generality, let $\gamma^{w}_{p_{sub},\tau_i,t}$ be a function of the family of spare-time functions of task τ_i with $0 < w < \mathfrak{s}_{\tau_i,t}$. Because of $w > 0$ we know that $\gamma^{w}_{p_{sub},\tau_i,t}(r_i^0) > r_i^0$. Furthermore we know that $\gamma^{w}_{p_{sub},\tau_i,t}(r_i^{\mathfrak{s}_{\tau_i,t}}) < r_i^{\mathfrak{s}_{\tau_i,t}}$ because of $w < \mathfrak{s}_{\tau_i,t}$. Then the existence of a fixed-point r_i^w for $\gamma^{w}_{p_{sub},\tau_i,t}$ follows from Lemma (5.10). ■

The notion of spare-time is used to formulate a sufficient condition expressing under which circumstances tasks that are already allocated to a given ECU remain schedulable if additional tasks are allocated to the same ECU with higher priorities. A formal definition of the sufficient condition is given in Theorem 5.12

Theorem 5.12 (Task Spare-Time Schedulability Condition)
Let τ_i be a schedulable task allocated by allocation $\mathcal{A}$ to an ECU e with ECU type t and let $\mathfrak{s}_{\tau_i,t}$ be the spare-time of that task. Let $\mathcal{T}^{\mathtt{new}}$ be a set of tasks which shall be allocated to the same ECU with higher priorities than τ_i. Task τ_i remains schedulable after those new tasks have been allocated if the following condition holds:

$$\mathfrak{s}_{\tau_i,t} \geq \sum_{\tau \in \mathcal{T}^{\mathtt{new}}} \left\lceil \frac{\mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)}{\mathrm{period}^{\mathtt{task}}(\tau)} \right\rceil \mathrm{wcet}_p(\tau, t) \tag{5.13}$$

PROOF τ_i is schedulable if its worst case response time is smaller or equal to its deadline, formally $r_i \leq \text{deadline}^{\texttt{task}}_{conf}(\tau_i)$. As stated in Equation (2.1) (page 19) the duration of all preemptions for task τ_i caused by a set of higher priority tasks is defined by

$$\sum_{\tau \in \mathcal{T}^{\texttt{new}}} \left\lceil \frac{r_i}{\text{period}^{\texttt{task}}(\tau)} \right\rceil \text{wcet}_p(\tau, t)$$

where r_i is the (known) worst case response time of τ_i. An upper bound for the duration of preemptions can be found by substituting the response time r_i with the deadline $\text{deadline}^{\texttt{task}}_{conf}(\tau_i)$ resulting in the term

$$\sum_{\tau \in \mathcal{T}^{\texttt{new}}} \left\lceil \frac{\text{deadline}^{\texttt{task}}_{conf}(\tau_i)}{\text{period}^{\texttt{task}}(\tau)} \right\rceil \text{wcet}_p(\tau, t)$$

Let y denote that upper bound (as defined above) and $x \in \{0, \ldots, y\}$ the real additional preemption duration as would be calculated by the original fixed-point equation. For every $w \in \mathbb{N}, 0 \leq w \leq y$ the family of fixed-point functions for τ_i contains a function $\gamma^w_{p_{sub},\tau_i,t}$. Due to the precondition that task τ_i is schedulable if no additional tasks are allocated to the ECU it follows from Theorem 5.11 that there exists a fixed-point r_i^w for function $\gamma^w_{p_{sub},\tau_i,t}$ with $r_i^w \leq \text{deadline}^{\texttt{task}}_{conf}(\tau_i)$. Thus task τ_i remains schedulable. ■

Note that by using an upper bound for the preemption duration caused by the additionally allocated tasks with higher priorities the calculation of the exact number of preemptions is avoided. This calculation would require to solve the original fixed-point equation during the optimization process for each task allocation onto each of the ECU for the respective currently chosen ECU-types. Using the spare-time approach the decision whether or not a set of additional tasks can be allocated without harming a given already allocated task is simplified by using the pre-calculated spare-time values.

The downside of using an upper bound for the preemption duration is a slightly loss of precision. For example, if an allocated task has a very small worst case response time far less than its deadline than using that large deadline instead of the small worst case response time for determining the number of preemptions will result in a significantly too large value. As a consequence some sets of allocated tasks would be rejected by the test proposed in Theorem (5.13) although they would fit on the given ECU as could be shown by performing a full schedulability analysis.

So far only one allocated task has been considered with one or more additional tasks to be allocated. Of course, usually there are multiple tasks allocated to each ECU. Theorem 5.13 defines a schedulability test for all the initially allocated tasks on an ECU, which simply tests the schedulability condition specified in Theorem (5.13) for all those tasks.

Theorem 5.13 (ECU Spare-Time Schedulability Condition)
Let $\mathcal{T}^{\texttt{alloc}}$ be a set of schedulable tasks already allocated to a given ECU e with ECU-type t. A set of tasks $\mathcal{T}^{\texttt{new}}$ can be additionally allocated to that ECU without rendering any of

the already allocated tasks unschedulable if the condition given in Theorem 5.12 holds for every already allocated task. □

Proof Follows directly from Theorem 5.12 and the fact that the schedulability of each task on a given ECU does only depend on the activation periods and worst case execution times of the other tasks on the same ECU with a higher priority but not on whether or not they themselves are schedulable. ■

5.2.3.2. Task MaxWCET Analysis

Spare-Time analysis only allows to decide whether or not a set of additional tasks can be allocated to an ECU without rendering the already allocated tasks unschedulable. It is not suitable for deciding whether or not the additionally allocated tasks will satisfy their deadlines if being allocated to that ECU.

The MaxWCET analysis has been developed to fill this gap. The rationale behind MaxWCET analysis is to determine generic parameters which allow to decide whether or not a set of additional tasks can be allocated to an ECU such that all of them are schedulable. Those parameters are *generic* in the sense that it is not required to already know the actual set of additional tasks for determining the parameters. Instead, it is sufficient to know some basic information about the task set, namely a set of **pseudo deadlines** which abstract from the actual task deadlines.

Definition 5.14 (Pseudo Deadline Set)
The pseudo deadline set for a given set of tasks/signals is defined as the set of all task/signal deadlines. □

The pseudo deadline set is especially useful if many of the unallocated tasks have the same deadline. In this case the huge multiset of deadlines (with multiple identical values) can be expressed with a very small set of pseudo deadlines. The MaxWCET values have to be calculated only for that small set of pseudo deadlines which saves a lot of effort.

Definition 5.15 (Task MaxWCET)
Let e be an ECU, t an ECU-type and $\hat{d}$ a pseudo deadline. The corresponding MaxWCET value is denoted by $\mathfrak{m}_{\hat{d},e,t}$ and defined as follows:

$$\begin{aligned}
&\mathfrak{m}_{\hat{d},e,t} \in \mathbb{N} \text{ is the greatest natural number such that} \\
&\quad \exists r \in \mathbb{N} : \delta_{p_{sub},\hat{d},e,t}(r) = r \wedge r \leq \hat{d} \text{ with} \\
&\delta_{p_{sub},\hat{d},e,t}(x) := \mathfrak{m}_{\hat{d},e,t} + \sum_{\substack{\tau \in \mathcal{T}_e \wedge \\ \text{deadline}^{\texttt{task}}_{conf}(\tau) \leq \hat{d}}} \left\lceil \frac{x}{\text{period}^{\texttt{task}}(\tau)} \right\rceil \text{wcet}_p(\tau, t)
\end{aligned} \tag{5.14}$$

where $\mathcal{T}_e$ is the set of tasks already allocated by $\mathcal{A}$ to ECU e. □

The right side of the fixed-point equation calculates the sum of all preemption durations caused by all tasks already allocated to an ECU e that have a deadline smaller or equal to the chosen pseudo deadline. All tasks with a larger deadline would have a lower priority in the final allocation to the ECU and therefore would not preempt any of the additional allocated task with a deadline equal or less than the pseudo deadline.

One “corner case” is already handled properly in the definition of MaxWCET but nevertheless should be mentioned explicitly. For each additional task which would be allocated with a priority higher than all of the already allocated tasks on an ECU (due to its smaller deadline), the corresponding MaxWCET value is equal to its pseudo deadline. This is because such tasks would solely be preempted by other additionally allocated tasks with an even higher priority but never by any of the already allocated tasks.

Again, a family of fixed-point equations can be defined.

Definition 5.16 (Family of Task MaxWCET Fixed-Point Equations)
For the local analysis problem p_{sub} specified above with the given allocation $\mathcal{A}$, an ECU e part of the subsystem hardware architecture, an ECU-type t and a pseudo deadline $\hat{d}$, with MaxWCET $\mathfrak{m}_{\hat{d},e,t}$ let

$$\delta^{z}_{p_{sub},\hat{d},e,t}(x) := z + \sum_{\substack{\tau \in \mathcal{T}_e \wedge \\ \mathrm{deadline}^{\mathtt{task}}_{conf}(\tau) \leq \hat{d}}} \left\lceil \frac{x}{\mathrm{period}^{\mathtt{task}}(\tau)} \right\rceil \mathrm{wcet}_p(\tau, t)$$

be the family of MaxWCET fixed-point equations with $z \in \mathbb{N}, 0 \leq z \leq \mathfrak{m}_{\hat{d},e,t}$. □

A fixed-point exists for each of the functions that belong to a given MaxWCET Function Family. The proof for this statement is very similar to the one for the existence of fixed-points in all functions of a given spare-time function family.

Theorem 5.17 (Fixed-Points in Task MaxWCET Functions)
For a given allocation $\mathcal{A}$, a pseudo deadline $\hat{d}$, an ECU e and an ECU-type t let $\mathfrak{m}_{\hat{d},e,t}$ denote the MaxWCET value defined as above.

Let $\delta^{z}_{p_{sub},\hat{d},e,t}$ with $z \in \{0, \ldots, \mathfrak{m}_{\hat{d},e,t}\}$ be the corresponding family of fixed-point functions. Then for every member of that family $\delta^{w}_{p_{sub},\hat{d},e,t}$ with $w \in \{0, \ldots, \mathfrak{m}_{\hat{d},e,t}\}$ there exists a fixed-point $r^w \in \mathbb{N}$ with $r^w \leq \hat{d}$. □

Proof The existence of a fixed-point $r^0 = 0$ for function $\delta^{0}_{p_{sub},\hat{d},e,t}$ directly follows from the definition of that function. The existence of a fixed-point $r^{\mathfrak{m}_{\hat{d},e,t}}$ for function $\delta^{\mathfrak{m}_{\hat{d},e,t}}_{p_{sub},\hat{d},e,t}$ follows directly from the definition of MaxWCET (see Definition 5.15).

Without loss of generality, let $\delta^{w}_{p_{sub},\hat{d},e,t}$ be a function of the family of MaxWCET functions with $0 < w < \mathfrak{m}_{\hat{d},e,t}$. Because of $w > 0$ we know that $\delta^{w}_{p_{sub},\hat{d},e,t}(r^0) > r^0$. Furthermore we know that $\delta^{w}_{p_{sub},\hat{d},e,t}(r^{\mathfrak{m}_{\hat{d},e,t}}) < r^{\mathfrak{m}_{\hat{d},e,t}}$ because of $w < \mathfrak{m}_{\hat{d},e,t}$. Then the existence of a fixed-point r^w for $\delta^{w}_{p_{sub},\hat{d},e,t}$ follows from Lemma (5.10). ■

Theorem 5.18 (Task MaxWCET Schedulability Condition)
Let $\mathcal{T}_e$ be the set of allocated tasks on ECU e and $\mathcal{T}^{\mathtt{new}}_e \subseteq \mathcal{T}^{\mathtt{pre}}$ be the subset of the pre-allocated tasks which shall be allocated to that ECU. Let $\tau_i \in \mathcal{T}^{\mathtt{new}}_e$ be an unallocated

task with deadline $\mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)$*. Let* $\hat{\mathbb{D}}$ *be the set of pseudo deadlines derived from the set of pre-allocated tasks and* $\hat{d} \in \hat{\mathbb{D}}, \hat{d} = \mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)$ *be the relevant pseudo deadline for* τ_i*. Let* $\mathfrak{m}_{\hat{d},e,t}$ *be the corresponding MaxWCET value for ECU-type t as specified in Definition 5.15.*

If all tasks in $\mathcal{T}^{\mathtt{new}}_e$ *would be allocated to ECU e with ECU-type t then task* τ_i *is guaranteed to be schedulable if the following condition holds:*

$$\mathfrak{m}_{\hat{d},e,t} \geq \mathrm{wcet}_p(\tau_i, t) + \sum_{\substack{\tau \in \mathcal{T}^{\mathtt{new}}_e \land \\ \mathrm{prio}^{>}(\tau,\tau_i)=\mathtt{true}}} \left\lceil \frac{\hat{d}}{\mathrm{period}^{\mathtt{task}}(\tau)} \right\rceil \mathrm{wcet}_p(\tau, t)$$

where the additionally allocated task τ_i *is only preempted by those additionally allocated tasks which would have a higher priority on ECU ECU (all tasks* $\tau \in \mathcal{T}^{\mathtt{new}}_e$ *where* $\mathrm{prio}^{>}(\tau, \tau_i) = \mathtt{true}$*).* □

Proof The proof is analog to that of Theorem 5.12. τ_i is schedulable if its worst case response time is smaller or equal to its deadline, formally $r_i \leq \mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)$. As stated in Equation (2.1) (see Page 19) the duration of all preemptions for task τ_i caused by the set of higher priority tasks can be calculated by

$$\sum_{\substack{\tau \in \mathcal{T}^{\mathtt{new}}_e \land \\ \mathrm{prio}^{>}(\tau,\tau_i)=\mathtt{true}}} \left\lceil \frac{r}{\mathrm{period}^{\mathtt{task}}(\tau)} \right\rceil \mathrm{wcet}_p(\tau, t)$$

where r is the (known) worst case response time of τ_i. An upper bound for the duration of preemptions can be found by substituting the response time r_i with the deadline $\mathrm{deadline}^{\mathtt{task}}_{conf}(\tau_i)$ resulting in the term

$$\sum_{\substack{\tau \in \mathcal{T}^{\mathtt{new}}_e \land \\ \mathrm{prio}^{>}(\tau,\tau_i)=\mathtt{true}}} \left\lceil \frac{\hat{d}}{\mathrm{period}^{\mathtt{task}}(\tau)} \right\rceil \mathrm{wcet}_p(\tau, t)$$

Let y denote that upper bound (as defined above) and $w \in \{0, \ldots, y\}$ the real preemption duration as would be calculated by the original fixed-point equation caused by all other additional tasks which would have a higher priority on the same ECU. For every possible value $0 \leq w \leq y$ the family of fixed-point functions for τ_i contains a function $\delta^w_{p_{sub},\tau_i,t}$. It follows from Theorem 5.17 that there exists a fixed-point r^w_i for function $\delta^w_{p_{sub},\tau_i,t}$ with $r^w_i \leq \hat{d}$. Thus task τ_i would be schedulable. ■

Theorem 5.19 (ECU MaxWCET Schedulability Condition)
Let $\mathcal{T}^{\mathtt{alloc}}$ *be a set of schedulable tasks already allocated to a given ECU e with ECU-type t. A set of tasks* $\mathcal{T}^{\mathtt{new}}$ *can be additionally allocated to that ECU such that all* $\tau \in \mathcal{T}^{\mathtt{new}}$ *are schedulable if the MaxWCET Schedulability condition specified in Theorem 5.18 holds for every of those tasks.* □

Proof Follows directly from Theorem 5.18 and the fact that the schedulability of each task on a given ECU does only depend on the activation periods and worst case execution times of the other tasks on the same ECU with a higher priority but not on whether or not they themselves are schedulable. ■

5.2.4. Combining Spare-Time and MaxWCET Analysis

When combining the notions of spare-time and MaxWCET a sufficient condition is available which — if evaluated to true — guarantees that a given ECU will remain schedulable if a given set of additional tasks is allocated to that ECU, see Theorem 5.20.

Theorem 5.20 (spare-time and MaxWCET Schedulability Condition)
Let $\mathcal{T}^{\texttt{alloc}}$ be a set of schedulable tasks already allocated to a given ECU e with ECU-type t. A set of tasks $\mathcal{T}^{\texttt{new}}$ can be additionally allocated to that ECU such that all tasks (already allocated and additionally allocated) and therefore the whole ECU are schedulable, if both the conditions specified in Theorem 5.13 (schedulability of all already allocated tasks as guaranteed by the spare-time condition) and Theorem 5.19 (schedulability of all additionally allocated tasks as guaranteed by the MaxWCET condition). □

Proof Follows directly from the proofs of Theorems 5.13 and 5.19. ■

An important property of both the spare-time and MaxWCET schedulability conditions are that they can easily be expressed via linear equations. Details how to exploit this property can be found in Section 5.4.

5.2.5. Analysis of Communication Media: Controller Area Network

This thesis focuses on priority-based communication for local communication buses. The proposed Spare-Time/MaxWCET approach can be applied with little extensions to priority-based communication buses. This work uses Controller Area Network (CAN) buses.

The theory behind the schedulability analysis of ECUs with fixed-priority preemptive scheduling and of communication buses with the priority-based CAN protocol is strongly related. The well-known fixed-point equations (see Equation (2.1) on page 19) can be applied to both of them, when considering one important difference: Once the transmission of a message on a CAN bus has been started it cannot be preempted. Only during the arbitration phase it is decided which of the active messages is sent on the bus. This is always the active message with the highest priority. But if such a message arrives (is activated) while a lower-priority message is being transmitted, the transmission of the higher-priority message cannot start immediately but is delayed until the next arbitration phase is reached. The lower-priority message blocks the higher-priority message. The blocking time for a message is defined as the maximal time any lower-priority message can delay its transmission. In Section 2.7 (on page 21) it is shown how blocking times have been incorporated in the classical schedulability analysis.

As the transmission of a message cannot be interrupted, the transmission time is not included in the fixed-point part of the worst case response time equation (see Section 2.7 (on page 21).

Let p_{sub} be a local analysis problem for a subsystem *sub* with allocation $\mathcal{A}$ and local bus b with bus type $\mathfrak{b}$. A message m_i associated to a signal s_i which is allocated to the local bus b is schedulable if its worst case response time ri is smaller or equal to its deadline $\text{deadline}^{\texttt{msg}}_{conf}(m_i)$:

$$\begin{aligned} r_i &= \text{wctt}(s_i, \mathfrak{b}) + \omega_i, \text{ with} && (5.15)\\ \omega_i &= block_i + \sum_{\substack{s_j \in S_b:\\ \text{prio}^{>}(s_j,s_i)}} \left\lceil \frac{\omega_i + 1}{\text{period}^{\texttt{msg}}(s_j)} \right\rceil \text{wctt}(s_j, \mathfrak{b}), \text{ and} && (5.16)\\ block_i &:= \max_{\substack{s_j \in S_b:\\ s_i = s_j \,\vee\\ \text{prio}^{>}(s_i,s_j)}} \{\text{wctt}(s_j, \mathfrak{b})\} \end{aligned}$$

where S_b denotes the set of signals allocated to the local bus b via messages and $\text{wctt}(s_j, \mathfrak{b})$ the worst case transmission time of a signal s_i if sent on a bus with bus type $\mathfrak{b}$ via message m_i. Signal s_i is schedulable on bus b if:

$$r_i \leq \text{deadline}^{\texttt{msg}}_{conf}(m_i) \qquad (5.17)$$

where m_i is the message associated to signal s_i.

In Equations (5.15) the worst case transmission time (WCRT) is given by r_i, the worst case response time by r_i and the busy period (see e.g. [Dav+07] for definition) by ω_i. The busy period for any signal s on the local bus b is defined as fixed-point equation (see Equation (5.16)) which due to the uninterruptible nature of CAN signal transmission includes interruptions caused by signal s itself. This is realized by summing the interruptions caused by signals with higher or equal priority. To avoid the problem that for a signal with blocking time equal to zero an empty busy period (with a duration of 0) would be a valid solution for the fixed-point equation, the smallest possible transmission time (which is 1) is added to the busy period variable during the calculation of the number of preemptions. $block_i$ is the maximal blocking time for signal s_i caused by lower-priority signals or by a prior invocation of signal s_i. Finally Equation (5.17) specifies the condition under which signal s_i is schedulable.

5.2.5.1. Signal Spare-Time Analysis

Equations (5.15) and (5.16) can be modified to include the necessary Spare-Time variable. This is accomplished by replace the fixed-point equation used to calculate the busy period by a new function Θ.

Definition 5.21 (Signal spare-time)
Let p be a DSE Problem and p_{sub} a local analysis problem as defined above. Let b be a bus contained in the hardware architectural pattern of p with bus type $\mathfrak{b}$. Let S_b be the set of signals allocated via messages to the local bus b and $S^{\mathtt{new}}$ the set of signals which are currently not allocated to that bus but might get allocated during the local analysis step. The spare-time $\mathfrak{s}_{s_i,b}$ of a signal s_i allocated to that bus using a message m_i is defined as:

$$\begin{aligned} &\mathfrak{s}_{s_i,b} \in \mathbb{N} \textit{ is the greatest natural number such that} \\ &\exists r_i \in \mathbb{N} : \theta_{p_{sub},s_i,b}(r_i) = r_i \land \\ &\qquad (r_i + \mathrm{wctt}(s_i, \mathfrak{b})) \leq \mathrm{deadline}^{\mathtt{msg}}_{conf}(m_i) \end{aligned} \tag{5.18}$$

where

$$\begin{aligned} \theta_{p_{sub},s_i,b}(x) &:= block^{\mathtt{ub}}_{p_{sub}}(s_i) + \mathfrak{s}_{s_i,b} + \\ &\left\lceil \frac{x+1}{\mathrm{period}^{\mathtt{sig}}(s_j)} \right\rceil \mathrm{wctt}(s_j, \mathfrak{b}) \end{aligned} \tag{5.19}$$

$$\begin{aligned} block^{\mathtt{ub}}_{p_{sub}}(s_i) &:= \max\{\mathrm{wctt}(s_j, \mathfrak{b}) \mid s_j \in S_b \cup S^{\mathtt{new}} : \\ &\qquad s_i = s_j \lor \mathrm{prio}^{>}(s_i, s_j)\} \end{aligned} \qquad \square$$

For simplicity, in the above equations most of the notations are defined with references to signals not to the corresponding messages, e.g. the spare-time is defined for a signal on a given bus (which is unambiguous because each signal corresponds to maximal one message per bus). There is one exception: The deadlines for signals sent over a subsystem-local bus are defined via their corresponding messages.

While calculating spare-time values, it is not known yet which of the unallocated signals will be allocated to the local bus during the following optimization step. Therefore the definition of spare-time for signals uses an upper bound for the blocking time which considers for each signal not only the worst case transmission times for all the already allocated signals with lower priority but additionally all unallocated signals. This may lead to a blocking time which is too large. But using too large blocking times is a safe over-approximation which does not invalidate the spare-time results.

As for the spare-time, a family of signal spare-time fixed-point equations is defined:

Definition 5.22 (Family of Signal Spare-Time Fixed-Point Equations)
Let p be a DSE Problem and p_{sub} a local analysis problem for a subsystem sub. Let s_i be a signal with corresponding schedulable message m_i which is allocated to the local bus b of the subsystem. By replacing the variable $\mathfrak{s}_{s_i,b}$ in Equation (5.19) by a parameter $z \in \{0, \ldots, \mathfrak{s}_{s_i,b}\}$ a family of functions $\theta^z_{p_{sub},s_i,b}(x)$ is defined by:

$$\begin{aligned} \theta^z_{p_{sub},s_i,b}(x) &:= block^{\mathtt{ub}}_{p_{sub}}(s_i) + z + \\ &\sum_{\substack{s_j \in S_b: \\ \mathrm{prio}^{>}(s_j,s_i)}} \left\lceil \frac{x+1}{\mathrm{period}^{\mathtt{sig}}(s_j)} \right\rceil \mathrm{wctt}(s_j, \mathfrak{b}) \end{aligned} \qquad \square$$

In Theorem 5.23, the existence of a fixed-point for each function included in that family is proven.

Theorem 5.23 (Fixed-Points in Signal Spare-Time Functions)
Let s_i be a schedulable signal allocated to a local bus via message m_i, and $\mathfrak{s}_{s_i,b}$ be the corresponding Spare-Time. Let $\theta^z_{p_{sub},s_i,b}(x)$ with $z \in \{0,\dots,\mathfrak{s}_{s_i,b}\}$ be the family of signal fixed-point functions as defined above. Then for every function of that family there exists a fixed-point $r_i^z \in \mathbb{N}$ with

$$r_i^z \leq \text{deadline}^{\texttt{msg}}_{conf}(m_i) - \text{wctt}(s_i, \mathfrak{b})$$

□

Proof The existence of a fixed-point r_i^0 for function $\theta^0_{p_{sub},s_i,b}$ follows from the precondition that the message m_i is schedulable. The existence of a fixed-point $r_i^{\mathfrak{s}_{s_i,b}}$ for function $\theta^{r_i^{\mathfrak{s}_{s_i,b}}}_{p_{sub},s_i,b}$ follows directly from the definition of signal spare-time (see Equation (5.19)). Without loss of generality, let $\theta^w_{p_{sub},s_i,b}$ be a function of the family of spare-time functions of signal s_i with $0 < w < \mathfrak{s}_{s_i,b}$. Because of $w > 0$ we know that $\theta^w_{p_{sub},s_i,b}(r_i^0) > r_i^0$. Furthermore we know that $\theta^w_{p_{sub},s_i,b}(r_i^{\mathfrak{s}_{s_i,b}}) < r_i^{\mathfrak{s}_{s_i,b}}$ because of $w < \mathfrak{s}_{s_i,b}$. Then the existence of a fixed-point r_i^w for $\theta^w_{p_{sub},s_i,b}$ follows from Lemma (5.10). ■

The busy periods calculated for signals do not include the signals own worst case transmission times. As a consequence that transmission time has to be subtracted from the deadline while calculating the fixed-point (see Definition 5.21). This is the most important difference in fixed-point Spare-Time function family for signals defined in Theorem 5.23 compared to its equivalent for tasks.

Theorem 5.24 (Signal Spare-Time Schedulability Condition)
Let p be a DSE Problem and p_{sub} a local analysis problem as defined above. Let b be a bus contained in the hardware architectural pattern of p with type $\mathfrak{b}$. Let S_b be the set of signals already allocated via messages to the local bus b and $S^{\texttt{new}}$ be the set of unallocated signals. $\mathfrak{s}_{s_i,b}$ be the spare-time of a schedulable signal $s_i \in S_b$ already allocated to that bus.

A signal s_i allocated to the local bus via message m_i remains schedulable even if all signals in $S^{\texttt{new}}$ are allocated to the same bus, as long as the following condition holds:

$$\mathfrak{s}_{s_i,b} \geq \sum_{\substack{s_j \in S^{\texttt{new}} \\ \text{prio}^{>}(s_j,s_i)}} \left\lceil \frac{\text{deadline}^{\texttt{msg}}_{conf}(m_i)}{\text{period}^{\texttt{sig}}(s_j)} \right\rceil \text{wctt}(s_j, \mathfrak{b})$$

where m_i is the message on the local bus b that corresponds to signal s_i. □

Proof This proof is similar to the proof for Theorem 5.12 (page 96) for tasks. Due to the precondition that the messages m_i is schedulable, a family of signal Spare-Time functions exists as defined above. As long as the additional preemptions (during the CAN arbitration phase) by additional signals with higher priority is smaller or equal to the calculated Spare-Time, there exists a corresponding function in the family of signal Spare-Time functions which has a fixed-point according to Theorem 5.23. ■

5.2.5.2. Signal MaxWCET Analysis

Similar to its equivalent for tasks, the MaxWCET analysis for signals uses a set of pseudo deadlines. As the unallocated signals are not represented by a message on the local bus yet, their deadlines are approximated to be equal to their activation periods when determining the set of pseudo deadlines. Unlike the MaxWCET values for tasks the MaxWCET values for signals do not depend on any bus type but only on the local bus itself (in the scope of a local analysis problem whose subsystem contains that bus). Referring only to the bus is sufficient because one limitation of the DSE approach presented in this work is that the type of buses remains unmodified during the whole optimization phase.

In contrary to the signal spare-time where the original mathematically formulation for schedulability as in Equation (5.17) can be used, a simple over-approximation is required for the signal MaxWCET analysis. As explained before, in the original equations the worst case transmission time of the signal under evaluation is not part of the fixed-point equation itself. This is because a transmission, once started, cannot be preempted. Therefore the fixed-point equation only subsumes the durations of the preemptions occurring during the arbitration phase before the actual transmission is started.

The over-approximation required to adapt the MaxWCET analysis as formulated for tasks in order to make it applicable to signals on CAN buses is to assume that the transmission could be preempted. This implies that the worst case transmission time of the signal is added to the fixed-point equation as shown in Definition 5.25. This modification of the original mathematical formulation is necessary because it avoids to distinguish between the worst case transmission time of the signal under evaluation and the preemption caused for that signal by other additionally allocated signals. Those preemptions could lead to an increase of the busy period, while the worst case transmission time of the signal under evaluation could not, if using the original mathematical formulation.

Definition 5.25 (Signal MaxWCET)
Let p be a DSE Problem and p_{sub} a local analysis problem as defined above. Let b be a bus with type $\mathfrak{b}$, S_b is the set of signals already allocated to bus b, and $\hat{d}$ a pseudo deadline. The corresponding MaxWCET value is denoted by $\mathfrak{m}_{\hat{d},b}$ and defined as follows:

$$\mathfrak{m}_{\hat{d},b} \in \mathbb{N} \textit{ is the greatest natural number such that}$$
$$\exists r \in \mathbb{N} : \kappa_{p_{sub},\hat{d},b}(r) = r \wedge r \leq \hat{d} \textit{ with}$$

$$\kappa_{p_{sub},\hat{d},b}(x) := \mathit{block}^{\mathtt{ub}}_{p_{sub}}(\hat{d}) + \mathfrak{m}_{\hat{d},b} + \sum_{\substack{s \in S_b \,\wedge \\ \mathrm{deadline}^{\mathtt{msg}}_{conf}(m) \leq \hat{d}}} \left\lceil \frac{x+1}{\mathrm{period}^{\mathtt{sig}}(s)} \right\rceil \mathrm{wctt}(s, \mathfrak{b}) \tag{5.20}$$

$$\mathit{block}^{\mathtt{ub}}_{p_{sub}}(\hat{d}) := \max\{\mathrm{wctt}(s, \mathfrak{b}) \mid s \in S_b \cup S^{\mathtt{new}} \wedge \hat{d} \leq d_s\} \textit{ and} \tag{5.21}$$

$$d_s := \begin{cases} \mathrm{deadline}^{\mathtt{msg}}_{conf}(m) & s \in S_b \\ \mathrm{period}^{\mathtt{sig}}(s) & \textit{otherwise} \end{cases} \tag{5.22}$$

where m denotes the corresponding message for s. □

Note that for signals which are not yet allocated to the local bus via a message no local deadline for that bus has been synthesized yet. Therefore, the above definition uses the signal's period for deciding whether or not an unallocated signal has to be considered for calculating the blocking time of another signal.

With the over-approximated model, an increase in either the worst case transmission time of the signal under evaluation or the preemption duration could potentially have the same effect: an increasing busy period. The modification is a safe over-approximation: if a fixed-point smaller than or equal to the deadline can be found with the modified fixed-point equation then there exists a fixed-point for the original fixed-point equation as well which is smaller than the one for the modified equation minus the MaxWCET value. This fixed-point for the original equation satisfy the condition stated in Equation (5.17).

The family of MaxWCET fixed-point functions for signals is defined as follows:

Definition 5.26 (Family of Signal MaxWCET Fixed-Point Equations)
Let p be a DSE Problem and p_{sub} a local analysis problem as defined above. Let S_b denote the set of signals already allocated to the local bus b, $\hat{d}$ a pseudo deadline, and $S^{\mathtt{new}}$ a set of unallocated signals.

By replacing the variable $\mathfrak{m}_{\hat{d},b}$ in Equation (5.20) by a parameter $z \in \{0, \dots, \mathfrak{m}_{\hat{d},b}\}$ a family of functions $\kappa^{z}_{p_{sub},\hat{d},b}(x)$ is defined with:

$$\kappa^{z}_{p_{sub},\hat{d},b}(x) := block^{\mathtt{ub}}_{p_{sub}}(\hat{d}) + z + \sum_{\substack{s \in S_b \wedge \\ \mathrm{deadline}^{\mathtt{msg}}_{conf}(m) \leq \hat{d}}} \left\lceil \frac{x+1}{\mathrm{period}^{\mathtt{sig}}(s)} \right\rceil \mathrm{wctt}(s, \mathfrak{b})$$

where $block^{\mathtt{ub}}_{p_{sub}}(\hat{d})$ is defined as in Definition 5.25 and m denotes the message corresponding to signal s. □

Again, it can be shown that a fixed-point exists for every function in the family of signal MaxWCET functions.

Theorem 5.27 (Fixed-Points in Signal MaxWCET Functions)
For a pseudo deadline $\hat{d}$, and a local bus with type $\mathfrak{b}$ let $\mathfrak{m}_{\hat{d},b}$ denote the MaxWCET value defined as above.

Let $\kappa^{z}_{p_{sub},\hat{d},b}$ with $z \in \{0, \dots, \mathfrak{m}_{\hat{d},b}\}$ be the corresponding family of fixed-point functions. Then for every member of that family there exists a fixed-point $r^z \in \mathbb{N}$ with $r^z \leq \hat{d}$. □

Proof The existence of a fixed-point r^0 for function $\kappa^{0}_{p_{sub},\hat{d},b}$ follows from the precondition that the bus is schedulable if left unchanged. The existence of a fixed-point $r^{\mathfrak{m}_{\hat{d},b}}$ for function $\kappa^{r^{\mathfrak{m}_{\hat{d},b}}}_{p_{sub},\hat{d},b}$ follows directly from the definition of the signal MaxWCET (see Definition 5.25). Without loss of generality, let $\kappa^{w}_{p_{sub},\hat{d},b}$ be a function of the family of MaxWCET functions with $0 < w < \mathfrak{m}_{\hat{d},b}$. Because of $w > 0$ we know that $\kappa^{w}_{p_{sub},\hat{d},b}(r^0) >$

r^0. Furthermore we know that $\kappa^w_{p_{sub},\hat{d},b}(r^{\mathfrak{m}_{\hat{d},b}}) < r^{\mathfrak{m}_{\hat{d},b}}$ because of $w < \mathfrak{m}_{\hat{d},b}$. Then the existence of a fixed-point r^w for $\kappa^w_{p_{sub},\hat{d},b}$ follows from Lemma (5.10). ∎

Theorem 5.28 (Signal MaxWCET Schedulability Condition)
For a given local analysis problem p_{sub} with local bus b with bus type $\mathfrak{b}$ let $\hat{d}$ be a pseudo deadline and $\mathfrak{m}_{\hat{d},b}$ the corresponding MaxWCET as defined above. Let S_b denote the set of signals which are already allocated to the local bus via messages.

A set of unallocated signals $S^{\mathtt{new}}$ can be additionally allocated to that bus such that an unallocated signal $s_i \in S^{\mathtt{new}}$ which deadline is equal to the pseudo deadline is schedulable, if the following condition holds:

$$\mathfrak{m}_{\hat{d},b} \geq \mathrm{wctt}_p(s_i, \mathfrak{b}) + \sum_{\substack{s \in S^{\mathtt{new}} \wedge \\ \mathrm{prio}^{>}(s,s_i)}} \left\lceil \frac{\hat{d}}{\mathrm{period}^{\mathtt{sig}}(s)} \right\rceil \mathrm{wctt}_p(s, \mathfrak{b}) \qquad \square$$

Proof The proof is analog to that of Theorem 5.18. s_i is schedulable if the corresponding message m_i on the local bus is schedulable. That message is schedulable if its worst case response time is smaller or equal to its deadline, formally $r_i \leq \mathrm{deadline}^{\mathtt{msg}}_{conf}(m_i)$. As stated in Equation (2.1) (see Page 19) the duration of all preemptions for message m_i caused by a set of higher priority messages can be calculated as:

$$\sum_{\substack{s \in S^{\mathtt{new}} \wedge \\ \mathrm{prio}^{>}(s,s_i)}} \left\lceil \frac{r_i}{\mathrm{period}^{\mathtt{sig}}(s)} \right\rceil \mathrm{wctt}_p(s, \mathfrak{b})$$

where r_i is the (known) worst case response time of m_i. An upper bound for the duration of preemptions can be found by substituting the response time r_i with the pseudo deadline resulting in the term

$$\sum_{\substack{s \in S^{\mathtt{new}} \wedge \\ \mathrm{prio}^{>}(s,s_i)}} \left\lceil \frac{\hat{d}}{\mathrm{period}^{\mathtt{sig}}(s)} \right\rceil \mathrm{wctt}_p(s, \mathfrak{b})$$

Let y denote that upper bound (as defined above) and $w \in \{0, \ldots, y\}$ the real preemption duration as would be calculated by the original fixed-point equation caused by all other additional messages which would have a higher priority on the same local bus. For every possible value $0 \leq w \leq y$ the family of fixed-point functions for s_i contains a function $\kappa^w_{p_{sub},\hat{d},b}$. It follows from Theorem 5.27 that there exists a fixed-point r^w_i for that function with $r^w_i + \mathrm{wctt}_p(s_i, \mathfrak{b}) \leq \hat{d}$. Thus signal s_i would be schedulable. ∎

Theorem 5.29 (Signal spare-time/MaxWCET Schedulability)
Let p_{sub} be a given local analysis problem for a subsystem sub with local bus b and bus type $\mathfrak{b}$. Let S_b denote the set of signals which are already allocated to the local bus via messages.

A set of unallocated signals $S^{\texttt{new}}$ can be additionally allocated to that bus such that all allocated signals remain schedulable and all additionally allocated signals are schedulable as well, if the spare-time condition specified in Theorem 5.24 holds for all allocated signals in S_b and the MaxWCET condition specified in Theorem 5.28 holds for all unallocated signals in $S^{\texttt{new}}$. □

Proof Follows directly from the referenced theorems and the fact that the schedulability is evaluated independently for each signal. ■

5.3. Calculating Spare-Time and MaxWCET

The calculation of both spare-time and MaxWCET values for each task has to be done separately for each ECU in the hardware subsystem for each applicable ECU-type. In the following subsections t denotes an ECU-type without loss of generality.

The spare-time $\mathfrak{s}_{\tau_i,t}$ for a task τ_i allocated to an ECU e can be determined by iteratively calculating the fixed-point equation for increasing candidate spare-time values starting with 0 until a value is reached for which τ_i is not schedulable anymore. Then the spare-time is the last value for which that task still has been schedulable.

5.3.1. Task Spare-Time: Brute-Force Approach

Algorithm 1 implements a brute-force approach in the sense that in each iteration the candidate spare-time value is increased only by 1.

The algorithm begins by initializing the spare-time variables with 0 (Line 2) and running the classical schedulability test in the inner loop (Lines 4–13). In case a worst case response time r_i smaller or equal to the task's deadline d_i is found, the spare-time variable is increased by 1 and the response time analysis is restarted. If no valid response time is found, the last spare-time for which the schedulability analysis succeeded is returned (see Line 11).

5.3.2. Sophisticated Approach

Compared to the previous algorithm this more sophisticated approach is able to skip many intermediate spare-time candidates which have only a minor effect on the fixed-point equation without rendering the task unschedulable.

The first part of Algorithm 2 is identical to its brute-force counter-part. In the second part, instead of increasing the spare-time variable used in the next iteration by 1, it is initialized with the largest remaining amount of time (Line 13). Obviously the Spare-Time can never be larger than the difference of the deadline and the response time calculated so far. Then the algorithm iterates over all higher priority tasks and calculates the largest Spare-Time value at which each of the ceiling functions will remain stable (no increase). The minimum of those values is the spare-time where the whole fixed-point equation is still in balance. The calculated spare-time is valid, but not necessarily the largest possible spare-time. If the spare-time value equals to the best known value the algorithm

Algorithm 1 spare-time: Brute-force Algorithm

```
1: function CalcSpareTimeOfTask($\mathcal{A}$, $\tau_i$)
2:     $\mathfrak{s}_{\tau_i,t} \leftarrow 0; \mathfrak{s}^{'}_{\tau_i,t} \leftarrow 0$
3:     $r_i \leftarrow 0$
4:     while $r_i \leq d_i$ do                ▷ Outer loop: Handle spare-time variable $\mathfrak{s}_{\tau_i,t}$
5:         $r_i \leftarrow 0; r'_i \leftarrow (\text{wcet}_p(\tau_i, t) + \mathfrak{s}^{'}_{\tau_i,t})$
6:         while $r_i < r'_i$ do              ▷ Inner loop: Calculate fixed-point equation
7:             $r_i \leftarrow r'_i; r'_i \leftarrow (\text{wcet}_p(\tau_i, t) + \mathfrak{s}^{'}_{\tau_i,t})$
8:             for $\tau_j \in \text{prio}^{>}(\tau_i, e)$ do
9:                 $r'_i \leftarrow \left(r'_i + \left\lceil \frac{r_i}{\text{period}^{\text{task}}(\tau_j)} \right\rceil \text{wcet}_p(\tau_j, t)\right)$
10:            end for
11:            if $r'_i > d_i$ then return $\mathfrak{s}_{\tau_i,t}$      ▷ Return last working spare-time
12:            end if
13:        end while
14:        $\mathfrak{s}_{\tau_i,t} \leftarrow \mathfrak{s}^{'}_{\tau_i,t}; \mathfrak{s}^{'}_{\tau_i,t} \leftarrow (\mathfrak{s}^{'}_{\tau_i,t} + 1)$      ▷ Increase spare-time variable
15:    end while
16: end function
```

terminates by returning that value. Otherwise the algorithm saves that value and restarts the iteration with the found spare-time value increased by 1 (thus destabilizing the ceiling function).

5.3.3. Task MaxWCET

The algorithm for calculating task MaxWCET values for pseudo deadlines is very similar to the sophisticated algorithm for the calculation of spare-time values presented above. Please refer to the implementation for details.

5.3.4. Signal Spare-Time and MaxWCET

The algorithms for calculating signal spare-time and MaxWCET values for pseudo deadlines are also very similar to the sophisticated algorithm presented above. Please refer to the implementation for details.

5.4. Spare-Time and MaxWCET: Encoding and Optimization

In this section all formulas are referring to exactly one subsystem *sub*. Therefore I skip the subsystem qualification of some sets such as the set of tasks contained in a subsystem or pre-allocated to that subsystem $\mathcal{T}_{sub}$ and write $\mathcal{T}$ instead. The following list defines the meaning of the additionally symbols used in the next sections. All other symbols retain their specific meaning. *tn* denotes the task network of the DSE problem *p*.

Algorithm 2 Task spare-time: Sophisticated Algorithm

```
function CalcSpareTimeOfTask($\mathcal{A}$, $\tau_i$)
    $\mathfrak{s}_{\tau_i,t} \leftarrow 0; \mathfrak{s}^{'}_{\tau_i,t} \leftarrow 0$
    $r_i \leftarrow 0$
    while $r_i \leq d_i$ do
        $r_i \leftarrow 0; r'_i \leftarrow (\mathrm{wcet}_p(\tau_i, t) + \mathfrak{s}^{'}_{\tau_i,t})$
        while $r_i < r'_i$ do
            $r_i \leftarrow r'_i; r'_i \leftarrow (\mathrm{wcet}_p(\tau_i, t) + \mathfrak{s}^{'}_{\tau_i,t})$
            for $\tau_j \in \mathrm{prio}^{>}(\tau_i, e)$ do
                $r'_i \leftarrow \left(r'_i + \left\lceil \frac{r_i}{\mathrm{period}^{\mathtt{task}}(\tau_j)} \right\rceil \mathrm{wcet}_p(\tau_j, t)\right)$
            end for
            if $r'_i > d_i$ then return $\mathfrak{s}_{\tau_i,t}$
            end if
        end while
        $\mathfrak{s}_{\tau_i,t} = \mathfrak{s}^{'}_{\tau_i,t}$                                  ▷ Save spare-time value
        $\mathfrak{s}^{'}_{\tau_i,t} \leftarrow (d_i - r'_i)$                    ▷ Initialize with largest possible value
        for $\tau_j \in \mathrm{prio}^{>}(\tau_i, t)$ do
            $\mathfrak{s}^{'}_{\tau_i,t} \leftarrow \min\left(\mathfrak{s}^{'}_{\tau_i,t}, \left(\left\lceil \frac{r'_i}{\mathrm{period}^{\mathtt{task}}(\tau_j)} \right\rceil \mathrm{period}^{\mathtt{task}}(\tau_j) - 1 - \mathrm{wcet}_p(\tau_i, t)\right)\right)$
        end for
        if $\mathfrak{s}_{\tau_i,t} = \mathfrak{s}^{'}_{\tau_i,t}$ then return $\mathfrak{s}_{\tau_i,t}$
        end if
    end while
end function
```

Symbol	Explanation
$\mathcal{T}^{\mathtt{alloc}}$	Set of all allocated tasks in this subsystem
$\mathcal{T}^{\mathtt{pre}}$	Set of pre-allocated tasks
$\mathcal{T} \supseteq \mathcal{T}^{\mathtt{pre}} \dot{\cup} \mathcal{T}^{\mathtt{alloc}}$	Set of all tasks in the whole system
$\overline{S}$	Set of unallocated signals sent or received by tasks in $\mathcal{T}$
$S^{\mathtt{alloc}}$	Set of already allocated signals send or received by tasks in $\mathcal{T}$
$S \supseteq \overline{S} \dot{\cup} S^{\mathtt{alloc}}$	Set of all signals sent or received by tasks in $\mathcal{T}$
$S^{\mathtt{user}} \subseteq S$	Set of signals which are forced to be allocated to the local bus during the analysis

Table 5.1. – Symbols used for the encoding of the Spare-Time/MaxWCET Analysis

5.4.1. Objective Function

$$\text{Minimize} \sum_{e\in E}\sum_{t\in\mathbb{E}} \text{cost}(t)x_{e,t} + \sum_{\tau\in\mathcal{T}^{\texttt{pre}}} \text{penalty}(\tau)\bar{y}_\tau \tag{5.23}$$

$$x_{e,t} \in \{0,1\} \qquad e\in E, t\in\mathbb{E} \tag{5.24}$$

$$\bar{y}_\tau \in \{0,1\} \qquad \tau\in\mathcal{T} \tag{5.25}$$

The following free variables are defined: $x_{e,t} = 1$ means that ECU-type t is assigned to ECU e (defined in Equation (5.24)). $\bar{y}_\tau = 1$ means that task τ is in part of the odd set (defined in Equation (5.25)). For simplicity such variables exists not only for all unallocated tasks but also for all already allocated tasks for which the value is always 0.

The objective function in Equation (5.23) minimizes the sum of the total hardware cost and the penalty cost. The total hardware cost is defined to be the sum of the individual costs for each ECU. By choosing a certain ECU-type for an ECU the cost associated to that ECU-type incur. A value of $x_{e,t} = 1$ means that for ECU e the ECU-type t has been chosen. The function cost(t) yields the cost for that ECU-type.

5.4.2. Assignment of ECU-Types and Task Allocation

$$\sum_{e\in E}\sum_{t\in\mathbb{E}} \text{cost}(t)x_{e,t} \le cost^{\texttt{max}}_{sub} \tag{5.26}$$

$$\sum_{t\in\mathbb{E}} x_{e,t} \le 1 \qquad \forall e\in E \tag{5.27}$$

$$x_{e,t} = 0 \qquad \forall e\in E, t\in \mathbb{E}\backslash\text{allowed}^E(e) \tag{5.28}$$

$$y_{\tau,e} = 1 \qquad \forall\tau\in\mathcal{T}^{\texttt{alloc}}, e = \text{taskalloc}_{\mathcal{A}}(\tau) \tag{5.29}$$

$$y_{\tau,e} = 0 \qquad \forall\tau\in\mathcal{T}^{\texttt{alloc}}, e\in E, e\neq \text{taskalloc}_{\mathcal{A}}(\tau) \tag{5.30}$$

$$\bar{y}_\tau + \sum_{e\in E} y_{\tau,e} = 1 \qquad \forall\tau\in\mathcal{T} \tag{5.31}$$

$$\bar{y}_\tau = 1 \qquad \forall\tau\in\mathcal{T}\backslash\left(\mathcal{T}^{\texttt{pre}}\dot{\cup}\mathcal{T}^{\texttt{alloc}}\right) \tag{5.32}$$

$$e_{\tau_1,\tau_2} + 1 \ge y_{\tau_1,e} + y_{\tau_2,e} \qquad \forall e\in E, \tau_1,\tau_2\in\mathcal{T} \tag{5.33}$$

$$e_{\tau_1,\tau_2} \le 1 + y_{\tau_1,e} - y_{\tau_2,e} \qquad \forall e\in E, \tau_1,\tau_2\in\mathcal{T} \tag{5.34}$$

$$e_{\tau_1,\tau_2} \le 1 + y_{\tau_2,e} - y_{\tau_1,e} \qquad \forall e\in E, \tau_1,\tau_2\in\mathcal{T} \tag{5.35}$$

$$e_{\tau_1,\tau_2} \le 2 - \bar{y}_{\tau_1} - \bar{y}_{\tau_2} \qquad \forall\tau_1,\tau_2\in\mathcal{T} \tag{5.36}$$

$$\sum_{\tau\in\mathcal{T}} y_{\tau,e} - \sum_{t\in\mathbb{E}} x_{e,t}\,|\mathcal{T}| \le 0 \qquad \forall e\in E \tag{5.37}$$

$$y_{\tau,e} \in \{0,1\} \qquad \tau\in\mathcal{T}, e\in E \tag{5.38}$$

$$e_{\tau_i,\tau_j} \in \{0,1\} \qquad \tau_i,\tau_j\in\mathcal{T} \tag{5.39}$$

The following free variables are defined: $y_{\tau,e} = 1$ means that task τ is allocated to ECU e (defined in Equation (5.38)). $e_{\tau_i,\tau_j} = 1$ means that both tasks τ_i, τ_j are allocated to the same ECU (defined in Equation (5.39)).

The local analysis has to satisfy the cost limit $cost_{sub}^{\texttt{max}}$. This is enforced by Equation (5.26). Equation (5.27) enforces that maximal one ECU-type is assigned to each of the ECUs and Equation (5.28) disallows every ECU-type which is not in the set of allowed types for the ECU. All tasks which are already allocated to ECUs have to remain there, formalized by Equations (5.29) and (5.30).

Equation (5.31) states that each task may be allocated to maximal one ECU and forces the "penalty switch" $\bar{y}_\tau$ to 1 if task τ is not allocated to any ECU. Equation (5.32) handles a special case: The task set $\mathcal{T}$ also contains all tasks that have been pre-allocated to other subsystems. The equation enforces that they are never allocated to ECUs of this subsystems. Equations (5.33)–(5.36) define for each pair of tasks τ_1, τ_2 a variable e_{τ_1,τ_2} which is 1 exactly if both tasks are allocated to the same ECU. Note that Equation (5.36) handles the special case that both tasks are not allocated to any ECU in which case the binary variable is forced to be 0. Equation (5.37) requires that an ECU must have an ECU-type in order to allocate tasks to it (note: the size of the set of tasks $|\mathcal{T}|$ is used as "big M" here).

5.4.3. Tasks: Effective Worst Case Execution Time

The effective WCET for each allocated task depends on the ECU-type chosen for the ECU it is allocated to. For each of the pre-allocated tasks it is determined as specified in Equations (5.40)–(5.42). Note that tasks get an effective WCET of 0 for all ECUs they are not allocated to (Equation (5.42)). The corresponding variables are defined in Equation (5.43). The inequations use an upper bound for the task's WCET with is defined as stated in Equation (5.44).

$$wcet_{\tau,e}^{\texttt{eff}} + (1 - y_{\tau,e}) wcet_{\tau}^{\texttt{ub}} \geq \sum_{t \in \mathbb{E}} \mathrm{wcet}_p(\tau, t) x_{e,t} \qquad \forall \tau \in \mathcal{T}, e \in E \qquad (5.40)$$

$$wcet_{\tau,e}^{\texttt{eff}} \leq \sum_{t \in \mathbb{E}} \mathrm{wcet}_p(\tau, t) x_{e,t} \qquad \forall \tau \in \mathcal{T}, e \in E \qquad (5.41)$$

$$wcet_{\tau,e}^{\texttt{eff}} \leq wcet_{\tau}^{\texttt{ub}} y_{\tau,e} \qquad \forall \tau \in \mathcal{T}, e \in E \qquad (5.42)$$

$$wcet_{\tau,e}^{\texttt{eff}} \in \mathbb{N} \qquad \tau \in \mathcal{T}, e \in E \qquad (5.43)$$

$$wcet_{\tau}^{\texttt{ub}} := \max_{t \in \mathbb{E}} \{\mathrm{wcet}_p(\tau, t)\} \qquad \tau \in \mathcal{T} \qquad (5.44)$$

5.4.4. Tasks: Spare-Time

Spare-time calculation is used to ensure, that none of the already allocated tasks on an ECU exceeds its deadline if new tasks with higher priorities are allocated. Due to the a-priori defined total order of task with respect to their priorities (defined by $\mathrm{prio}^{>}$, see (5.1)) the (relative) priority of a task newly allocated to an ECU is defined. Let for example tasks τ_1, τ_2 be allocated to the same ECU. An additional task τ_i with a deadline $\mathrm{deadline}_{\mathcal{A}}^{\texttt{task}}(\tau_1) < \mathrm{deadline}_{\mathcal{A}}^{\texttt{task}}(\tau_i) < \mathrm{deadline}_{\mathcal{A}}^{\texttt{task}}(\tau_2)$ would get a higher priority than task τ_2 but a lower priority than task τ_1 if allocated to that ECU. Obviously the absolute priority of τ_i depends on the whole set of allocated tasks on that ECU and can easily be calculated after the optimization step.

$$\begin{aligned} \mathfrak{s}_{\tau,t} + (1 - x_{e,t})\mathfrak{s}^{\mathtt{ub}}_{\tau,t} \geq & & \forall \tau \in \mathcal{T}^{\mathtt{alloc}}, t \in \mathbb{E}, \\ \sum_{\substack{\tau' \in \mathcal{T}^{\mathtt{pre}} \\ \mathrm{prio}^{>}(\tau',\tau)}} \left\lceil \frac{\mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)}{\mathrm{period}^{\mathtt{task}}(\tau')} \right\rceil wcet^{\mathtt{eff}}_{\tau',e} & & e{:=}\mathrm{taskalloc}_{\mathcal{A}}(\tau) \end{aligned} \tag{5.45}$$

In Equation (5.45) the spare-time condition is formulated with linear equations. For each ECU-type and each allocated task, $\mathfrak{s}_{\tau,t}$ denotes the pre-calculated spare-time value. Obviously, only the spare-time for the currently chosen ECU-type for the ECU where a task is allocated is relevant. On the left side of the inequation a constant larger than the maximal possible sum of preemption duration, as calculated on the right side of the equation, is added for every ECU-type which has NOT been chosen for the ECU where task τ has been allocated. This constant ensures that the inequations is inherently satisfied for all the ECU-types which have not been chosen (and therefore have to be ignored). Those constants are defined as in Equation (5.46):

$$\mathfrak{s}^{\mathtt{ub}}_{\tau,t} {:=} \sum_{\substack{\tau' \in \mathcal{T}^{\mathtt{pre}} \\ \mathrm{prio}^{>}(\tau',\tau)}} \left\lceil \frac{\mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)}{\mathrm{period}^{\mathtt{task}}(\tau')} \right\rceil \mathrm{wcet}_p(\tau', t) \qquad \forall \tau \in \mathcal{T}^{\mathtt{alloc}}, t \in \mathbb{E} \tag{5.46}$$

On the right hand side of Equation (5.45) the sum of the capacity requirements of the pre-allocated tasks is calculated. The capacity requirement of a pre-allocated task τ' is relevant only if that task would have a higher priority than the allocated task τ. Therefore only those pre-allocated tasks are considered where $\mathrm{prio}^{>}(\tau, \tau')$ is true.

Note that the number of preemptions of task τ by pre-allocated task τ' during the maximal allowed runtime of τ is constant due to the over-approximation based on the allocated task's deadline $\mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)$ instead of the actual worst case response time:

$$\left\lceil \frac{\mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)}{\mathrm{period}^{\mathtt{task}}(\tau')} \right\rceil$$

The only free variable on the right side is the effective WCET of the pre-allocated tasks. The effective WCET of a task is only greater than 0 for the ECUs the task is allocated to.

5.4.5. Tasks: MaxWCET

MaxWCET values for tasks specify for a given pseudo deadline and a given ECU how much capacity would be available for preemptions by other pre-allocated tasks with higher priority. The calculation depends on their WCETs which is determined by the chosen ECU-types of the ECU they are allocated on.

$$\begin{aligned}
&\mathfrak{m}_{\hat{d},e,t} + (1 - x_{e,t})\mathfrak{m}^{\mathtt{ub}}_{\tau,t} + (1 - y_{\tau,e})\mathfrak{m}^{\mathtt{ub}}_{\tau,t} \geq && \forall \tau \in \mathcal{T}^{\mathtt{pre}}, t \in \mathbb{E}, e \in E, \\
&\mathit{wcet}^{\mathtt{eff}}_{\tau,e} + \sum_{\substack{\tau' \in \mathcal{T}^{\mathrm{pre}} \\ \mathrm{prio}^{>}(\tau',\tau)}} \left\lceil \frac{\hat{d}_\tau}{\mathrm{period}^{\mathtt{task}}(\tau')} \right\rceil \mathit{wcet}^{\mathtt{eff}}_{\tau',e} && \hat{d}_\tau = \mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)
\end{aligned} \tag{5.47}$$

MaxWCET values have been pre-calculated for each of the pseudo deadlines, each of the ECUs and each ECU-type. Those values appear as $\mathfrak{m}_{\hat{d},e,t}$ (for a pseudo deadline $\hat{d}$, an ECU e and an ECU-type t) on the left hand side of Equation (5.47). The inequation is made vacuously satisfied by adding an upper bound for the required MaxWCET values $\mathfrak{m}^{\mathtt{ub}}_{\tau,t}$ if a different ECU-type has been chosen for the ECU (left hand side: second term) or if the task τ has not been allocated to ECU e. On the right hand side of the equation the own WCET of task τ appears as first term. A second term is added to the first one which holds the number of preemptions potentially caused by the all other unallocated tasks which would have a higher priority on the same ECU multiplied by their respective effective worst case execution time. Again, the number of preemptions is a constant because the deadline of task τ is used for calculating them instead of the worst case response time as in the original fixed-point equation (see Equation (2.1)).

The upper bound constants are defined as in Equation (5.48).

$$\begin{aligned}
\mathfrak{m}^{\mathtt{ub}}_{\tau,t} :=& \mathrm{wcet}_p(\tau, t) + \\
& \sum_{\substack{\tau' \in \mathcal{T}^{\mathrm{pre}} \\ \mathrm{prio}^{>}(\tau',\tau)}} \left\lceil \frac{\mathrm{deadline}^{\mathtt{task}}_{\mathcal{A}}(\tau)}{\mathrm{period}^{\mathtt{task}}(\tau')} \right\rceil \mathrm{wcet}_p(\tau', t) \qquad \forall \tau \in \mathcal{T}^{\mathtt{pre}}, t \in \mathbb{E}
\end{aligned} \tag{5.48}$$

5.4.6. Tasks: Memory

$$\mathrm{mem}(t) + (1 - x_{e,t})\mathrm{mem}^{\mathtt{ub}}(t) \geq \sum_{\tau \in \mathcal{T}} \mathrm{memreq}_p(\tau, t) y_{\tau,e} \qquad \forall e \in E, t \in \mathbb{E} \tag{5.49}$$

In Equation (5.49) the memory constraints for ECUs are formulated. For every ECU the available memory of every ECU-type mem(t) has to be greater than or equal to the sum of the memory required by all tasks on that ECU. An upper bound for the sum of the maximal required memory $\mathrm{mem}^{\mathtt{ub}}(t)$ is added to the size of available memory (left hand side of the equation) for all the ECU-types which have not been chosen for the ECU. On the right side the sum of required memory size is calculated, but only for those tasks which are actually allocated to the ECU (factor $y_{\tau,e}$ is 0 otherwise). The term $\mathrm{memreq}_p(\tau, t)$ represents the constant memory requirement of a task τ if allocated to an ECU-type t. The upper bound is defined in Equation (5.50):

$$\text{mem}^{\text{ub}}(t) := \sum_{\tau \in \mathcal{T}} \text{memreq}_p(\tau, t) \qquad \forall t \in \mathbb{E} \tag{5.50}$$

5.4.7. Task Allocation Constraints

For a given constraint specification tuple of DSE problem p

$$constr = (\psi^{\texttt{ECUs}}, \psi^{\texttt{types}}, \psi^{\texttt{never}}.\psi^{\texttt{always}}, \psi^{\texttt{allowedSubsys}}, \psi^{\texttt{sameSubsys}}, \psi^{\texttt{diffSubsys}}, \psi^{\texttt{onBus}})$$

the allocation constraints defined in Section 3.1.4 on page 38 are encoded as follows:

$$\mathfrak{t}_{\tau,t} \geq x_{e,t} + y_{\tau,e} - 1 \qquad \forall \tau \in \mathcal{T}, e \in E, t \in \mathbb{E} \tag{5.51}$$

$$\mathfrak{t}_{\tau,t} \leq x_{e,t} + (1 - y_{\tau,e}) \qquad \forall \tau \in \mathcal{T}, e \in E, t \in \mathbb{E} \tag{5.52}$$

$$\sum_{t \in \mathbb{E} \setminus \psi^{\texttt{types}}(\tau)} \mathfrak{t}_{\tau,t} = 0 \qquad \tau \in \mathcal{T}^{\texttt{unalloc}}_{conf} \tag{5.53}$$

$$\sum_{\tau \in X} \sum_{e \in E} y_{\tau,e} \leq 1 \qquad \forall X \in \psi^{\texttt{diffSubsys}} \tag{5.54}$$

$$\bar{y}_{\tau_0} \cdot (|X| - 1) - \sum_{\tau \in X \setminus \{\tau_0\}} \bar{y}_\tau = 0 \qquad \forall X = \{\tau_0, \ldots, \tau_n\} \in \psi^{\texttt{sameSubsys}} \tag{5.55}$$

$$\sum_{e \in E \setminus \psi^{\texttt{ECUs}}(\tau)} y_{\tau,e} = 0 \qquad \forall \tau \in \mathcal{T}^{\texttt{pre}} \tag{5.56}$$

$$y_{\tau_0,e} \cdot (|X| - 1) - \sum_{\tau \in X \setminus \{\tau_0\}} y_{\tau,e} = 0 \qquad \forall e \in E, X = \{\tau_0, \ldots, \tau_n\} \in \psi^{\texttt{always}} \tag{5.57}$$

$$\sum_{\tau \in X} y_{\tau,e} \leq 1 \qquad \forall e \in E, X \in \psi^{\texttt{never}} \tag{5.58}$$

$$\mathfrak{t}_{\tau,t} \in \{0, 1\} \qquad \forall \tau \in \mathcal{T}, t \in \mathbb{E} \tag{5.59}$$

New auxiliary variables are defined in Equation (5.59) for every task/ECU-type which equal to 1 if a task τ is assigned to an ECU of the current subsystem for which the ECU-type t has been chosen, otherwise 0. This is formalized by Equation (5.51) which forces the value to 1 if the task is allocated to an ECU with the ECU-type and Equation (5.52) which forces the variable to 0 otherwise.

Based on these auxiliary variables, Equation (5.53) states that tasks must not be allocated to an ECU to which an ECU-types has been assigned, which is forbidden for that task. Equation (5.54) encodes that for each of the sets of tasks which must not be allocated to the same subsystem maximal one task per set is allocated to an ECU in the current subsystem. The counterpart of this constraint is encoded in Equation (5.55) which states that for each set of task which must be allocated to the same subsystem either all of the contained tasks or none of them are in the oddset. Equation (5.56) enforces that tasks are never allocated to ECUs which are forbidden for them by the respective constraint. The Equation (5.57) implements the constraint enforcing that specified sets of tasks are always allocated to the same ECU: for each ECU in the current subsystem either all tasks in the set must be allocated to that ECU or none. Finally Equation (5.58) encodes — similar to the subsystem constraints above — that of each specified sets of tasks which must not be allocated to the same ECU maximal one task is allocated to each of the ECUs.

5.4.8. Signals

Signals are represented as messages on local communication buses only if the transmission of a signal is required due to the distributed allocation of the sender and receiver tasks. In this section some properties which are actually properties of a message are used by referring to the corresponding signal for simplicity. For example the worst case transmission time of a signal s on the local bus actually refers to the worst case transmission time of the message m which corresponds to that signal. This is possible due to the precondition that exactly one local bus exists in the subsystems to be analyzed and that every signal corresponds to maximal one message per local bus.

$$\bar{e}_{\tau_1,\tau_2} + 1 \geq y_{\tau_1,e} + y_{\tau_2,e} \qquad \forall \tau_1, \tau_2 \in \mathcal{T}, e \in E \backslash \{e^{\mathtt{gw}}\} \tag{5.60}$$
$$\bar{e}_{\tau_1,\tau_2} \leq 1 + y_{\tau_1,e} - y_{\tau_2,e} \qquad \forall \tau_1, \tau_2 \in \mathcal{T}, e \in E \backslash \{e^{\mathtt{gw}}\} \tag{5.61}$$
$$\bar{e}_{\tau_1,\tau_2} \leq 1 + y_{\tau_2,e} - y_{\tau_1,e} \qquad \forall \tau_1, \tau_2 \in \mathcal{T}, e \in E \backslash \{e^{\mathtt{gw}}\} \tag{5.62}$$
$$\bar{e}_{\tau_1,\tau_2} \leq \sum_{e \in E \backslash e^{\mathtt{gw}}} (y_{\tau_2,e} + y_{\tau_1,e}) \qquad \forall \tau_1, \tau_2 \in \mathcal{T} \tag{5.63}$$
$$g_\tau + \sum_{e \in E \backslash \{e^{\mathtt{gw}}\}} y_{\tau,e} = 1 \qquad \forall \tau \in \mathcal{T} \tag{5.64}$$
$$\bar{g}_{\tau_1,\tau_2} \geq g_{\tau_1} + g_{\tau_2} - 1 \qquad \forall \tau_1, \tau_2 \in \mathcal{T} \tag{5.65}$$
$$\bar{g}_{\tau_1,\tau_2} \leq g_{\tau_1} \qquad \forall \tau_1, \tau_2 \in \mathcal{T} \tag{5.66}$$
$$\bar{g}_{\tau_1,\tau_2} \leq g_{\tau_2} \qquad \forall \tau_1, \tau_2 \in \mathcal{T} \tag{5.67}$$
$$\bar{e}_{\tau_1,\tau_2} \in \{0,1\} \qquad \tau_1, \tau_2 \in \mathcal{T} \tag{5.68}$$
$$g_\tau \in \{0,1\} \qquad \tau \in \mathcal{T} \tag{5.69}$$
$$\bar{g}_{\tau_i,\tau_j} \in \{0,1\} \qquad \tau_i, \tau_j \in \mathcal{T} \tag{5.70}$$

Some additional free variables are defined: $\bar{e}_{\tau_1,\tau_2} = 1$ means that both tasks τ_1, τ_1 are allocated to an ECU of the current subsystem which is not the gateway ECU (defined in Equation (5.68)). g_τ means that task τ is either on the gateway ECU or in another subsystem (defined in Equation (5.69)). These variables are complemented by variables of the form $\bar{g}_{\tau_i,\tau_j}$ which — if set to 1 — mean that both tasks τ_i, τ_j are either on the gateway or in another subsystem but not necessarily on the same resource (defined in Equation (5.70)).

For encoding the conditions under which a signal must have a corresponding message on the subsystem local bus as formalized in Predicate (3.5) (page 48), it is necessary to know whether or not two given tasks (e.g. τ_i, τ_j are allocated to the same non-gateway subsystem ECU. If this is the case then the variable $\bar{e}_{\tau_1,\tau_2} = 1$ as formalized in Equations (5.60)–(5.63). The predicate itself is encoded in Equations (5.71)–(5.73).

Equation (5.64) ensures that variable g_τ equals to 1 if task τ (without loss of generality) is not allocated to any of the non-gateway ECUs of the current subsystem, otherwise that variable is forced to be 0. Equations (5.65) –(5.67) define a derived variable which is 1 exactly if two tasks are both allocated to either the gateway ECU or to other subsystems.

$$h_s - h^{\mathtt{user}}_{s,b} \geq 0 \qquad \forall s \in S \tag{5.71}$$

$$h_s + \bar{g}_{\tau^{\mathtt{send}},\tau^{\mathtt{recv}}} + \bar{e}_{\tau^{\mathtt{send}},\tau^{\mathtt{recv}}} \geq 1 \qquad \forall s \in S, \tau^{\mathtt{recv}} \in \mathrm{recv}(tn, s), \; \tau^{\mathtt{send}} := \mathrm{sender}(tn, s) \tag{5.72}$$

$$h_s \leq h^{\mathtt{user}}_{s,b} + |X| - \sum_{\tau \in X} \left(\bar{e}_{\tau^{\mathtt{send}},\tau} + \bar{g}_{\tau^{\mathtt{send}},\tau}\right) \qquad \forall s \in S, \tau^{\mathtt{send}} := \mathrm{sender}(tn, s), \; X := \mathrm{recv}(tn, s) \tag{5.73}$$

$$h_s \in \{0, 1\} \qquad s \in S \tag{5.74}$$

First a set of new binary variables stating whether or not a signal has to be allocated via a message to the local bus is defined in Equation (5.74) (a value of 1 means that the signal has to be allocated).

Then the Predicate (3.5) is encoded: Equation (5.71) states that a signal always has a message on the local bus if required by the user (by setting $h^{\mathtt{user}}_{s,b} = 1$ for a signal s and a bus b). Equation (5.72) enforces the signal to be on the bus if the sender task and one receiver task are not on the same ECU (right hand side) and not both on the gateway, unallocated or in other subsystems (left hand side). The case where a signal must not be allocated to the local bus is formalized in Equation (5.73). Variable h_s has to be 0 if the user does not force the signal s onto the local bus and the number of receiver tasks is equal to the number of sender task/receiver task combinations where both the sender and the receiver task are either on the same non-gateway ECU in the subsystem, or any combination of: Both are unallocated, or on the gateway ECU, or in another subsystem.

5.4.9. Signals: Spare-Time

As seen in the last sections the handling of signals/messages on CAN buses is very similar to that of tasks on FPS-scheduled ECUs. The approach described in Section 5.2.5.1 is encoded via MILP in the next paragraphs.

Note that m denotes the message corresponding to signal s. For every signal in the set of signals allocated to the local bus $S^{\mathtt{alloc}}$ a spare-time value has been calculated a-priori. As the transmission of a message on the bus cannot be interrupted and may block the transmission of higher priority messages for a certain duration, all spare-time values consider an upper bound for the blocking time already (see Section 5.2.5.1 for details).

Let $\mathfrak{b}$ denote the bus type of the current subsystem's local bus.

$$\mathfrak{s}_{s,b} \geq \sum_{\substack{s' \in \overline{S} \\ \text{prio}^{>}(s',s)}} \left\lceil \frac{\text{deadline}^{\texttt{msg}}_{conf}(m)}{\text{period}^{\texttt{sig}}(s')} \right\rceil \text{wctt}_p(s', \mathfrak{b})\, h_{s'} \qquad \forall s \in S^{\texttt{alloc}} \tag{5.75}$$

Equation (5.75) enforces for every signal already allocated to the local bus that its spare-time (on the left hand side of the inequation) is larger or equal to the preemption durations of all signals which are additionally allocated to the bus. On the right side of the inequation the sum of all preemption durations is calculated. For this calculation only those signals are relevant that are included in the set of pre-allocated signals $\overline{S}$ for the subsystem. Inside of the sum operator only those pre-allocated signals are considered which would have a higher priority than the currently evaluated allocated signal (where $\text{prio}^{>}(s', s) = 1$). For each pre-allocated signal s' the number of preemptions it would cause on the allocated signal is over-approximated by multiplying the ceiling of the deadline of the allocated signal divided by the period of the pre-allocated signal with the (constant) worst case transmission time of the pre-allocated signal and the auxiliary binary variable $h_{s'}$. This variable is set to 1 by the solver only if that signal has to be allocated to the local bus as implied by the allocation of its sender and receiver tasks.

5.4.10. Signals: MaxWCET

$$\begin{aligned} &\mathfrak{m}_{\hat{d},b} + \mathfrak{m}^{\texttt{ub}}_{s}(1 - h_s) \geq \text{wctt}_p(s, \mathfrak{b}) + \\ &\quad \sum_{\substack{s' \in \overline{S} \\ \text{prio}^{>}(s',s)}} \left\lceil \frac{\hat{d}}{\text{period}^{\texttt{sig}}(s')} \right\rceil \text{wctt}_p(s', \mathfrak{b})\, h_{s'} \qquad \forall s \in \overline{S}, \hat{d} = \text{period}^{\texttt{sig}}(s) \end{aligned} \tag{5.76}$$

Equation (5.76) expresses that the pre-calculated MaxWCET value for every signal which has been allocated during the optimization process to the bus has to be larger than its worst case transmission time $\text{wctt}_p(s, \mathfrak{b})$ and the sum of the preemption durations. On the left hand side of the equation an upper bound is used to vacuously satisfy the inequation whenever a signal is not allocated to the local bus. Note the use of the signal's period instead of its (not yet known) deadline. This is a safe over-approximation as all deadlines are required to be smaller or equal to the periods. The upper bound is defined as in Equation (5.77):

$$\begin{aligned} &\mathfrak{m}^{\texttt{ub}}_{s} := \text{wctt}_p(s, \mathfrak{b}) + \\ &\quad \sum_{\substack{s' \in \overline{S} \\ \text{prio}^{>}(s',s)}} \left\lceil \frac{\hat{d}}{\text{period}^{\texttt{sig}}(s')} \right\rceil \text{wctt}_p(s', \mathfrak{b}) \qquad \forall s \in \overline{S}, \hat{d} = \text{period}^{\texttt{sig}}(s) \end{aligned} \tag{5.77}$$

This formula is very similar to the right side of Equation (5.76) with the difference that it is assumed that all lower-priority signals are allocated to the local bus which results in the largest value the right hand side of Equation (5.76) could possibly take.

5.5. Alternative Approaches

5.5.1. “Eis” (Eisenbrand) Approach

This approach is named (in this thesis) after one of the authors of [Eis+06], where a concept to model the fixed-point equation for schedulability analysis on ECUs with fixed-priority preemptive schedulers is proposed. This approach has been reimplemented for this thesis. For this reimplementation, many MILP formulations presented in the previous section have been reused with only little changes or none at all, e.g. the task allocation constraints, the inequations for describing whether or not a signal has to be allocated to the local bus, etc. Of course, the equations for spare-time and MaxWCET analysis have not been reused. In [Eis+06], the fixed-point equation is represented directly by calculating for each task and each signal lower and upper bounds for the response times as part of the optimization process. It is left up to the MILP solver to choose the actual response times satisfying those bounds.

5.5.2. JoSe: Column Generation Approach

This approach has been published in [ANT11]. It is based on linear programming and uses the same notion for computation capacity as the previous one (the “Eisenbrand” constraints) but aims for improving scalability by utilizing the well-known technique of column generation. The column generation approach starts with a small set of variables (columns) and solves a so called master problem first. This problem does not care for the actual schedulability of ECUs and the local bus directly, but only allocates tasks to ECUs and signals to the local bus if required. The check for schedulability is left to the so called pricing problems, which are solved separately for each ECU and for the local bus. The result of a pricing problem might indicate that certain variables (e.g. representing the allocation of a task to a certain ECU which are not yet represented in the master problem would lead reduce the total cost. If this happens, those variables are added to the master problem. Please refer to the aforementioned publication for more details.

5.5.3. RTSAT: SMT Solver with Scheduling Theory

The third alternative approach for the local analysis problem has been published in [Met+06]. This approach is based on Satisfiability Modulo Theories (SMT) with a theory module specialized on performing schedulability analysis of ECUs scheduled by fixed-priority preemptive schedulers. Unfortunately the available version of the tool “rtsat” was not working reliable enough to be considered for the evaluation of the local analysis approaches (see Chapter 6). Please refer to the aforementioned publication for details.

5.6. Comparison of Approaches for Local Analysis

Table 5.2 provides an overview on the tools implementing the presented local analysis approaches and the set of features supported by them. Note that “Stm” refers to the implementation of the spare-time and MaxWCET approach presented in this thesis.

Feature / Approach	**“Stm”**	**“Eis”**	**“JoSe”**	**“RTSAT”**
Minimize hardware cost	●	●	●	–
Limit hardware cost	●	●	●	●
Allowed ECUs per Task	●	●	●	–
Allowed ECU Types per Task	●	●	●	–
Never on same ECU	●	●	●	–
Always on same ECU	●	●	●	–
Allowed ECUs	●	●	●	–
Forbidden ECUs	●	●	●	–
Signal always on local bus	●	●	●	–
Other Features				
One WCET per ECUType	●	●	●	●

Table 5.2. – Features of Modules for Local Analysis
(●) fully supported, (○) partially supported, (–) unsupported

It can be seen that all tools except for the “RTSAT” tool have an identical set of supported features. This is because in contrary to the “RTSAT” tool all the other tools are under active development which makes it possible to add missing features quickly. In Chapter 6 more detailed information is presented about the technical realization of the “Stm” and “Eis” local analysis modules.

6. Implementation and Evaluation

This chapter describes the software implementation of the methods proposed in the previous chapters, compares them with the alternative approaches based on a set of benchmarks and then discusses the results using empirical methods.

For each of the methods proposed in this thesis several properties like runtime and solution quality in terms of the costs of the solutions are evaluated for all of the benchmark models. The benchmarks consist of models derived by modifying examples found in academic publications, models generated randomly following different rules for creation of quasi-realistic application, and a set of models based on a full-featured control application developed by students of the University of Oldenburg as part of their collegiate working group.

6.1. Implementation

The approaches presented in the preceding chapters have been implemented as part of the thesis in an integrated tool named **Zerg**.

Figure 6.1 shows the software architecture of Zerg. The black arrows depict interfaces between Zerg modules and external software applications. Zerg consists of a set of meta-models based on a common core (not shown in the figure, contains amongst others an implementation of a hierarchical graph), a light-weight process engine with several modules for reading/writing (XML) file formats, and interface layers to multiple SMT and MILP solvers. The whole software is written in C++ and makes use of the QT framework (version 4.x). Additionally, some analysis modules for local analysis are using GLPK (see [Mak10]) and are implemented in the modeling language *MathProg* which is a subset of the AMPL modeling language (see [LLC12]). Statistics on the source code are shown in Table 6.1.

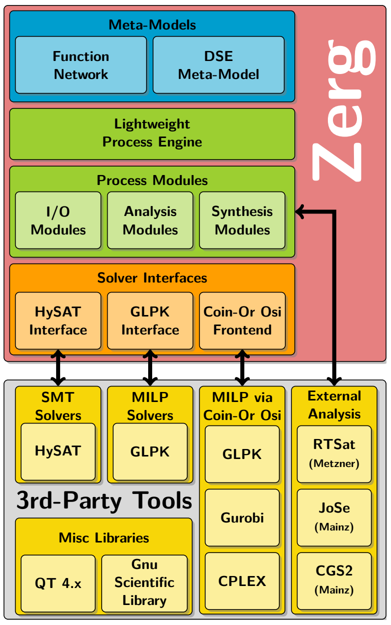


Figure 6.1. – Zerg Software Architecture

Currently Zerg supports two different meta-models: The DSE meta-model is used for the specification of task networks, hardware architectures and mappings between both. The function network meta-model is — as the name suggests — for specifying function networks (see 2.2.2). Both meta-models are

Language	**Code Metric**	**Value**
C++	Number of Classes[1]	185
	Lines of Code (non-blank, non-comment)[1]	92383
	Comment lines[1]	15130
MathProg	Lines of Code (non-blank, non-comment)[2]	3346
	Comment lines[3]	1990

Table 6.1. – Zerg Source Code Statistics (on September 4, 2012)

[1] Measured by using *cccc*, see http://cccc.sourceforge.net

[2] Measured by `'grep -v -e "^[[:space:]]*#.*$" -e "^[[:space:]]*$" -e "^[[:space:]]*/\*" *.glpk| wc -l'`

[3] Measured by `'grep -e "^[[:space:]]*#.*$" -e "^[[:space:]]*/\*" *.glpk| wc -l'`

based on the implementation of hierarchical graphs contained in the common core of Zerg.

The process engine allows the user to freely define the order of execution of one or more instances of the provided process modules and to set appropriate parameters for each process module. The available process modules are categorized into I/O modules for reading and writing of model files, analysis modules (e.g. for schedulability analysis) and synthesis modules (e.g. for the generation of artificial benchmark models). In a typical scenario first some files are read, e.g. a task network, a hardware architecture and a mapping between both, then some analyses are performed, and finally the result is written to a set of files. The analysis and synthesis modules that have been created and used for this thesis are described in detail in Section 6.2.

Some analysis modules make use of external backends, e.g. open-source and commercial MILP solvers.

6.1.1. External Backends

The following sections describe some details concerning the choice and integration of the backend solvers used in Zerg.

6.1.1.1. Satisfiability Modulo Theories Solver

Two Satisfiability Modulo Theories (SMT) solvers have been evaluated for this thesis: HySAT (see [Her12]) and Yices (see [Tea12]). Yices was initially seen as an alternative to HySAT but has been dropped due to its rather complex file syntax. In contrary, HySAT has the huge advantage that its source code is available, because it has been developed at the University of Oldenburg. Some required but initially missing features could be implemented leading to a reentrant fork of the original HySAT.

6.1.1.2. MILP Solver

Several MILP solver backends are used in Zerg, namely the GNU Linear Programming Kit, ILOG CPLEX and Gurobi.

GNU Linear Programming Kit (GLPK) The GNU Linear Programming Kit (GLPK) was the first tool considered as alternative to the SMT-based approaches. GLPK is open source software and was therefore available on all platforms under consideration: Linux and Windows (XP or higher). The Linux distributions Debian ([Deb]) and Ubuntu ([Can]) both directly provide binary archives which made this MILP solver the first choice. The API is easy to use and especially the supported MathProg language is a great plus.

Note that GLPK has been integrated into Zerg using two different APIs: First, the API provided by GLPK itself has been used. This is accomplished by linking GLPK directly to the Zerg binary. Second, by linking the COIN-OR Osi API (see next section) to Zerg automatically made GLPK available through this generic API as well. Here, GLPK is dynamically linked to Zerg indirectly through the COIN-OR Osi library along with all the other MILP solvers supported by COIN-OR Osi. This has the great advantage that Zerg may even detect that the GLPK library is unavailable for some reason (e.g. because Zerg has been shipped to an industrial user who uses one of the commercial MILP solvers exclusively).

COIN-OR Osi The COIN-OR (COmputational INfrastructure for Operations Research) project is dedicated to providing open-source software for operations research. COIN-OR subprojects provide a number of stand-alone optimization tools and also interfaces to third-party products for the analysis of deterministic linear and non-linear problems, stochastic problems, non-differentiable problems, and more.

The software developed as part of this thesis makes use of the COIN-OR Osi (Open Solver Interface) subproject which provides a standard interface for a number of open-source and commercial-grade optimization products. For this thesis, the commercial solvers CPLEX and Gurobi and the open-source solver GLPK have been used. All the named solvers have been evaluated for this thesis. For this thesis, the official release of COIN-OR Osi has been extended by implementing some missing features.

6.2. Experimental Setup

The experimental evaluation aims at providing reliable data concerning the quality and efficiency of the approaches proposed as part of this thesis in comparison to some of the alternative approaches. Naturally, whether or not a given tool can be evaluated depends on several factors, such as the functionality it provides, the usability of its API (if any), its availability, and its reliability.

6.2.1. Hardware and Software

All experiments have been carried out on the same multi-core server. Some details about the hardware configuration can be found in Table C.1 (page 195), details about the installed software are given in Table C.2 (page 195).

6.2.2. Analysis Modules

A set of modules is available for validating the models used as input for the analysis modules to be evaluated and also the computed solutions. The Zerg analysis modules used for the experimental evaluation can be categorized as follows.

`DSESchedulingAnalysis` performs a (non-holistic) schedulability analysis for all the FPS-scheduled ECUs and all local priority-based buses

`DSEMemoryAnalysis` is responsible for validating that there is enough RAM available on each ECU for all the software tasks allocated to it.

`DSEConsistencyAnalysis` validates the correctness of the allocation, mainly that signals are always allocated to local buses and the global bus correctly if this is required according to the allocation of their sender/receiver tasks.

Statistical data is collected by two modules:

`DSEGreatComparator` is responsible for the global evaluation. It collects data of the global analysis runs.

`DSELocalAnalysisEvaluation` is responsible for the local evaluation. It collects data of local analysis runs.

Both modules accept a list of analysis modules to be run as input together with parameters to be used for those modules. During the evaluation process the global evaluation module sequentially executes all specified modules. The local evaluation module is run by the global analysis modules each time they start a local analysis run. The local evaluation module then executes one or more local analysis modules.

However, all global analysis modules expect to get exactly one local analysis result after the local analysis has finished, not multiple ones. For this reason a so-called **local analysis reference module** is defined for the local analysis evaluation module. All local analysis runs are evaluated as usual but only the result of the local analysis reference module is handed back to the calling global analysis module. Of course, running multiple local analysis modules skews the timing measurement of the global analysis which does not know that multiple local analysis modules are to be run and compared. The global analysis evaluation module takes care of this bias by keeping track of all the time intervals the local analysis evaluation has spent on other modules than the local reference module. Those time intervals are later subtracted from the total runtime of the global analysis module under evaluation. Each global/local analysis result is tested for schedulability, feasibility of the memory constraints and consistency of the allocation after the respective analysis module finished its execution.

The powerful statistics software **R** (see [Ano12]) has been used to examine the result data, further aggregate the results and produce the numerous figures and tables presented in the following sections.

The analysis modules and their characteristics are described in detail in the following sections. Table 6.2 compares the features of the different global analysis modules.

Approach / Feature	**Global MILP-based Analysis**		**Kernighan-Lin**	**MILP+Column Generation**
	HySAT	**MILP**		
Objectives				
Minimize HW-Cost	●	●	●	●
Minimize Number of Global Bus Slots	○	○	●	●
Constraints				
Allowed ECU Types per Task	●	●	○	●
Never on same ECU	L	L	L	●
Always on same ECU	○	○	○	●
Allowed ECUs	○	○	○	●
Forbidden ECUs	L	L	○	●
Allowed Subsystems	●	●	●	●
Forbidden Subsystems	●	●	●	●
Signal always on global bus	●	●	●	●

Table 6.2. – Features of Modules for Global Analysis / Analysis of Complete Models (●) fully supported, (○) partially supported, (L) realizable only by local analysis,(–) unsupported

The table shows that all global analysis modules that are based on the two-tier optimization approach described in this thesis are unable to handle certain constraint types (marked as L). A good example is the constraint type "Allowed ECUs" which allows to define a set of ECUs for each task in order to enforce that the task is allocated only to one of them. All global analysis modules based on the two-tier approach do not allocate tasks to ECUs directly, but rather pre-allocate them to subsystems. Obviously these analysis modules cannot fulfill such a constraint because the actual allocation of tasks to ECUs is left to the local analysis. Nevertheless, derived information can be used on the global tier.

6.2.2.1. Overall Analysis Module

There is only one module in this category. Its purpose is to optimally solve the DSE problem as described in Chapter 3 without dividing the overall problem into subproblems following the two-tier optimization approach described in this thesis.

"MZ": Overall Analysis based on MILP with Column Generation This analysis module has been developed at the University of Mainz as part of the collaboration in the Transregional Collaborative Research Center 14 AVACS (see [Bec+12]). It takes as input a task network, a hardware architectural pattern, a partial allocation, and (optionally) constraints regarding the allocation and the choice of ECU-types for the ECUs contained in the hardware architectural pattern. The analysis module either returns a complete allocation or exits unsuccessfully if the DSE problem is infeasible, a time-out occurred, or an error condition has been identified. The implementation is based on a formulation of the whole DSE optimization problem as MILP problem and additionally uses a technique

called column generation, which exploits — similar to the two-tier optimization approach proposed in this work — that different levels of granularity are introduced by the notion of subsystem. The module is capable of both calculating complete schedules for the global bus and using a simplification which only ensures that the minimal required number of bus slots is available per allocated message. The details of this approach are described in [Alt+12].
The Zerg module name is `DSEOverallAnalysisMainz`

6.2.2.2. Global Analysis Modules

This set of analysis modules is based on the two-tier optimization concept as described in this thesis. Each analysis module realizes either the global analysis approach presented in this work or the concurring approach presented in [Bük+11a].

"LP_*": Global Analysis with ECU Type Bins based on MILP ("LP_GRB": Gurobi, "LP_CPX": CPlex, "LP_GLP": GLPK) This analysis module takes as input a task network, a hardware architectural pattern, a partial allocation, and (optionally) constraints regarding the allocation and the choice of ECU-types for the ECUs contained in the hardware architectural pattern. The details of this approach are described in Chapter 4. The analysis module either returns a complete allocation or exits unsuccessfully if the DSE problem is infeasible, a time-out occurred, or an error condition has been identified. This implementation is based on problem formulation as MILP problem. One of the parameters defines which local analysis module to use. The use of the COIN-OR Osi abstraction layers enables the use of all MILP solver backends currently supported by COIN-OR Osi without changing the Zerg module. The number of used global bus slots can be minimized, but this implementation is not capable of calculating complete feasible schedules for the global bus.
The Zerg module name is `DSEGlobalAnalysisMILP`.

"HS": Global Analysis with ECU Type Bins based on HySAT This analysis module takes as input a task network, a hardware architectural pattern, a partial allocation, and (optionally) constraints regarding the allocation and the choice of ECU-types for the ECUs contained in the hardware architectural pattern. The details of this approach are described in Chapter 4. The analysis module either returns a complete allocation or exits unsuccessfully if the DSE problem is infeasible, a time-out occurred, or an error condition has been identified. This implementation is based on a problem formulation as SMT problem with "linear arithmetic" theory module. One of the parameters defines which module for performing the local analysis for each subsystem is used.
The Zerg module name is `DSEGlobalAnalysisHysat`.

"KL": Global Analysis based on a Kernighan-Lin partitioning algorithm The module takes as input a task network, a hardware architectural pattern, a partial allocation, and (optionally) constraints regarding the allocation and the choice of ECU-types for

the ECUs contained in the hardware architectural pattern. The details of this approach are described in Chapter 4. The analysis module either returns a complete allocation or exits unsuccessfully if the DSE problem is infeasible, a time-out occurred, or an error condition has been identified. Unfortunately, the current implementation does not provide information to the global evaluation module about whether it exited due to an error condition or because it was unable to find a solution.
The Zerg module name is `DSEGlobalAnalysis`.

6.2.2.3. Local Analysis Modules

Currently, there are four different local analysis modules, namely the Spare-Time / MaxWCET module presented in this work, the module based on the "Eis" constraints (see [Eis+06]), the tool "JoSe" (a local analysis module based on column generation written at the University of Mainz) and RTSAT. In this thesis only two of them could be compared: The Spare-Time/MaxWCET module and the Eisenbrand module. The JoSe module has not been evaluated in this thesis because a previous evaluation in [Tha+10] has shown that for small local analysis problems the applied column generation approach is not beneficial due to the overhead costs. The RTSAT module could not be evaluated because it does not support many of the evaluated features.

"Stm": Spare-Time/MaxWCET with GLPK+Gurobi This tool uses a pre-analysis step for characterizing the available capacity on ECUs with fixed-priority preemptive schedulers using the notion of spare-time and MaxWCET values. Using this abstraction, the problem of finding task/message allocations with minimal hardware modification cost is reduced to a combinatorial optimization problem. The current implementation uses GLPK for creating the MILP model but the commercial solver Gurobi for solving the model. Note that the runtime measured for this analysis module contains all the times used for the individual steps starting from the pre-analysis to the call to the Gurobi backend (if used) to the preparation of the result model (also including all file I/O operations).
The Zerg module name is `DSEMaxWCETSparetimeGLPK`

"Eis": "Eisenbrand" Constraints with GLPK+Gurobi The MILP formulation of this local analysis module is similar to the one for the Spare-Time/MaxWCET analysis module. However, there is no need to distinguish allocated tasks and un-allocated tasks. Allocated tasks are just a special case where the tasks are forced to be allocated onto a specific ECU while in general tasks may be allocated to one ECU out of a set of ECUs. As a result of this unification the MILP constraints are simpler. The real-time "capacity" of ECUs is characterized using the constraints published in [Eis+06]. Similar to the "Stm" local analysis module the commercial MILP solver Gurobi is used to actually solve the MILP.
The Zerg module name is`DSEEisenbrandAnalysis`

6.2.3. Parameters for the Optimization Process

6.2.3.1. Restrictions on usable CPU Cores during Optimization

The optimization backends that are used in some of the global and local analysis modules — namely Gurobi and CPlex — support the use of multiple processor cores during the optimization phase. Both are capable of automatically distributing the MILP optimization to multiple threads, each running on a different CPU core. However, the benefit of this distributed computation heavily depends on the structure of the problem to be solved. Both backends provide switches to restrict the number of available cores. For each of the benchmarks, experiments have been carried out with different settings for these switches with the purpose of achieving data on the usefulness of optimization using multiple CPU cores. The respective results are presented in the sections below separately for each of the benchmarks.

If not stated otherwise, the following defaults apply to the data presented in this chapter: For comparing different analysis methods using graphical means, only those results are selected where the number of CPU cores made available to both the global and the local analysis has been restricted to a number larger than one. E.g. for the benchmark “TindellScaled” one half of the optimization runs of the global and the local analysis modules have been restricted to either use maximal 12 cores and the other half to use maximal one core.

The global analysis module “KL” does not support multi-threading. This raises the question of how to make its results comparable to the results computed by other modules which uses more than one thread. A “fair” approach for comparing the results could be to restrict all those modules to use maximal one CPU core as well. I decided against this because of the industrial-oriented nature of this thesis: An engineer would most certainly enable the use of multiple CPU cores for all analysis modules wherever this is supported.

Note that while the global analysis modules “KL” and “HS” do not have multi-core support itself, they still can take advantage from multiple CPU cores because the used local analysis module can benefit from running on a multiple cores, too.

6.3. Metrics Suitable for Characterizing Benchmarks

6.3.1. Model Size

The size of a model in the sense of how large the design space of the resulting combinatorial solutions is results from the following properties: Number of tasks (total), number of unallocated tasks, number of signals, number of ECU-types, number of ECUs, and number of subsystems.

Some indicators may be derived from other properties, e.g. the fraction of unallocated tasks. The size of a model in the sense of how hard it is to find optimal solutions cannot be characterized as easy as in the previous definition, because this metric depends on the actual analysis methods (global and local ones) and their respective strategy for finding optimal solutions.

6.3.2. Execution Time Partial Order on ECU Type Set

Every task has a calculated or estimated WCET for one or more processor types suitable for being used in the system's hardware architecture. Imagine the case that with respect to the WCETs of a given set of tasks $\mathcal{T}$ there exist a partial-order $\leq^f$ (*faster as*) defined on the set of available processor types $\mathbb{E}$, formally

$$t_1 \ \leq^f \ t_2 :\Longleftrightarrow \forall \tau \in \mathcal{T} : \mathrm{wcet}(\tau, t_1)) \leq \mathrm{wcet}(\tau, t_2) \text{ for } t_1, t_2 \in \mathbb{E}$$

where $\mathrm{wcet}(\tau, t)$ is the worst case execution time of a task τ measured for an ECU-type t. Note that the function used for the worst case execution time is extended with a reference to a given DSE problem later on.

The existence of such a partial order for a particular problem would allow to define modifications like *Replace an existing processor type by a faster processor type.*

6.3.3. Constant Linear Speed-Up Factors

Assuming that for a given DSE problem there exists an execution time partial order on the ECU-type as defined in the previous section, an even stronger property can be formulated. In the following, DSE problems are called **WCET-proportional** if for each two different ECU-types there exists a constant speed-up factor stemming from a proportional relation between the WCETs of all software tasks in the DSE problem for those two types. Furthermore, the speed-up factors may be aligned with the costs of the ECU-types, such that for every two different ECU-types the more expensive one has the lower WCETs for all tasks contained in the model. DSE problems for which no speed-up factors can be found are called **WCET non-proportional**.

Example: In *Subset 1* of the benchmark set "TindellScaled" presented in Section 6.5.1 for every given task of the model its WCET for ECU-type `ECUType0` is exactly twice its WCET for the ECU-type `ECUType1`. Even more, the speed-up factors are also conforming to the intuition that more expensive ECU-types is faster than the cheaper one.

6.4. Hypotheses

All of the following hypotheses compare the "KL" global analysis module with the global analysis approach "LP_$*$" proposed in this thesis, separately for each of its (solver-specific) incarnations "LP_GRB", "LP_CPX" and "LP_GLP". The evaluation of the hypotheses is done by first assessing them separately for each benchmark and then aggregating the findings in the summary at the end of this chapter.

It is important to mention that the hypotheses specified in this thesis are not assessed with the goal of generalizing the findings in mind. Generalization would require a sample of benchmark models that is representative for all embedded systems for which a design space exploration as defined in this thesis is carried out. Given the availability of such a representative sample, statistical hypothesis tests such as the t-test (parametric test, applicable only if the test statistic follows the Student's t distribution) or the Wilcoxon rank-sum test (non-parametric test, no assumption about the distribution of the data) could be used to assess the hypotheses. Whenever a null hypothesis (stating that the sample data does NOT provide strong evidence supporting the assumed relationship) has to be rejected the alternative hypothesis (stating that the sample data supports the assumed relationship) is likely to be true (with a given probability of error).

In this thesis a representative sample was not available and — given the huge heterogeneous field of embedded systems development — such a sample quite probably does not exist. For this reason no statistical hypothesis tests have been used.

6.4.1. Global Analysis Modules: Comparing Hardware Cost and Runtime

Hypothesis H_A^{1a}: The majority of "LP_∗" solutions have lower or equal hardware costs than "KL" For a given DSE problem the costs of the solutions calculated with "KL" are compared directly to the costs of the solutions calculated with each of the "LP_∗" modules. The corresponding null hypothesis H_0^{1a} states that solutions calculated with "LP_∗" have higher hardware costs than solutions calculated with "KL".

Hypothesis H_A^{1b}: The majority of "LP_∗" solutions have costs similar to the "KL" solutions (depending on a given threshold) For a given DSE problem the cost of the "KL" solution is similar to the costs of each of the "LP_∗" solutions in the sense that the difference is smaller than a pre-defined threshold. This threshold can be any arbitrary value. In the following, two thresholds are evaluated: the cost of the cheapest and the cost of the most expensive ECU-type used in the benchmark DSE problem.

This hypothesis is supposed to be more helpful than the first one for the reason that the global analysis modules cannot directly control the cost values because the actual choice of ECU-types for each of the ECUs is done in the local analyses.

The corresponding null hypothesis H_0^{1b} states that the majority of "KL" solutions have lower costs for the given threshold.

Note that this hypothesis is only evaluated if useful thresholds can be found (for some of the benchmarks this is not the case).

Hypothesis H_A^2: "LP_∗" finds the majority of solutions in less time than "KL" This hypothesis is evaluated based on the runtime measured in wall clock time because this is what the user will notice. Only those cases are considered where the analysis module found a complete and feasible result within the specified time limit (if any). The corresponding null hypothesis H_0^2 states that the runtime of "KL" is equal to or less than the runtime of "LP_∗".

6.4.2. Local Analysis Modules: Comparing Hardware Cost and Runtime

Hypothesis H_A^3: The majority of solutions found by "Stm" have lower or equal hardware costs than those found by "Eis" The corresponding null hypothesis H_0^3 states that the costs of the solutions found by "Eis" are less than the costs of the solutions found by "Stm" in most of the cases.

Hypothesis H_A^4: "Stm" finds the majority of solutions in less time than "Eis" Again, this hypothesis is evaluated based on the runtime measured in wall clock time. Only those cases are considered where both analysis modules found complete and feasible solutions within the specified time limit (if any). The corresponding null hypothesis H_0^4 states that the runtime of "Eis" is equal to or less than the runtime of "Stm" in most of the cases.

6.4.3. Single vs. Multi-Core

Hypothesis H_A^5: The use of multiple CPU cores reduces the runtimes of multi-core-enabled global analysis modules This hypothesis is also evaluated based on wall clock time. The corresponding null hypothesis H_0^5 states that the runtimes of multi-core enabled global analysis modules are not smaller if multiple threads are allowed.

6.5. Benchmarks

6.5.1. Benchmark "TindellScaled"

The first benchmark named "TindellScaled" is based on an example in [TBW92]. The original model contains 43 tasks with eight ECUs. Based on that original model the set of benchmark models has been designed. Figure D.2 (in the appendix) shows the task network of the smallest models contained in this benchmark.

The hardware architectural pattern has been extended to contain three subsystems, two subsystems with three ECUs and one with two ECUs (see Figure D.1 in the appendix). One ECU in each subsystem is defined as the gateway between the global time-slot based bus (FlexRay) and the subsystem-local bus. Each ECU in a subsystem is connected to the subsystem-local bus. Four ECU-types with different cost and memory capacity are defined (see Table D.1 in the appendix). Each ECU can be typed by each of the ECU-types. Two bus types are defined, one for the global bus (time-slot based) and one priority-based (CAN-type) which is used by each of the local buses. There are 25 bus slots on the global bus.

The task network is still very similar to the original task network, with the following changes: Some signals contained in the original task network have been deleted to make the models compliant to the current limitation of both local analysis modules that each task may receive maximal one signal. Worst case execution times and memory consumption have been defined for each of the tasks for each of the ECU-types defined in the architectural pattern. There are two versions of the task network which are identically except for the chosen worst case execution times and memory consumptions.

All models of **Set 1** are "WCET proportional", e.g. there exists a set of speed-up factors defining the relation between all used ECU-types with respect to their computational power. The WCEts of all tasks are defined such that they conform to those speed-up factors. All models of **Set 2** are "WCET non-proportional", in the sense that for them the WCETs have been redefined such that linear speed-up factors for pairs of ECU-types effectively do not exist.

The original allocation of tasks to ECUs has mainly been preserved. In the original model some unallocated tasks may only be allocated to specific sets of ECUs. The global and local analysis modules implementing the approaches presented in this thesis are powerful enough to handle such constraints, but some of the other analysis modules cannot handle them. Therefore, those constraints have not been considered for this benchmark.

During preparation of the two model sets the approach proposed in this thesis has already been applied to find some complete and schedulable allocations for smallest models contained in Set 1 and Set 2 respectively. With those feasible solutions at hand, for both models three separate variants have been generated simply by unallocating some of the tasks again, resulting in one model with 14% unallocated asks, one with 20%, and one with 37% in each of the two sets. The tasks to be unallocated have been chosen carefully such that a reduction of the communication load (number of required bus slots) on the global bus can be achieved additionally.

At this point six hand-crafted benchmark models were available, three in each set. In the next step, each of those models has been scaled by a constant factor between 2 and 18. Scaling is done by duplicating every element of the hardware architectural pattern and the task network with the exception of the hardware types (ECU-types and bus types) and the global bus. Duplicating in this context means to create a copy of the entity (e.g. a task) which has a new object identity (e.g. task name) but is identically to the original object in all other aspects. Duplicating the hardware architectural pattern is done on the level of subsystem: While duplicating a given subsystem all its ECUs are also duplicated and associated to the subsystem copy. The global bus requires special care, as it is the only shared element in the whole hardware architecture. In the copy of each subsystem the ECU that is the copy of the original gateway ECU is connected to both the global and the local bus and used as gateway for that new subsystem.

As stated in Chapter 4, the minimal number of required bus slots for the transmission of a given signal on the global bus is defined as: $\left\lceil \frac{d}{x} \right\rceil$ where d denotes the signal's deadline and x denotes the length of the TDMA round on the global bus. By duplicating each bus slot on the global bus the length of the TDMA increases and so does the number of required bus slots for each signal on the global bus. That dependency can be solved (if a solution exists) by using a fixed-point iteration for calculating the required number of bus slots: First, as many copies of each bus slots as required by the scaling factor are created. Then the first iteration is started by recalculating the number of bus slots for each signal on the global bus. The new sum of required bus slots is then used for recalculating the number of required slots but only if the number of required bus slots has increased since the last iteration. It may happen during this process that no fixed-point can be found. In this case the model cannot be scaled by the given factor (and not by any larger factor, too). The fixed-point calculation used for scaling gives up after a certain number of iterations. According to the way the scaled models are constructed, there are three dimensions of variety:

1. The distinction between models with ECU-type speed-up factors and those without. *Set 1* contains all models with speed-up factors. *Set 2* contains all other models.

2. The total number of tasks, varying from 43 to 774 (18 times larger than the original model).
3. The percentage of unallocated tasks, one of 14%, 20%, or 37%

Not all possible combinations along the above mentioned dimensions of variety are part of the benchmark. The combination of a large scaling factor and a small number of unallocated tasks leads to situations where no fixed point for number of bus slots on the global bus can be found. In such cases the messages initially allocated to the global bus already need more capacity than available.

All analysis modules under evaluation have been run with a time limit of 60 minutes. As the analysis module "MZ" never succeeded for larger benchmarks (with more than 86 tasks), it has been disabled purposely for all large models.

The overall analysis approach ("MZ") and all the global analysis methods except for the SMT approach based on HySAT ("HS") have been evaluated based on the benchmark "TindellScaled". The "HS" global analysis module has not been evaluated at all because it became clear during the first experiments that this module finished successfully for only a few very small models (with up to 86 tasks, up to 14% unallocated tasks). For all larger models "HS" timed out without finding any useful result.

6.5.1.1. Results: Global Analysis

This benchmark "TindellScaled" consists of 68 models. Each analysis module has been evaluated twice on each of the input models: The first time the number of processor cores available to the analysis backend has been restricted to 12, the second time that limit was lowered to 1. Note that currently only the "MZ" approach and the "LP_∗" formulation presented in this paper are capable of using parallel optimization on multiple processor cores, the latter one only if used in conjunction with either Gurobi or CPlex, but not GLPK. However, while the "HS" and "KL" approaches are not meant to be parallelized on multiple cores, they can still benefit if used in combination with either the *Spare-Time/MaxWCET* local analysis approach presented in this thesis or the *Eisenbrand* approach as long as Gurobi is used during the (local) optimization phase. In the following figures of this section the results are presented grouped by the sets (either Set 1 or Set 2) and the number of unallocated tasks. Only those results are shown in the figures where 12 cores have been available.

A table with all global results for this benchmark can be found in the appendix (see Section D.1.1). In the following, the result data is presented in aggregated form suitable for studying the hypothesis formulated earlier in this chapter.

Table 6.3 shows general information about the results for this benchmark. The first column contains the names of all global analysis modules under evaluation and the overall analysis "MZ". The next columns contain the number of analysis runs for each of the analysis modules which have been successful, encountered a time-out, returned incomplete results (e.g. one or more tasks remained unallocated) or experienced some kind of error during their execution. An analysis run is classified as **Successful** if the analysis module returned a valid result within the specified time limit. The validity of each results is verified by running a schedulability analysis and a consistency check which for example checks for unallocated tasks, missing messages, etc. An analysis run is classified as **Timeout** if the analysis module encountered a time-out condition. An analysis run is classified as **Incomplete** if the analysis module finished within in the specified time limit but the solution is not complete, for example tasks are still unallocated or messages are missing. An analysis run is classified as **Error** if the analysis module encountered an error condition and notified the evaluator module about this, or if the analysis module threw an exception.

For the "KL" analysis module the evaluator module was unable to distinguish between results where an incomplete solution was returned due to a programming bug and results where the underlying algorithm gave up because it was unable to find a valid solution. Therefore all results where at least one task remained unallocated calculated by this analysis module have been categorized as *Incomplete*. The last column shows the total number of analysis runs per module. As mentioned before, the overall analysis has been used only on small models which explains the small number of total analysis runs for that module.

Global Module	Successful	Timeout	Incomplete	Error	Total
KL	58	41	37	0	**136**
LP_GRB	52	84	0	0	**136**
LP_CPX	38	88	10	0	**136**
LP_GLP	22	76	0	0	**98**
MZ	10	0	6	0	**16**
Total	**180**	**289**	**53**	**0**	**522**

Table 6.3. – Benchmark "TindellScaled": General Information

As most important metrics the costs of the resulting solutions and the runtime (wall clock) for all analysis runs where the number of available CPU cores has been limited to 12 is depicted in Figure 6.2 for Set 1 (benchmarks with speed-up factors) and Figure 6.3 for Set 2 (benchmarks with no speed-up factors). Both figures consists of two rows of charts. The upper row contains charts visualizing the runtime (wall clock) starting with 14% percent unallocated tasks, to 20%, and finally to 37%. In the second row the corresponding result costs are depicted respectively. The x and y axes have been aligned to allow a quick comparison of the results.

The results show that for the given benchmark the "KL" analysis module scales better than "LP_∗" with increasing number of tasks with respect to the runtime. This is the expected when comparing a (good) heuristic method and an optimal method such as MILP. Rather unexpected however was the fact, that "KL" performs slightly better in many cases with respect to the achieved cost, too.

One reason is that this benchmark is beneficial for "KL": In all cases a solution can be found which is very close to the initial hardware cost of the DSE problem. The initial cost is a lower bound for the resulting cost because according to the definition of a DSE Problem the initial cost have to be minimal. Allocating all unallocated software tasks causes more load on the involved ECUs which can lead to additional cost but never to reduced cost. The initial cost have been visualized by adding a dashed line to the cost charts in Figure 6.2 and Figure 6.3. “KL” is implemented such that it almost always underestimates the required cost forcing the local analysis module to choose very tight solutions. This works especially well for DSE problems where solutions exist which require only minimal additional hardware cost. Even if the capacity estimation of the “LP_$*$” approach is more accurate than the one used in “KL”, this would not lead to better results than those calculated by “KL”, because the results of “KL” are optimal in most cases. This has been shown for those benchmark models where the “MZ” approach successfully calculated a result. The costs of solutions calculated by the “MZ” global analysis module are a lower bound[1] for the achievable cost due to the concept of the “MZ” approach. By looking at those results in Figures 6.2 and 6.3 where the “MZ” global module was able to find a solution, it can be seen, that the “KL” costs are identically to the “MZ” costs. In those cases an optimal solution has been found by “KL” (and there exist no better solution). It can be concluded that the models contained in the benchmark “TindellScaled” are beneficial for “KL” to the disadvantage of the “LP_$*$” global analysis modules.

[1]“MZ” calculates optimal solutions but in doing so ignores all communication on local buses. Therefore the resulting cost are possibly less than the minimal cost achievable when additionally considering local communication.

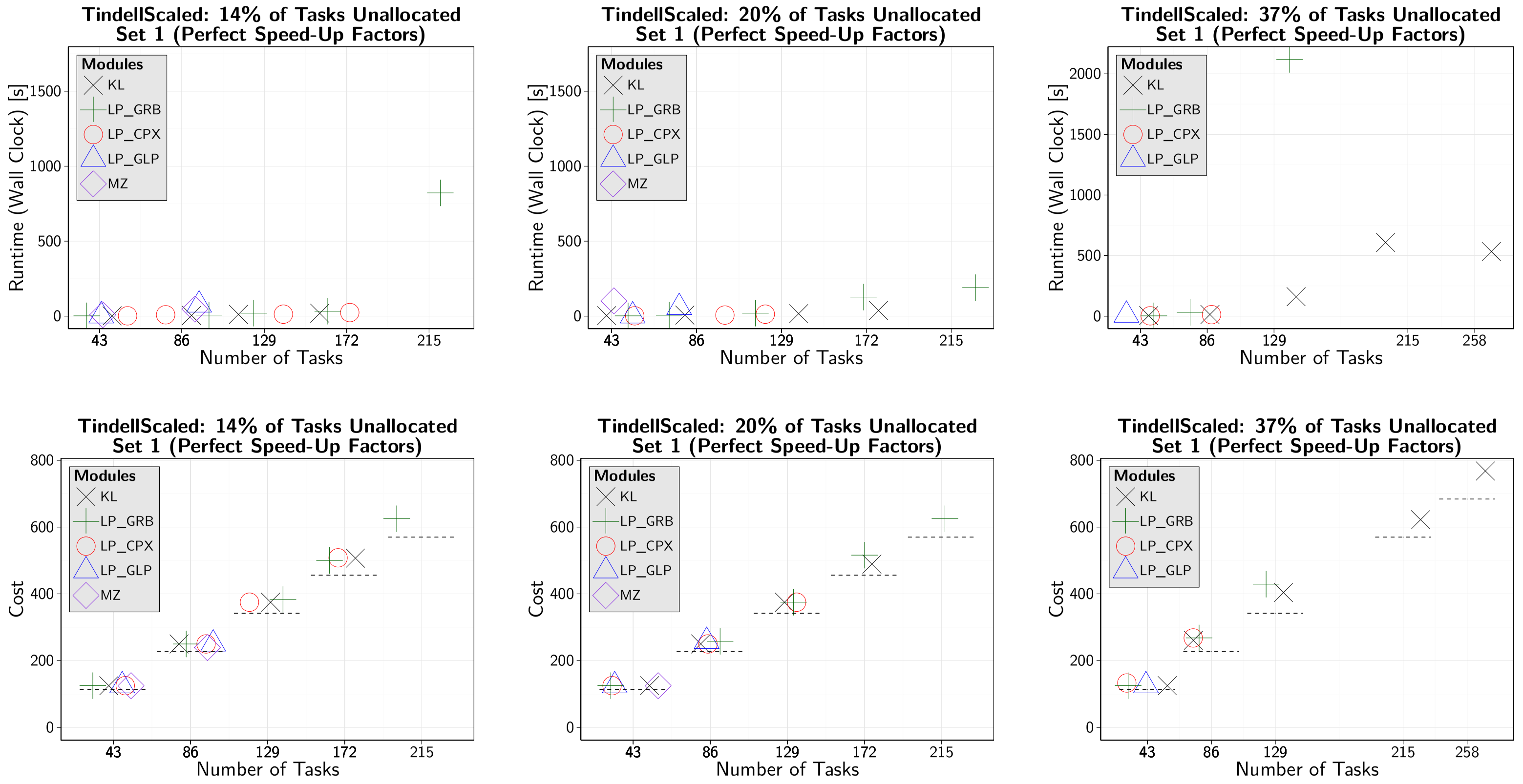


Figure 6.2. – Benchmark “TindellScaled”: Global Analysis, Set 1. Note that for better readability some points have been slightly shifted along the x-axis (in the data set their x-value is exactly on the coordinates marked on the x-axis)

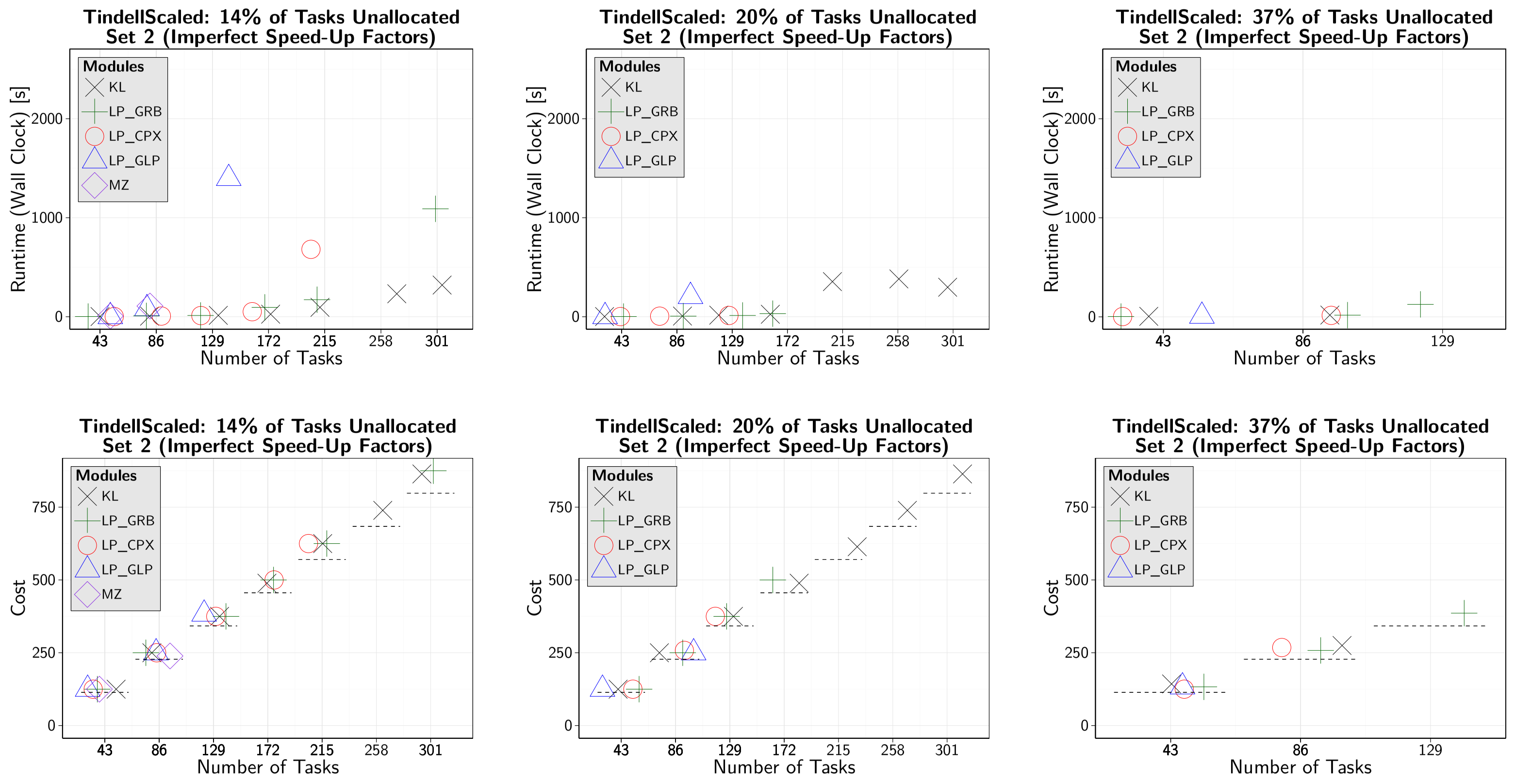


Figure 6.3. – Benchmark “TindellScaled”: Global Analysis, Set 2. Note that for better readability some points have been slightly shifted along the x-axis (in the data set their x-value is exactly on the coordinates marked on the x-axis)

Method	**Comparing Cost**			$\delta = \lvert cost_{KL} - cost \rvert$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 11$	$11 < \delta \leq 37$	$37 < \delta$
LP_GRB	6	22	18	36	10	0
LP_CPX	4	22	10	34	2	0
LP_GLP	2	16	4	22	0	0

Table 6.4. – Hypotheses 1a and 1b: Cost of Successful Runs (“TindellScaled”)

Carefully examining the data revealed that the “LP_∗” methods found in almost all cases solutions very similar to those found by “KL”. Table 6.4 separately compares “LP_GRB”, “LP_CPX” and “LP_GLP” with “KL”. Only those results are counted where both methods to be compared have successfully produced a valid (complete and feasible) solution. Adding the first three columns in a row gives the number of comparable results for the compared global analysis methods. Obviously the “LP_GRB” analysis module with its backend Gurobi calculated far more results comparable to “KL” than the other modules.

It can be seen that in many cases “KL” produces results with slightly lower costs compared to the results produced by each of the different “LP_∗” backends (which implement the approach proposed in this thesis). However, in most cases the absolute difference between the hardware costs is lower than the cost of the cheapest ECU-type (11). In almost all cases it is lower than the cost of the most expensive ECU-type (37). The cost charts presented as part of Figure 6.2 and Figure 6.3 make this result visible: In most cases the difference of the y-position between points indicating the resulting costs calculated by different analysis modules for a given model can barely be spotted at all.

Following from these findings, a decision about Hypothesis 1 with respect to the benchmark “TindellScaled” can be taken: The null hypothesis H_0^{1a} can be rejected for all “LP_∗” analysis modules but only because in most of the cases the resulting costs are equal. It can be stated that the “LP_∗” approach in most cases produces results with equal or lower costs (though lower costs are rarely achieved). Note however, that the hypothesis stating that solutions created by “KL” in most of the cases have lower or equal costs would be accepted, too, by rejecting the corresponding null hypothesis, with a higher significance due the large number of cases where “KL” has produced cases with lower costs.

The null hypothesis H_0^{1b} can also be rejected for both threshold values evaluated (see table 6.4). It can be concluded that for most of the cases the difference between the costs of solutions created by “KL” are very close to those created by “LP_∗”.

Table 6.5 compares the runtimes as measured for all “LP_∗” global modules of all successful runs with those for the “KL” module. The smallest number of comparable solutions has been found for “LP_GLP” (using the open-source backend GLPK). For those results “KL” clearly is the winner: There are only few cases where “LP_GLP” had a lower runtime than “KL”. As expected both commercial backends used in “LP_GRB” and “LP_CPX” produce better results. At first glance it might look like “LP_CPX” has the best results. But the total number of models which have been successfully analyzed

Method	Comparing Runtime			$\delta = \lvert runtime_{KL} - runtime \rvert$		
	Less than KL	Equal	Greater than KL	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
LP_GRB	17	6	23	29	7	10
LP_CPX	19	7	10	31	2	3
LP_GLP	2	6	14	12	3	7

Table 6.5. – Hypothesis 2: Comparing the Runtime (Wall Clock) of Successful Runs ("TindellScaled")

with that module is significantly lower than for "LP_GRB". This means that there are many models for which "LP_CPX" could not find any result inside of the specified time limit of 60 minutes.

Based on those observations it is not possible to reject the null hypothesis H_0^2 which means that it cannot be shown based on the presented data that "LP_$*$" finds the majority of solutions in less time than "KL".

6.5.1.2. Results: Local Analysis

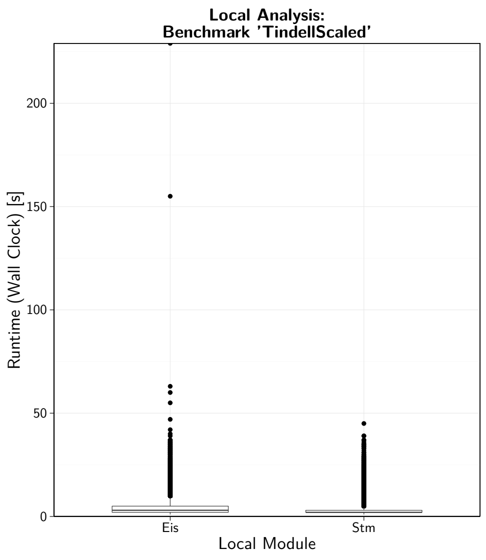
Table 6.6 shows an overview of the local analysis results measured while analyzing the benchmark "TindellScaled". All global analysis modules except "MZ" delegate the detailed subsystem analysis to one of the local analysis modules. Therefore, a huge amount of data has been produced while evaluating the benchmark. Each local measurement results from one run of a local analysis module. Note that during the optimization phase it is likely that two or more local analysis modules have been ran with exactly the same parameters on the same local DSE problem. These "redundant" (in the sense that the inputs for the local optimization have been identical) results could not be avoided without monitoring all model details which would have caused a large bias with respect to the runtime. Therefore, this special kind of redundancy has been accepted as unavoidable effect.

Module	Successful	Error	Total
Eis	47867	17	**47884**
Stm	47882	2	**47884**
Total	**95749**	**19**	**95768**

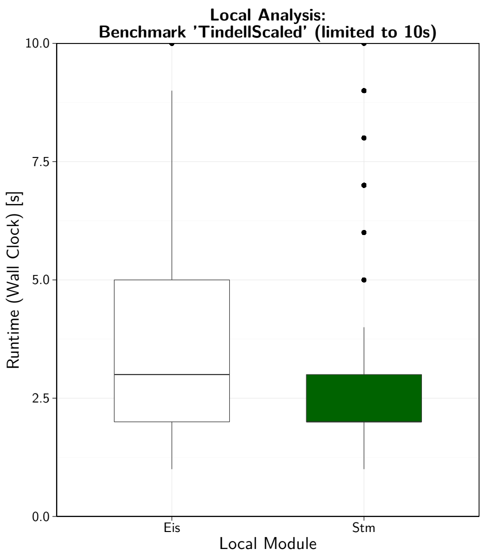
Table 6.6. – Benchmark "TindellScaled": General Information about Local Analysis Results

The sheer number of records requires a different approach for visualization. Figure 6.4 shows two figures each with two boxplots comparing the runtimes (wall clock) of all local analysis runs carried out for analyzing the benchmark "TindellScaled". Figure 6.4a shows all results, and in Figure 6.4b the right one truncates the y-axis at 10s thus hiding most of the outliers (dots above the boxes).

In these boxplots (and all the following ones), the lower "hinge" or "end" of the box corresponds to the first quartile (25th percentile, 25% of the data records have lower



(a) All results



(b) Scale truncated at 10 seconds

Figure 6.4. – Benchmark "TindellScaled": Overview of Local Analysis Results

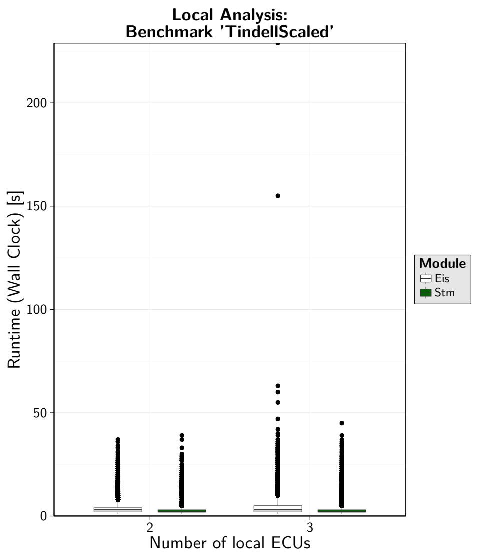
values) of the analyzed data (with respect to the chosen y-axis) and the upper "hinge" to the third quartile (75th percentiles, 25% of the data records have greater values), respectively. The horizontal line inside of the boxes marks the median. The distance between the lower and upper hinge (measured by the chosen variable on the y-axis) is called the interquartile range (IRQ). The lower whisker (vertical line) starts at the lower hinge of the box and extends down to the lowest measured value within $1.5 \star \text{IRQ}$ of that hinge. The upper whisker starts at the upper hinge of the box and extends up to the largest measured value within $1.5 \star \text{IRQ}$ of that hinge. Every measured result below or above the respective whisker can be considered an outlier.

Module	Runtime (Wall Clock) [s]			Cost		
	Mean	**Median**	**Std. Dev.**	**Mean**	**Median**	**Std. Dev.**
Eis	3.85	3.00	3.27	35.65	38.00	4.86
Stm	2.86	2.00	2.50	35.65	38.00	4.88

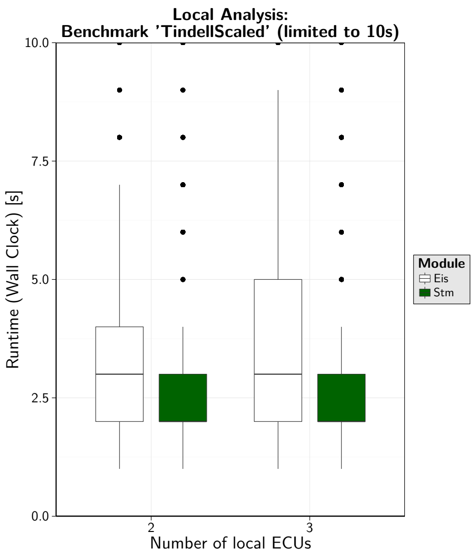
Table 6.7. – Benchmark "TindellScaled": Statistical Summary of Local Analysis Results

Table 6.7 shows median, mean and standard deviation of the runtime (wall clock) and costs separately for both local analysis modules "Stm" and "Eis". Of course, without further classifying the input models the explanatory power of the table and the boxplots is limited. But it can be seen that the proposed "Stm" approach significantly outperforms the referenced "Eis" approach and additionally is more stable considering the outliers measured for the runtime (wall clock). It has been expected that any improvements of the analysis runtime of the "Stm" approach compared to the "Eis" approach would come at the expense of a worse quality of the results produced by "Stm", namely slightly higher hardware costs. However, Table 6.7 shows that the mean of the hardware costs of all the local results computed by the "Stm" approach is identical (!) to the mean of the hardware costs found by the "Eis" approach on the same set of models. It can be concluded that for the benchmark "TindellScaled" the proposed "Stm" local analysis approach is on average about one third faster than the "Eis" approach while producing results of identical quality. This conclusion is especially astonishing given the fact that the "Stm" approach is affected by a loss of precision caused by applying the abstractions required for the Spare-time/MaxWCET formalism.

The presentation of the local analysis results for this benchmark is completed by showing the runtime (wall clock) of the respective local analysis modules plotted against the number of tasks, the number of unallocated tasks and the number of available ECUs.



(a) All Results



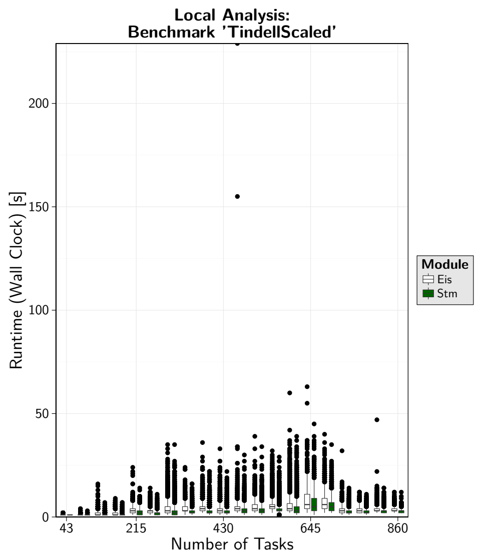
(b) Scale truncated at 10 seconds

Figure 6.5. – Benchmark "TindellScaled": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of local ECUs

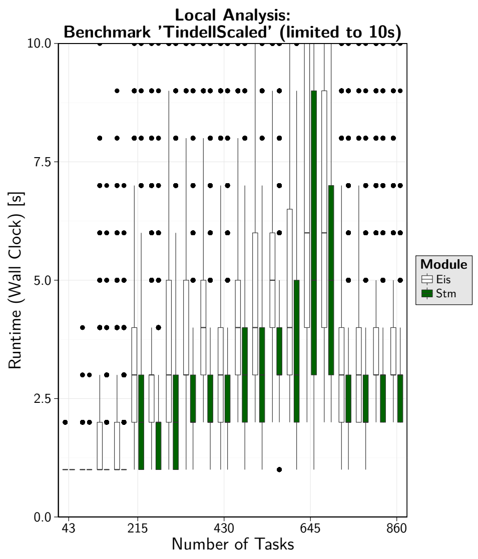
Figure 6.5 shows that the number of subsystem-local ECUs varies only between two and three. This is caused by the method used for constructing the benchmark's models: The subsystems contained in those small models used as templates for the upscaling process have simply been copied leaving their internal structure unchanged.

The benchmark "Generated" (described in Section 6.5.2) contains models with larger subsystems and is therefore more useful for evaluating the influence of the number of subsystem-local ECUs on the runtime of the optimization phase.

In Figure 6.6 the runtime is plotted against the total number of tasks handled by the respective local analysis module. Figure 6.7 the runtime is plotted against the number of

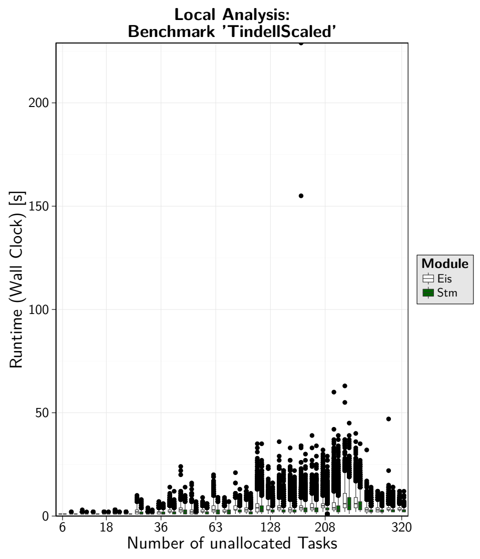


(a) All Results

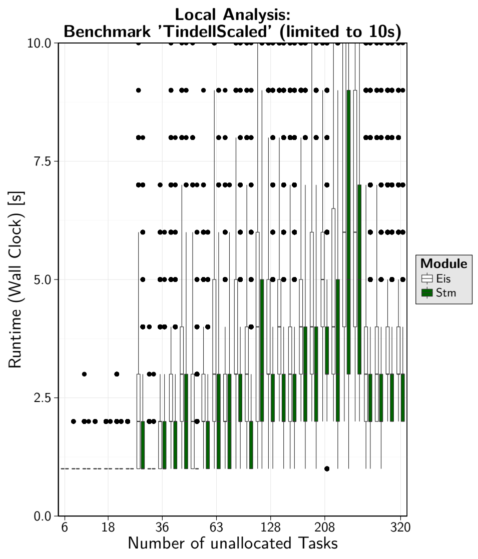


(b) Scale truncated at 10 seconds

Figure 6.6. – Benchmark "TindellScaled": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of Tasks



(a) All Results



(b) Scale truncated at 10 seconds

Figure 6.7. – Benchmark "TindellScaled": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of unallocated Tasks

unallocated tasks. It can be seen that — as expected — the performance of both local analysis modules is excellent for very small models. But the figures also show that there is no strong correlation between the runtime and the total number of tasks on the one hand, and the runtime and the number of unallocated tasks on the other hand.

Correlation (Stm)	**Runtime (Wall Clock)**		**Runtime (CPU)**	
	Linear	**Logarithmic**	**Linear**	**Logarithmic**
Number of Tasks	0.18	0.35	0.23	0.43
Unallocated Tasks	0.20	0.38	0.25	0.44
Tasks * Unallocated	0.13	0.28	0.20	0.38
Unallocated Tasks * ECUs	0.20	0.38	0.25	0.44

Table 6.8. – Benchmark “TindellScaled”: Analysis of Correlations: Local Module “Stm”

Correlation (Eis)	**Runtime (Wall Clock)**		**Runtime (CPU)**	
	Linear	**Logarithmic**	**Linear**	**Logarithmic**
Number of Tasks	0.13	0.28	0.20	0.38
Unallocated Tasks	0.15	0.30	0.21	0.39
Tasks * Unallocated	0.07	0.19	0.16	0.31
Unallocated Tasks * ECUs	0.15	0.30	0.22	0.39

Table 6.9. – Benchmark “TindellScaled”: Analysis of Correlations: Local Module “Eis”

Tables 6.8 and 6.9 show the calculated correlations [2] for the local analysis module “Stm” and “Eis”, respectively. Three variables have been considered: The total number of tasks (for the respective subsystem), the number of unallocated tasks, and an additional variable holding the result of the multiplication of the aforementioned variables. The correlation of these variables with the wall clock runtime and the CPU runtime (which could be much larger than the wall clock time whenever multiple cores are used extensively) has been calculated.

The concept of correlation between two variables in a data set relies on the assumption that both variables are increasing linearly. However, it is more likely that the runtime (both wall clock and CPU) increases exponentially. Therefore the correlation is not only tested for the directly measured runtimes (called “Linear” in the table headers), but also for additional variables derived by calculating the natural logarithm of the measured values (called “Logarithmic” in the table headers). A value of 0 would indicate no correlation at all, a value of 1 would indicate a perfect correlation, and a value of -1 would indicate a perfect negative correlation (one variable would always decrease linearly while the other increases and vice versa).

The tables illustrate that in general the correlations are rather small. For the “Stm” analysis approach the correlations are a little bit stronger than for the “Eis” approach. But considering the low overall correlations it can be concluded that the analyzed model-specific independent variables only have a low impact on the runtime.

[2] All correlation tests in this thesis use the *Pearson product-moment correlation coefficient.*

The complexity of both local analysis approaches and the huge influence of the used MILP backend renders the chance of finding strong correlations between single variables quite improbable.

Method	**Comparing Cost**			$\delta = \vert cost_{Eis} - cost\vert$		
	Less than Eis	**Equal**	**Greater than Eis**	$\delta \leq 11$	$11 < \delta \leq 37$	$37 < \delta$
Stm	0	47859	6	47860	5	0

Table 6.10. – Hypothesis 3: Comparing Local Cost of Successful Runs (“TindellScaled”)

Table 6.10 compares the costs of the results computed using the “Stm” approach with those of the “Eis” approach. The table shows that in the benchmark “TindellScaled” the “Eis” approach had a better result (one with lower cost) for only six results out of 47859. For five of those six solutions the difference between the costs was smaller than the cost for the cheapest ECU-type (11). In all cases that difference was smaller than the cost of the most expensive ECU-type. Based on these results the null hypothesis H_0^3 can be rejected in favor to the alternative hypothesis H_A^3 (“The majority of solutions found using “Stm” have lower or equal hardware costs than those found by “Eis””).

Method	**Comparing Runtime**			$\delta = \vert runtime_{Eis} - runtime\vert$		
	Less than Eis	**Equal**	**Greater than Eis**	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
Stm	28406	15718	3741	47309	554	2

Table 6.11. – Hypothesis 4: Comparing Local Runtime of Successful Runs (“TindellScaled”)

In the same manner the runtimes of the local analysis modules are compared in Table 6.11. According to the presented data the runtime of the “Eis” module was identically to the corresponding runtime of the “Stm” module (15718 cases), in some cases it was even lower (3741 cases). But in the majority of cases the runtimes of “Stm” were lower (28406 cases). Due to the small size of most of the local DSE problems the difference in the runtimes between “Stm” and “Eis” was less or equal to 10 seconds. But regardless of that, based on the data, the null hypothesis H_0^4 can be rejected in favor of the H_A^4 (“Stm” finds the majority of solutions in less time than “Eis”).

6.5.1.3. Single Core vs. Multi Core

As a side effect of evaluating this benchmark (and the following benchmarks) a huge amount of data has been collected about the usefulness of multiple CPU cores for those global and local analyses with multi-core support. All DSE problems of the benchmark “TindellScaled” have been analyzed twice, the first time with a restriction to maximal 12 CPU cores and the second time with a restriction to exactly one CPU core. Note that some global analysis modules (“HS”, “MZ” and “LP_GLP”) have not been executed for those DSE problems whenever it was obvious that they would not succeed (because they already failed for all smaller instances starting from a certain model size).

The results of the most powerful global analysis modules are shown in the following figures, namely “KL”, “LP_GRB” and “LP_CPX”. While “KL” itself is not able to use more than one CPU core, its local analysis backend (“Stm”) was able to use multiple cores (because of using Gurobi for solving the local DSE problems).

In Figure 6.8 the results for “KL” are depicted. The upper (black) part of the bars (sometimes barely visible for “KL”) represents the duration spent computing the global analysis. The lower (green) part of the bars show the total amount of time spent in the local analysis module. The left part of each of the figures shows the results where only one CPU core has been allowed. The right side shows those results where maximal 12 CPU cores were available to the global and local analysis modules. It can be seen that “KL” does not take advantage of multiple cores in most cases: For some DSE problems the time used for the local analyses was a little bit shorter, but sometimes it is even larger. The runtimes of “KL” itself are already very low compared to the fraction of the local analysis. According to the data for this benchmark it currently does not make sense to consider a parallelized implementation of “KL”.

But why does the local analysis not take advantage of using multiple CPU cores? The reason for this effect might be the small size of the subsystem-local DSE problems. For small local DSE problems the overhead of creating the MILP data structures is very large compared to the actual MILP optimization process. The huge number of individual local analysis steps (one for each subsystem in each global analysis iteration) multiplied by the short runtimes of the local analysis module results in the huge fraction of the computation time required for the local analysis part. Parallelization using multiple CPU cores inside of each local analysis runs would most likely not reduce the already small runtimes significantly. But the data measured for this thesis is not sufficient to evaluate the influence of using parallelization in the local analysis separately from the global analysis.

Figure 6.9 shows the results of the “LP_GRB” global analysis module (which uses Gurobi as its backend) and Figure 6.10 shows the results of the “LP_CPX” analysis (which is based on CPlex). It can be seen that the computation time for the global analysis based on MILP grows exponentially with the size of the DSE problem (here the number of tasks has been chosen as metric for the problem size).

Gurobi takes advantage of multiple CPU cores in most of the cases. But the achieved computation speed-up is significantly lower than the speed-up factor one might have expected when making available 12 cores instead of only 1 core. Instead, the speed-up is approximately 2 which is still a very good result considering the fact that the necessary parallelization is provided automatically by Gurobi without requiring any special measures while preparing the MILP problem formulation.

For CPlex, in some cases the use of multiple threads even led to a longer computation time.

It can be concluded that for this benchmark the use of multiple CPU threads during the optimization phase is beneficial for the global analysis modules “LP_GRB” and “LP_CPX” which directly support parallelization through their respective backends. Despite of the mixed picture, the null hypothesis H_0^5 can be rejected in favor of the

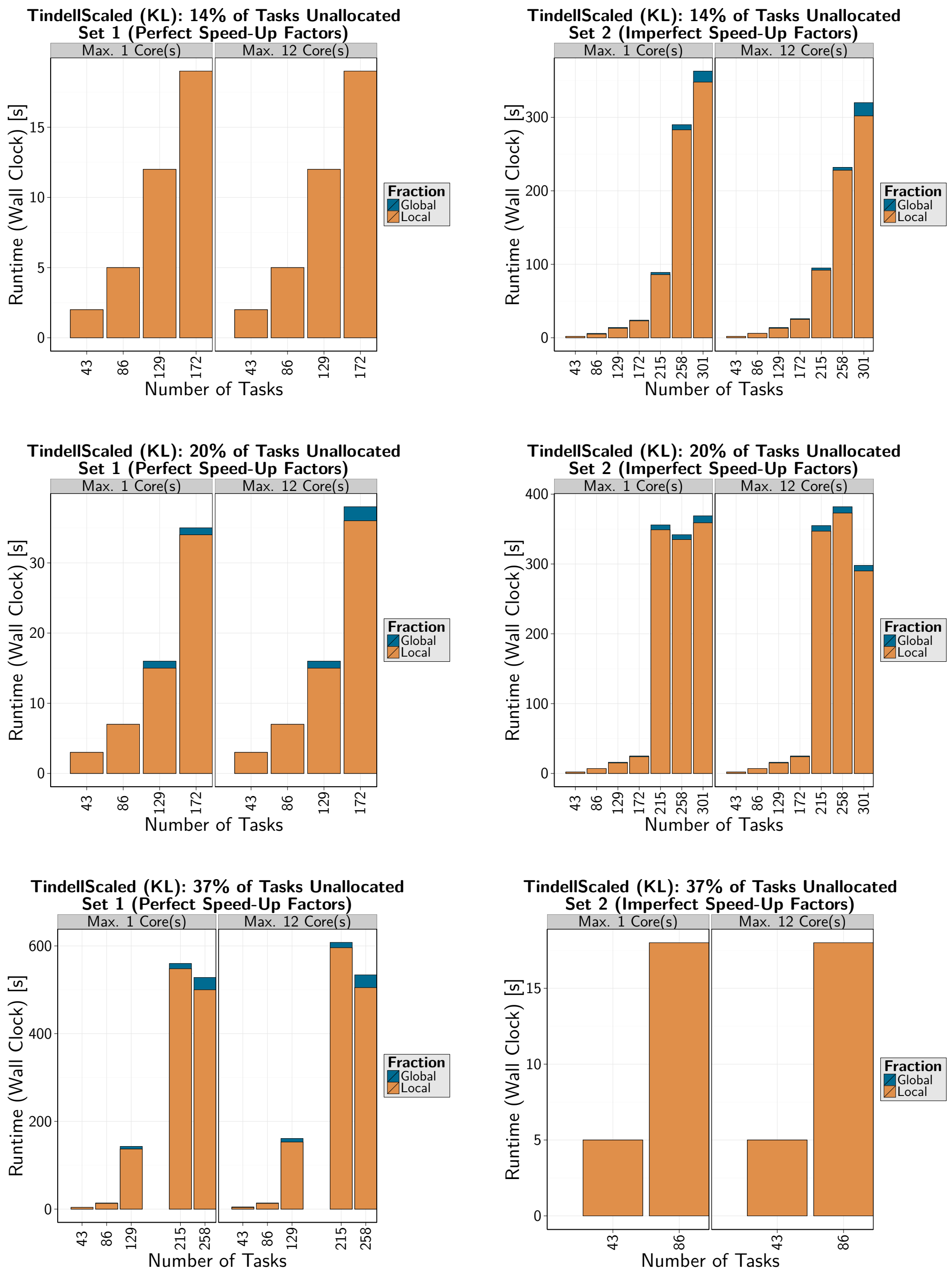


Figure 6.8. – Benchmark “TindellScaled”: Multi-core vs. Single-Core (“KL”)

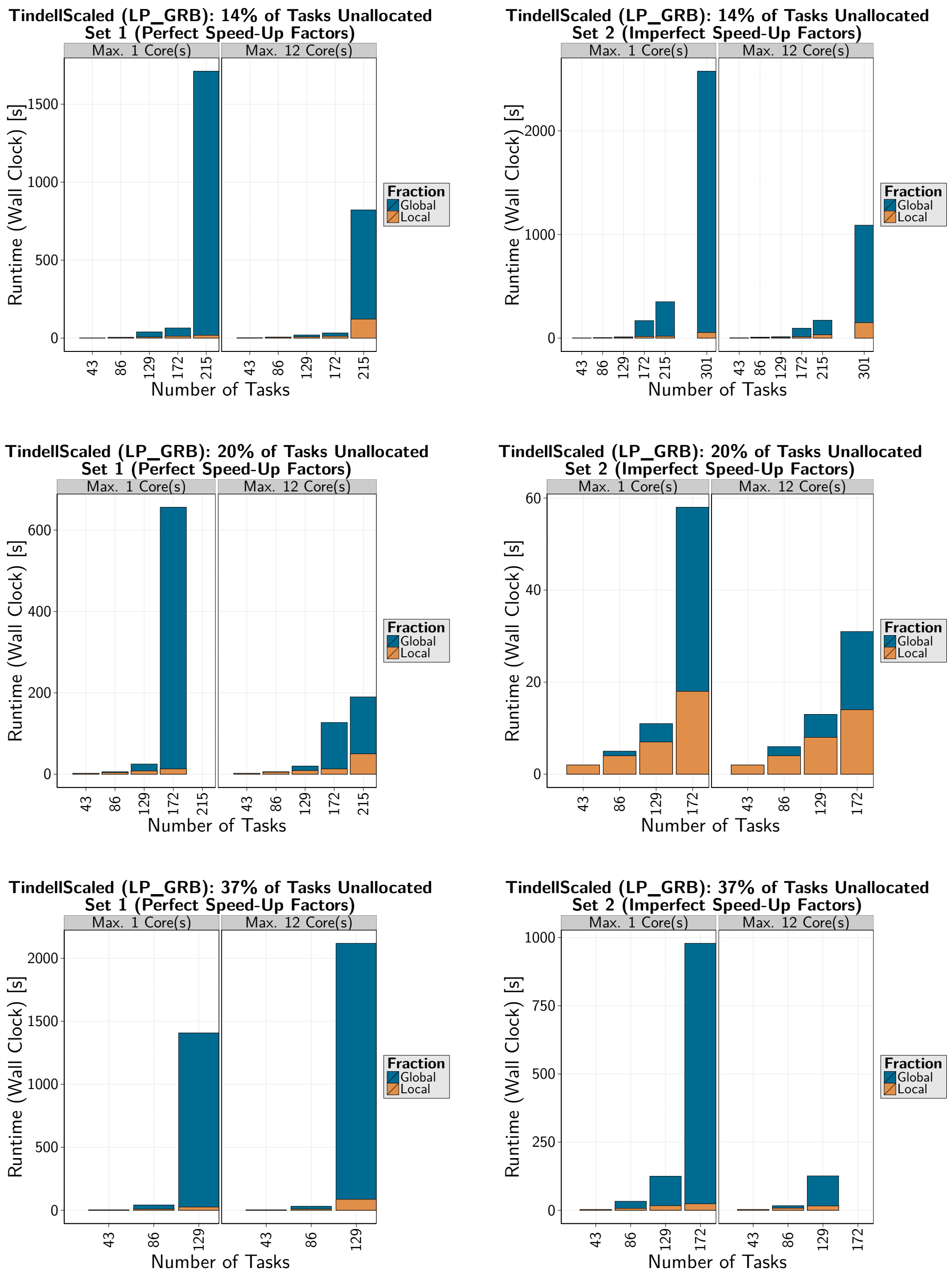


Figure 6.9. – Benchmark "TindellScaled": Multi-core vs. Single-Core ("LP_GRB")

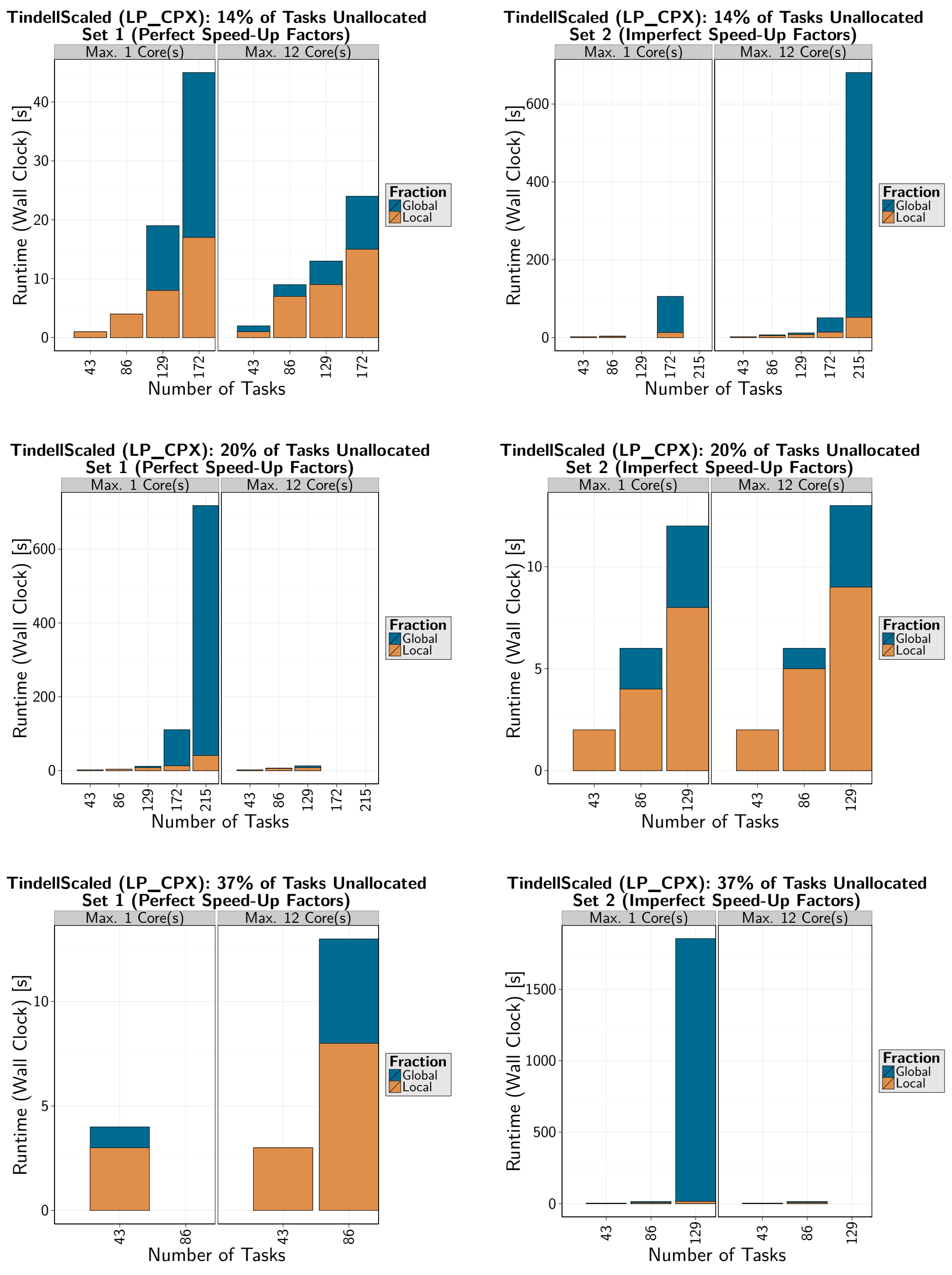


Figure 6.10. – Benchmark “TindellScaled”: Multi-core vs. Single-Core (“LP_CPX”)

alternative hypothesis H_A^5 ("The use of multiple CPU cores reduces the runtimes of multi-core enabled global analysis modules") due to the significant speed-up of the multi-core enabled global analysis modules. For the other global analysis modules the null hypothesis can either not be rejected ("LP_GLP" or has not been analyzed due to the very small number of results produced by the respective analysis module "MZ", "HS", and "MZ").

6.5.2. Benchmark "Generated"

The benchmark "Generated" is a set of completely artificial DSE problems which have been generated randomly by a special synthesis module integrated into Zerg. The synthesis module supports a huge amount of parameters such as "minimal number of tasks", "maximal number of tasks", "minimal number of ECU types", "maximal number of ECU types". During the synthesis process the actual number of tasks, number of ECU types, etc. are determined by randomly choosing a value inside of the specified intervals. Then the corresponding artifacts are created, e.g. tasks, signals, subsystems, ECUs, ECU-types, buses and bus types. Other parameters describe the intervals for choosing the artifact's properties such as task period and WCETs. A task network and a hardware architecture are constructed which comply with the limitations for task networks and hardware architectures described in this thesis. All tasks are allocated to the ECUs of the system. Messages are created where necessary. Of course, the algorithm for finding a valid allocation of the tasks has to consider the communication between them to avoid situations where too high communication loads are generated on the global bus or the local buses. As the optimization approach requires the initial system to be feasible, the synthesis has to take care to fulfill all requirements regarding schedulability, memory limitations, communication restrictions, and so on. After a complete feasible system (with all tasks allocated) has been generated successfully, some of the tasks are removed from their respective ECU and the corresponding messages are removed where possible/necessary. The result of a successful synthesis process is an artificial DSE problem where some tasks are initially allocated and other tasks are not yet allocated for which a solution exists.

During the development of the synthesis module it became more and more clear that two highly contradictory requirements have to be met: On the one hand, the constructed DSE problems have to be feasible, but on the other hand, they should pose a real challenge to the optimization approaches under evaluation. Generating feasible DSE problems taken alone already is a complicated task due to all the involved constraints. Constructing challenging DSE problems requires even more sophisticated algorithms because DSE problems are challenging only if they are close to being infeasible. These situation led to a very complex implementation of the synthesis module (3108 lines of code). It has been decided to prioritize the goal of constructing feasible DSE problems. Therefore, all models contained in the benchmark "Generated" are huge, but not too challenging for the analysis modules. Similar to the benchmark "TindellScaled", most of the models in this benchmark can be solved without additional hardware costs (by using only the initially available hardware architecture without modifying it). As stated before, the global analysis module “KL” works especially well if used on DSE problems where only little or even no changes at all of the hardware architecture are required in order to find solutions.

In Table 6.12 some details about the models contained in this benchmark are provided.

#Tasks	***x* Unallocated Tasks [%]**				
	$x < 10\%$	$10\% \leq x < 30\%$	$30\% \leq x < 50\%$	$50\% \leq x < 70\%$	**Total**
100	188	140	130	12	**470**
200	2	198	10	6	**216**
300	0	10	4	4	**18**
Total	**190**	**348**	**144**	**22**	**704**

Table 6.12. – Benchmark “Generated”: Number of Models with total Number of Tasks (Rows) of which x Percent are Unallocated (Columns)

The benchmark contains 352 models. All models consists of either 100, 200 or 300 tasks. Between 10% and of 70% of these tasks are unallocated.

6.5.2.1. Results: Global Analysis

This benchmark consists of 352 models. Table 6.13 provides an overview of the results of the global analysis for this benchmark. Not all available global analysis modules have been used for this benchmark. Namely “LP_GLP” was not used, because based on the previous results, it was clear that its backend GLPK would hardly be able to solve any of the provided DSE problems in reasonable time. The “MZ” module has been used only for smaller models. The global modules “KL”, “LP_GRB” and “LP_CPX” have been ran 704 times each, the “MZ” module only 240 times.

The results of all global analysis modules except for “MZ” are very good, considering the size of the models. For unknown reasons the “MZ” module failed to compute valid solutions for all 240 DSE problems.

Global Module	**Successful**	**Timeout**	**Incomplete**	**Error**	**Total**
KL	597	66	41	0	**704**
LP_GRB	556	77	71	0	**704**
LP_CPX	568	71	65	0	**704**
MZ	0	0	240	0	**240**
Total	**1721**	**214**	**417**	**0**	**2352**

Table 6.13. – Benchmark “Generated”: General Information

Figure 6.11 compares the runtimes of all successful global analysis runs. The boxplot on the left side shows all runs and the boxplot on the right side only those results with a runtime smaller or equal to 100s. The medians of both “LP_GRB” and “LP_CPX” are slightly lower than the median of the “KL” module. The box for “KL” is slightly larger and its upper end is located at a larger runtime value than the upper ends of both the box for “LP_GRB” and “LP_CPX”.

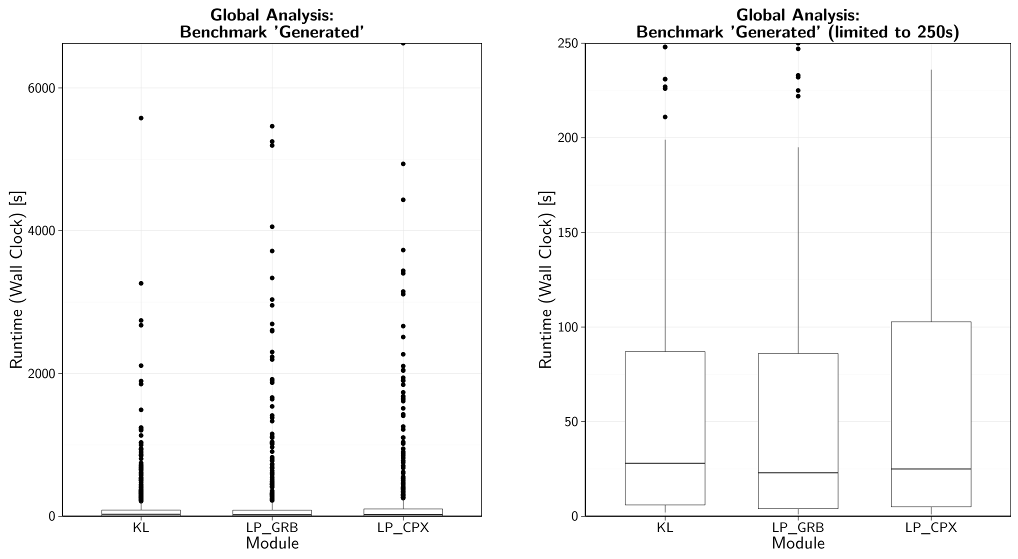


Figure 6.11. – Benchmark “Generated”, Comparing Runtimes

Module	Runtime (Wall Clock) [s]			Cost		
	Mean	**Median**	**Std. Dev.**	**Mean**	**Median**	**Std. Dev.**
KL	98.24	20.00	286.53	2386.33	2318.00	745.99
LP_CPX	195.28	18.50	566.82	2386.33	2318.00	745.99
LP_GRB	190.31	16.50	628.04	2386.33	2318.00	745.99

Table 6.14. – Benchmark “Generated”: Summary of Global Results (Where All Modules found a Solution)

Table 6.14 provides the median, mean and standard deviation of the results of the global analysis modules limited to those benchmark models for which all global analysis modules found a solution (which avoids bias).

Method	**Comparing Cost**		
	Less than KL	**Equal**	**Greater than KL**
LP_GRB	0	523	0
LP_CPX	0	535	0

Table 6.15. – Hypothesis 1: Comparing Cost of Successful Runs (“Generated”)

Based on the data for this benchmark shown in Table 6.15 both the null hypothesis H_0^{1a} and the null hypothesis H_0^{1b} can be rejected favoring the alternative hypotheses H_A^{1a} (“The majority of “LP_∗” solutions have lower or equal hardware costs than “KL”) and H_A^{1b} (“The majority of “LP_∗” solutions have similar costs than the “KL” solutions (depending on a given threshold)”).

Method	**Comparing Runtime**			$\delta = \lvert runtime_{KL} - runtime \rvert$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
LP_GRB	286	32	205	283	104	136
LP_CPX	225	38	272	279	98	158

Table 6.16. – Hypothesis 2: Comparing Runtime (Wall Clock) of Successful Runs (“Generated”)

Regarding the runtimes the picture is not that clear. Table 6.16 shows that the analysis module “LP_GRB” (with backend Gurobi) in most of the cases had a shorter runtime than “KL” which allows us to reject the null hypothesis H_0^2 favoring the alternative hypothesis H_A^2 (““LP_GRB” finds the majority of solutions in less time than “KL””). However, global module “LP_CPX” (with backend CPlex) has been slower than “KL” in around 272 of 535 comparable cases. Based on this result, the null hypothesis H_0^2 cannot be rejected for “LP_CPX”.

6.5.2.2. Results: Local Analysis

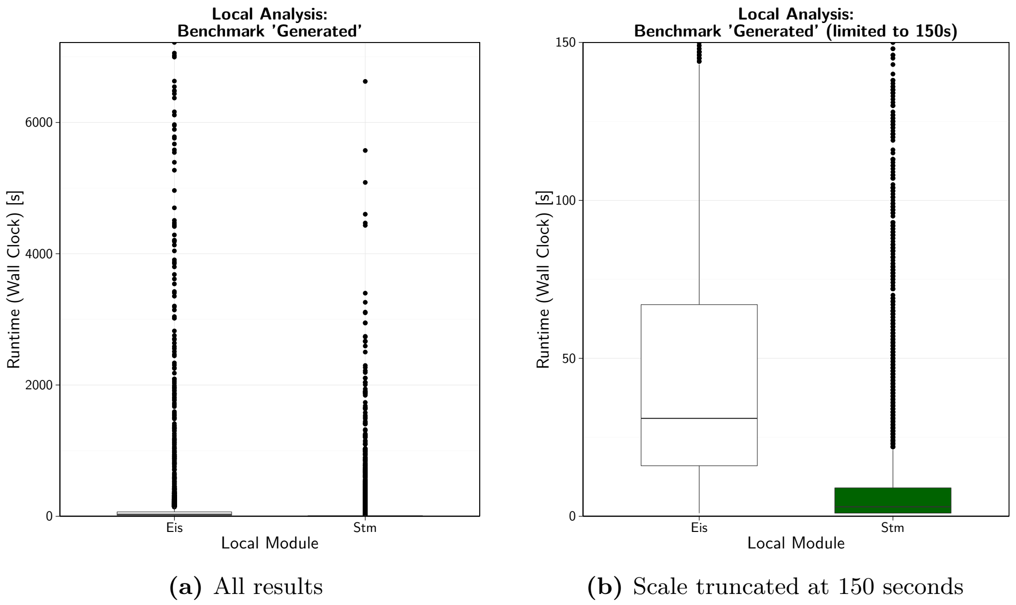
The results of the local analyses for this benchmark are more interesting than the local results of the other benchmark due to the greater variability of the models. The hardware architectures of the subsystems had between 11 and 35 ECUs and therefore are much larger than the subsystems found in e.g. the benchmark “TindellScaled”. This allows a more detailed evaluation of the strengths and weaknesses of the local analysis modules.

Table 6.17 shows an overview of the results. Even though still a large number of local analysis runs have been evaluated, the total number of analysis runs is much smaller than for the benchmark “TindellScaled”. The reason for the small number of runs is that the hardware architectures of the global DSE problems consists of only a small

Local Module	Successful	Error	Total
Eis	5883	298	**6181**
Stm	5957	224	**6181**
Total	**11840**	**522**	**12362**

Table 6.17. – Benchmark "Generated": General Information about Local Analysis Results

number of subsystem which are significantly larger than the subsystems used e.g. in the benchmark "TindellScaled". The table also reveals that both local analysis modules failed to calculate valid results in approximately 5% of all cases. I assume that this is caused by corner cases contained in the generated models which are not handled appropriately by the respective local analysis module. As most of the results are valid and the time situation for this thesis is too tight no further action has been taken to identify these corner cases. Note that the validity of all local analysis runs marked as "successful" in the table is proven because of the automatic checks for integrity performed after each local analysis run.



(a) All results **(b)** Scale truncated at 150 seconds

Figure 6.12. – Benchmark "Generated": Overview of Local Analysis Results

An overview comparing the runtimes of both approaches is depicted in Figure 6.12 via boxplots. In this and all following figures only those local analysis cases are compared where both modules calculated a valid solution. The boxplot on the left side (Figure 6.12a) shows the runtimes (wall clock) of all analysis runs. The runtimes vary between 0 and approximately 7000 seconds but most of the runs with large runtimes can be considered outliers. The plot on the right side (Figure 6.12b) shows the same results but limiting

Module	Runtime (Wall Clock) [s]			Cost		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Eis	123.26	31.00	512.07	549.86	542.00	151.94
Stm	51.45	3.00	274.19	549.86	542.00	151.94

Table 6.18. – Benchmark “Generated”: Statistical Summary of Local Analysis Results

the y-axis to values between 0 and 150 seconds thus hiding most of the outliers. The figure shows clearly that “Stm” was faster than “Eis” in most of the cases; the box corresponding to the results of “Stm” does not even overlap with the one for “Eis”. These findings are substantiated by the results shown in Table 6.18. The median of the “Eis” results is ten times greater than the median of the “Stm” results with respect to the runtime. Again the question is what the price for this great speed-up with respect to the quality of the solutions (measured in cost) is. The table gives an answer to this question as well: the median of the calculated costs of “Stm” is identical to that of “Eis”. In other words: a speed-up of factor 10 can be achieved using the Spare-Time/MaxWCET approach without loss of quality.

Method	Comparing Cost			$\delta = \lvert cost_{Eis} - cost \rvert$		
	Less than Eis	Equal	Greater than Eis	$\delta \leq 11$	$11 < \delta \leq 37$	$37 < \delta$
Stm	0	5762	0	5762	0	0

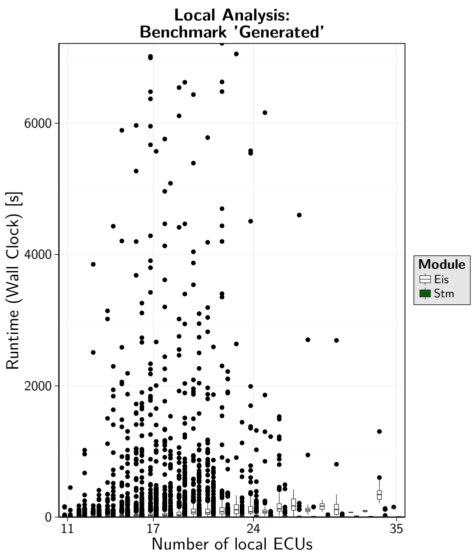
Table 6.19. – Hypothesis 3: Comparing Local Cost of Successful Runs (“Generated”)

Method	Comparing Runtime			$\delta = \lvert runtime_{Eis} - runtime \rvert$		
	Less than Eis	Equal	Greater than Eis	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
Stm	5338	59	365	1279	3110	1373

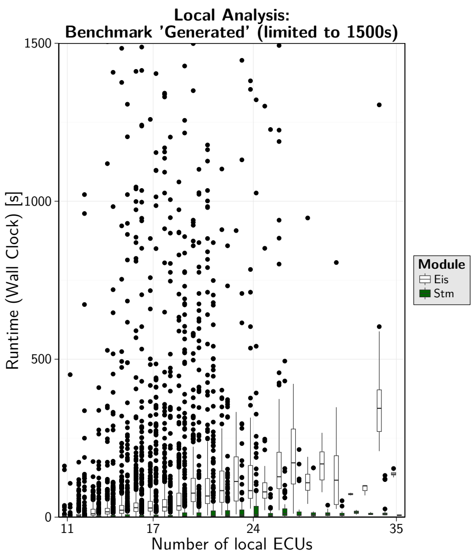
Table 6.20. – Hypothesis 4: Comparing Local Runtime of Successful Runs (“Generated”)

Tables 6.19 and 6.20 contain the data required to find an answer for Hypothesis 3 and 4. Because of the identical median of the calculated costs, null hypothesis H_0^3 can be rejected in favor of the alternative hypothesis H_A^3 (“The majority of solutions found using “Stm” have lower or equal hardware costs than those found by “Eis””). Regarding the runtimes, as shown in to Table 6.20 “Eis” has been faster than “Stm” for only approximately 5% of the local DSE problems contained in this benchmark. For 1% of the models both analysis modules had identical runtimes and for 94% the runtimes of “Stm” have been shorter. Based on these results the null hypothesis H_0^4 can be rejected favoring the alternative hypothesis H_A^4 (““Stm” finds the majority of solutions in less time than “Eis””).

In Figures 6.13, 6.14 and 6.15 the results are categorized by the number of local ECUs, the total number of tasks (system-wide) and the number of unallocated tasks (pre-allocated to the respective subsystem).

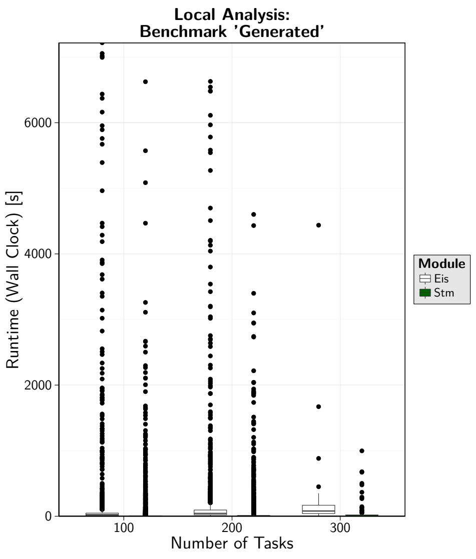


(a) All Results

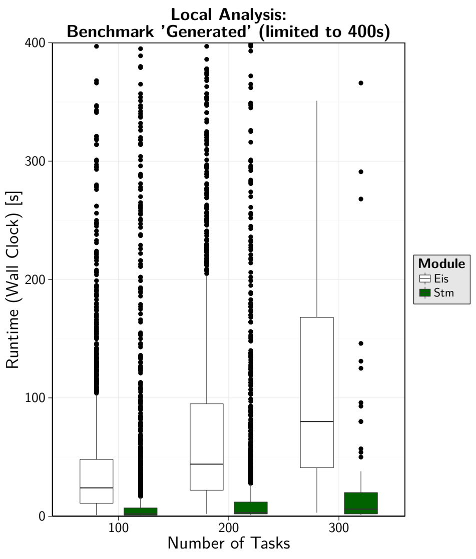


(b) Scale truncated at 1500 seconds

Figure 6.13. – Benchmark "Generated": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of local ECUs

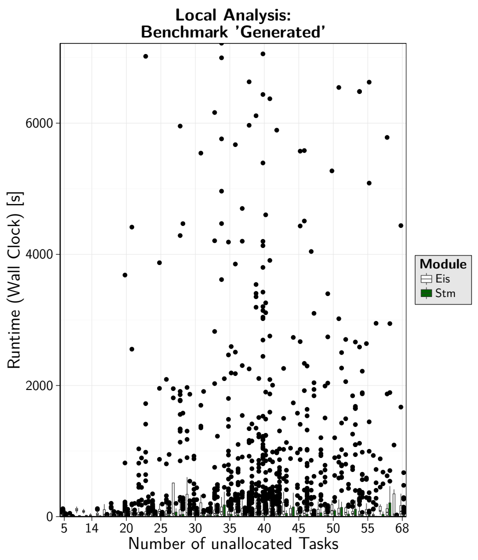


(a) All Results

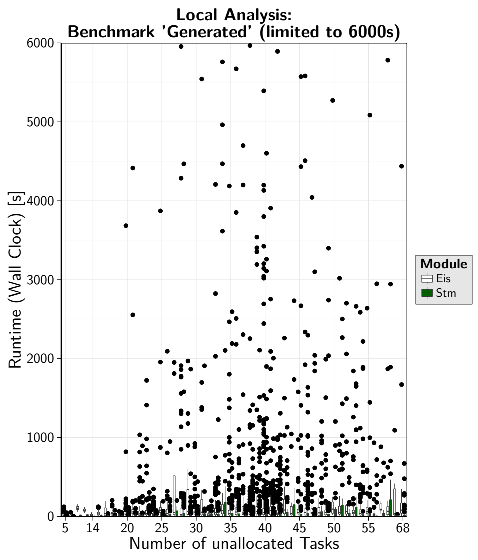


(b) Scale truncated at 400 seconds

Figure 6.14. – Benchmark "Generated": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of Tasks



(a) All Results



(b) Scale truncated at 10 seconds

Figure 6.15. – Benchmark "Generated": Runtime (Wall Clock) of Local Analysis Modules plotted against Number of unallocated Tasks

Obviously, increasing the total number of tasks increases the runtime of the local analyses because the number of tasks to be allocated to each subsystem increases, too. Figure 6.14 further shows that the runtime of "Stm" increases significantly slower with increasing number of tasks than the runtime of "Eis". The number of locally available ECUs is another important factor influencing the runtime as can be seen in Figure 6.13. The number of unallocated tasks, however, does not seem to be correlated with the runtime (see Figure 6.15).

Correlation (Stm)	**Runtime (Wall Clock)**		**Runtime (CPU)**	
	Linear	**Logarithmic**	**Linear**	**Logarithmic**
Number of Tasks	0.00	0.11	0.01	0.15
Unallocated Tasks	0.16	0.36	0.09	0.35
Tasks * Unallocated Tasks	0.08	0.23	0.05	0.25
Unallocated Tasks * ECUs	0.15	0.37	0.10	0.37

Table 6.21. – Benchmark "Generated": Analysis of Correlations: Local Module "Stm"

Tables 6.21 and 6.22 show for each local analysis module the correlation between the different factors number of tasks (total), number of unallocated tasks (local), number of tasks multiplied with the number of unallocated tasks, and number of tasks multiplied with the number of local ECUs, each in relation with the runtime. The different notions of runtime used are wall clock and CPU time. For both of them the measured values are used directly ("linear") and additionally the result of calculating the natural logarithm

Correlation (Eis)	Runtime (Wall Clock)		Runtime (CPU)	
	Linear	Logarithmic	Linear	Logarithmic
Number of Tasks	0.03	0.27	0.03	0.31
Unallocated Tasks	0.12	0.37	0.08	0.35
Tasks * Unallocated Tasks	0.07	0.32	0.05	0.33
Unallocated Tasks * ECUs	0.14	0.49	0.10	0.47

Table 6.22. – Benchmark "Generated": Analysis of Correlations: Local Module "Eis"

of the values is used. The found correlations are a little bit stronger than those found for the benchmark "TindellScaled". For both local analysis modules the strongest correlation is between the number of unallocated tasks multiplied by the number of ECUs with the logarithm of the runtimes, both wall clock and CPU. But even the strongest correlation found is less than 0.5. This substantiates the hypothesis that there exists no simple dependence from one or two properties of the used models to the runtime of the local analysis modules.

6.5.2.3. Single Core vs. Multi Core

Figure 6.16 compares for each of the global analysis modules "KL", "LP_GRB" and "LP_CPX" the runtimes of all models where both analysis runs — the first one limited to 12 CPU cores and the second one limited to one CPU core — successful found a solution. The results are grouped by the total number of tasks (100, 200, or 300). As each of these groups contains several individual results, boxplots are used to aggregate the runtimes per group. On the left side, all results including all outliers are depicted while on the right side the y-axes have been limited such that the boxplots and their whiskers completely fit into the chosen ranges.

Surprisingly, the presented results show that in general it does not pay off to use the built-in multi-threading capabilities of the multi-core-enabled MILP solvers Gurobi and CPlex for this benchmark. Furthermore the results indicate that the mean runtimes of the analysis runs where up to 12 CPU cores have been available are significant greater than the mean runtimes for those analysis runs where only one core has been used.

Based on the presented results it is not possible to reject null hypothesis H_0^5 for any of the global analysis methods "KL", "LP_GRB" or "LP_CPX".

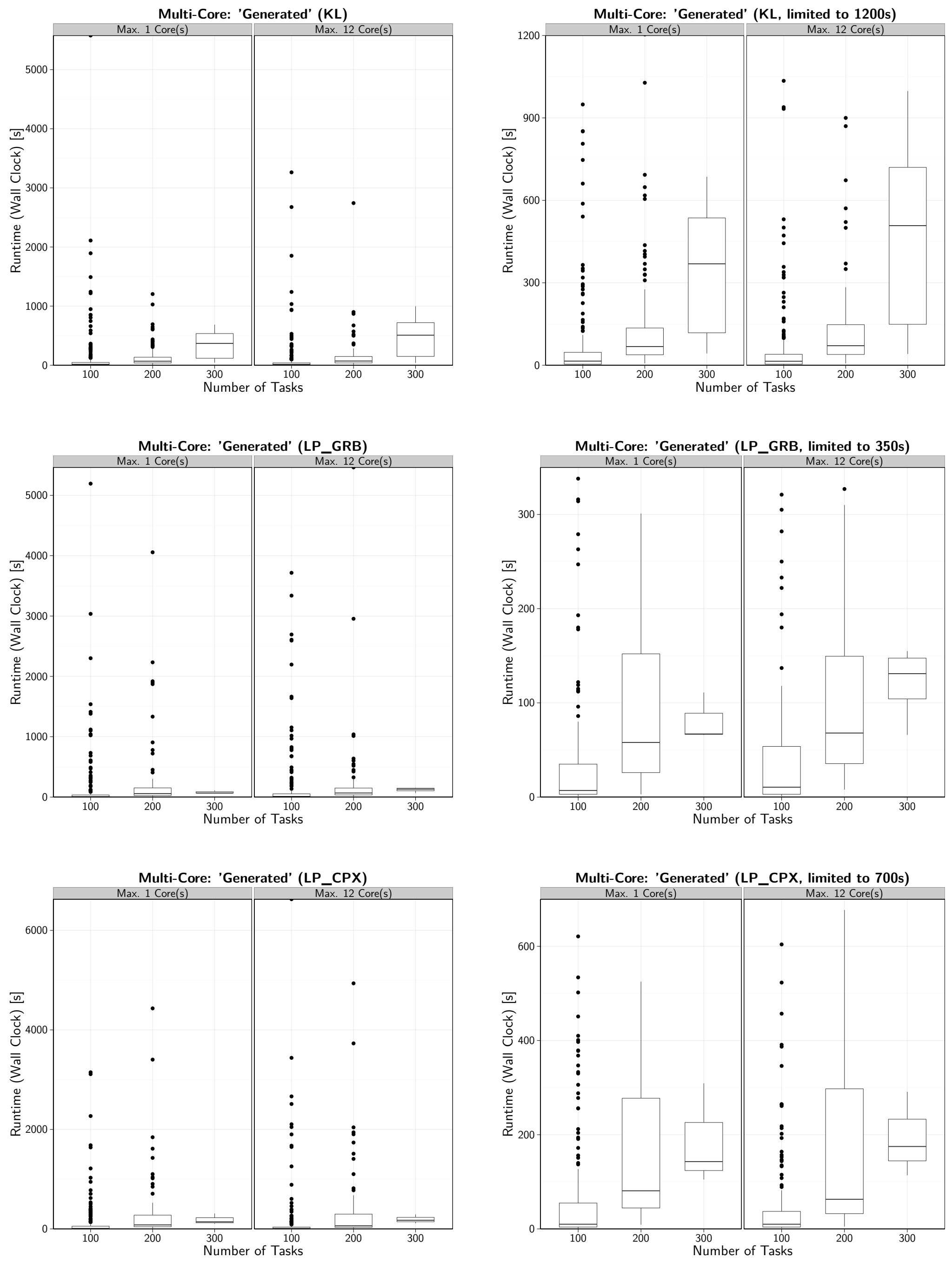


Figure 6.16. – Benchmark "Generated": Single-Core vs. Multi-Core

6.5.3. Benchmark “ViDAs”

The third benchmark with the name “ViDAs” is based on the results of a project group of the same name which was one of the master courses given at the University of Oldenburg in 2009/2010. The eight participating students had been given the task to develop a driver assistance system capable of automatically joining a two-lane motorway by using sensors to identify a sufficiently large gap between the cars driving on the right lane. The assistance system has been successfully implemented for a hardware system used for building a set of (identical) model cars. More information can be found in the final report [Bao+10b] and on the project group’s website [Bao+10a].

The driver assistance system was partially developed using Matlab Simulink®. The timing information has been extracted from the Matlab model using the approaches presented in [Bük12]. Real-Time Workshop® (recently renamed *Matlab Coder™*) was used to generate source code. The task creation as presented in [Bük12] was then used to partition the software into multiple software tasks. During that step, 10 unique solutions have been created, each with different numbers and sizes of software tasks. A common set of ECU-types has been defined for all solutions. For each of the software tasks timing analysis was applied to determine worst case execution times assuming that each software task is executable on every ECU type (based on binary executables compiled specifically for the respective target ECU type).

For the benchmark “ViDAs” used in my work four representatives have been chosen out of the 10 available solutions. The ViDAs driver assistance system is considered to be an extension of an existing system. Therefore, all software tasks stemming from the ViDAs project are considered to be unallocated initially. The existing system is the same as for the benchmark “TindellScaled”, which means that the hardware architectures are identical and that the same allocated software tasks of the benchmark “TindellScaled” are used in the benchmark “ViDAs” as well and are allocated to the same ECUs as in the benchmark “TindellScaled” (see Section 6.5.1 for details).

The (unallocated) tasks found by the task creation process could not be used directly in this benchmark. As of today the local analysis approach still requires that task networks have a tree-like structure; the tasks created by the task creation process violate that limitation. They have been changed manually by removing some signals in a way that they form a tree-like structure.

Each of the four models has been used as a template for creating a set of models by scaling the respective initial model in the same way as for the benchmark “TindellScaled”. Those sets Set 1, Set 2, Set 3 and Set 4 constitute the benchmark “ViDAsOriginal”.

During the evaluation it became obvious that the other benchmarks are not ideal for demonstrating the strength of the the global analysis approach proposed in this thesis. The reason for this is that all unallocated software tasks contained in each of the benchmark models of the “TindellScaled”, “Generated” and the “ViDAsOriginal” benchmarks can be allocate with only minimal additional hardware costs, as can be seen by the lower bound marker (dashed line) in Figures 6.2, 6.3 (benchmark “TindellScaled”) and Figures 6.17–6.20. This is a result of the initial utilization of all ECUs being too low to be considered a challenge to the optimization approaches.

As a consequence, I decided to modify the original ViDAs system such that significant additional hardware cost are required in order to allocate all tasks onto the hardware architecture. First, the costs of the ECU types have been changed (see Table 6.23), then the worst case execution times of all allocated and unallocated software tasks have been increased.

The four modified models have been used as templates for creating four sets which together constitute the benchmark "ViDAsExpensive".

ECU Type	**Cost**		**Memory**
	ViDAsOriginal	**ViDAsExpensive**	
leon3_LRU_middle_cache	38	32	30000
leon3_no_cache	28	28	20000
arm7_standard	18	22	15000
arm7_fastmem	14	25	10000

Table 6.23. – Benchmark "ViDAs": ECU Type Cost

6.5.3.1. Results: Global Analysis

ViDAsOriginal Table 6.24 shows general information about the results for this benchmark. This benchmark consists of 120 models. As for the other benchmarks, the overall analysis "MZ" has been used only on smaller models which explains the small number of total analysis runs for that module.

The data shows that only a small number of models could be analyzed successfully. This indicates that the benchmark models are significantly harder to solve than those of the previous benchmarks. The global analysis module "KL" clearly is the winner with respect to scalability: 38 out of 240 analysis runs have been successful. Both "LP_$*$" modules have been significantly less successful, with 26 ("LP_GRB") and 21 ("LP_CPX") successful analysis runs, respectively. The "MZ" was unable to solve any of the examples successfully, probably because of a bug in the implementation.

The Figures 6.17 to 6.20 depict for each of the four sets the data measured for the global analysis modules "MZ", "LP_GRB" and "LP_CPX". In each figure the runtimes are depicted in the left chart while the costs of the computed solutions are depicted in the figures on the right side. For Set 1 the "KL" does obviously scale much better with increasing number of tasks. For models with 135 tasks and above none of the other modules could find a solution at all. The same trend can be seen in the results for Set 2, however not that clearly. Interestingly, "KL" was unable to find solutions for the largest models contained in Set 3 and Set 4, respectively. In Set 3 only "LP_GRB" (with backend Gurobi) found a solution for the model with 148 tasks. In Set 4 a solution for the model with 99 tasks could only be found by "LP_CPX" (with backend CPlex).

The charts depicting the costs of the solutions show that all global modules under evaluation found solutions of similar quality.

Global Module	Successful	Timeout	Incomplete	Error	Total
KL	38	202	0	0	**240**
LP_GRB	26	214	0	0	**240**
LP_CPX	21	217	2	0	**240**
MZ	0	0	16	0	**16**
Total	**85**	**633**	**18**	**0**	**736**

Table 6.24. – Benchmark "ViDAsOriginal": General Information/Global Analysis

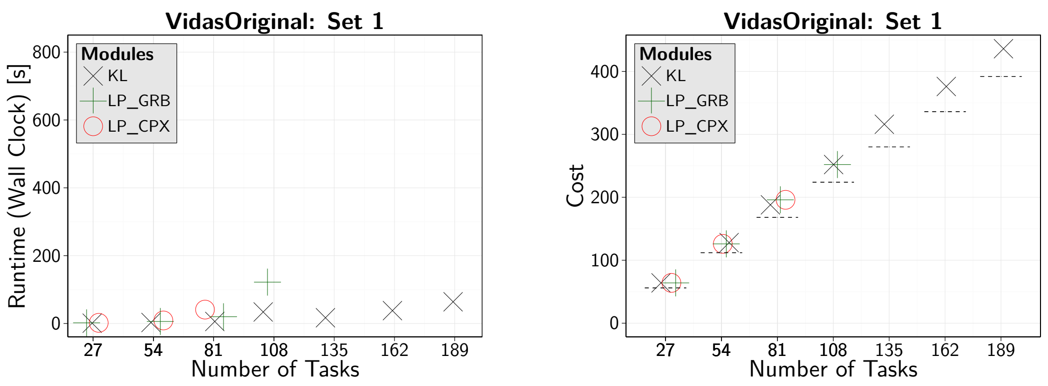


Figure 6.17. – Benchmark "ViDAsOriginal": Global Analysis, Set 1. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

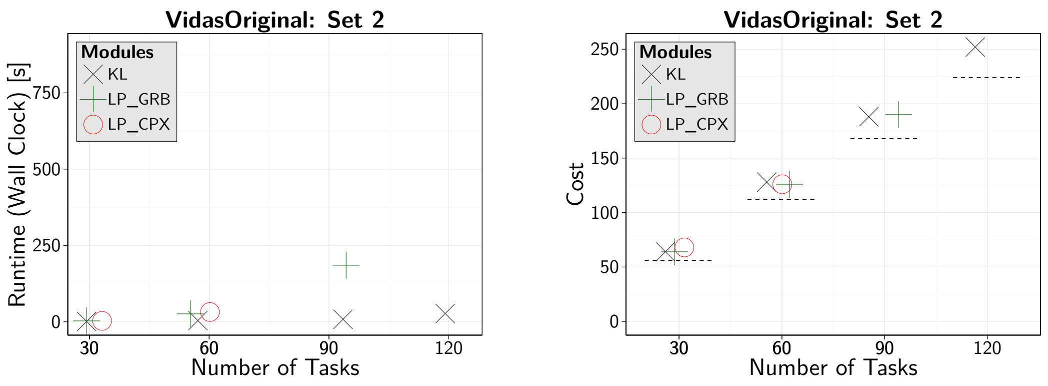


Figure 6.18. – Benchmark "ViDAsOriginal": Global Analysis, Set 2. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

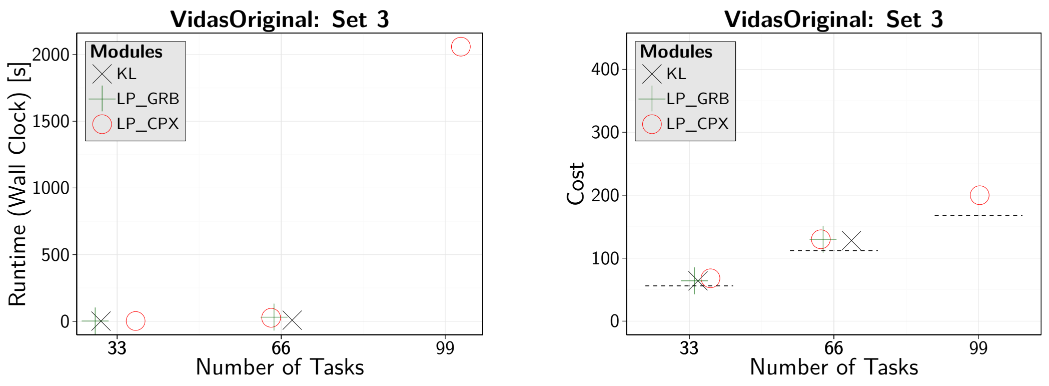


Figure 6.19. – Benchmark "ViDAsOriginal": Global Analysis, Set 3. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

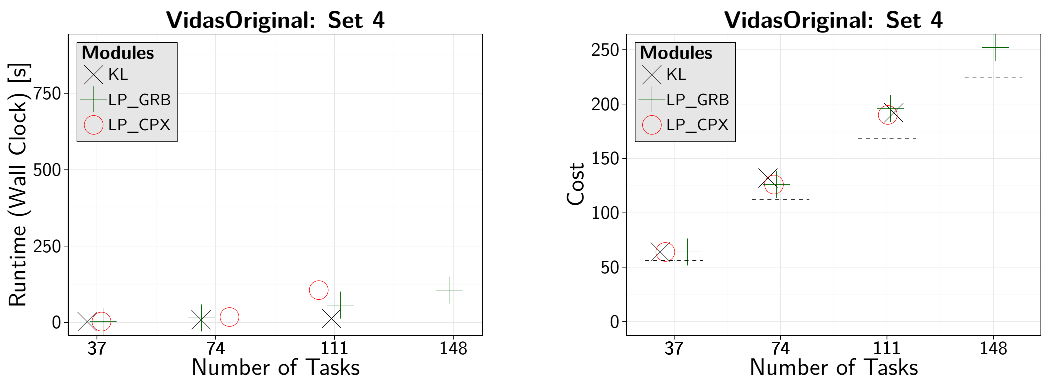


Figure 6.20. – Benchmark "ViDAsOriginal": Global Analysis, Set 4. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

Method	**Comparing Cost**			$\delta = \lvert cost_{KL} - cost \rvert$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 14$	$14 < \delta \leq 38$	$38 < \delta$
LP_GRB	7	10	7	24	0	0
LP_CPX	9	5	6	20	0	0

Table 6.25. – Hypothesis 1: Comparing Cost of Successful Runs ("VidasOriginal")

The charts depicting the costs of the solutions (on the right side) show that all global modules under evaluation found solutions of similar quality (cost). In Table 6.25 the required data for discussing Hypothesis 1 is presented. Each row in the table is calculated based on only those DSE problems where both the “KL” module and the module named in the first column found a valid solution. The table confirms that the calculated costs are very similar for all the global analysis modules. The absolute cost difference is in all cases less than the cost of the cheapest ECU type. The table also shows that in fact “LP_GRB” is on a par with “KL” with respect to the resulting costs: Each analysis module found in 7 cases a slightly better solution than the other one; in 10 cases the solutions had equal costs. At the first glance the results for “LP_CPX” are even better, but one has to consider the smaller number of comparable results as well (only 20 compared to 24 for “KL” vs. “LP_GRB”). Based on the data the null hypotheses H_0^{1a} and H_0^{1b} can be rejected for this benchmark in favor of the alternative hypotheses H_A^{1a} (“The majority of “LP_∗” solutions have lower or equal hardware costs than “KL”) and H_A^{1b} (“The majority of “LP_∗” solutions have similar costs than the “KL” solutions (depending on a given threshold)”).

The data presented in Table 6.26 regarding the runtime of the “LP_∗” modules compared to “KL” does not allow to reject the null hypothesis H_0^2. It is absolutely clear that the “LP_∗” modules have a greater runtime in most of the cases, even though the measured runtimes are still relatively close to each other.

Method	**Comparing Runtime**			$\delta = \lvert runtime_{KL} - runtime \rvert$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
LP_GRB	0	2	22	12	8	4
LP_CPX	0	2	18	12	4	4

Table 6.26. – Hypothesis 2: Comparing Runtime (Wall Clock) of Successful Runs (“VidasOriginal”)

ViDAsExpensive Compared to the original ViDAs benchmark models the modified *expensive* variants are much harder to solve as can be seen in Table 6.27. This benchmark consists of 40 models. Most of the analysis runs hit the time limit of 60 minutes. Later in this section the multi-core evaluation (Section 6.5.3.3) shows that in the case of the “KL” global analysis module the local analysis modules is responsible for the huge runtime. The runtime of the local analysis is huge for the “LP_∗” global modules as well, but here the runtime for the global analysis is very huge, too. The results of all successful analysis runs are depicted in Figures 6.21–6.24.

All global analysis modules have been able to successfully solve only half that many models in Set 1 of the benchmark “ViDAsExpensive” than in the benchmark “ViDAsOriginal” (see Figure 6.21). The larger WCETs of the tasks contained in the *expensive* variants of the original models reduces the remaining capacity on the ECUs and in the same time leads to a larger amount of capacity required by the unallocated tasks. The figure on the right shows that the costs of the found solutions are much greater compared to the results calculated for the original set (see Figure 6.17). The distance between the initial costs and the costs of the solutions is larger because more modifications of the hardware architecture are required to successfully allocate the whole task network. The fact that the costs of the ECU-type have been redefined for the *expensive* models such that they are very close to each other (in contrary to the costs of the ECU-types in the original set, see Table 6.23) further complicates the DSE problems due to the increased number of possible hardware configuration with similar total costs which have to be considered during the optimization phase.

The Set 2 in this benchmark seems to be significantly easier to solve for “KL” but surprisingly much harder for the other approaches, see Figure 6.18. Only “KL” and “LP_GRB” found solutions for models in Set 3 (“ViDAsExpensive”); “KL” scales slightly better (Figure 6.19). All three methods found solutions for the smallest model in Set 4. The solutions of both “LP_GRB” and “LP_CPX” had lower costs than the solution calculated by “KL”. The next larger DSE problem could by successfully solved only by “LP_GRB”.

Global Module	**Successful**	**Timeout**	**Incomplete**	**Error**	**Total**
KL	26	54	0	0	**80**
LP_GRB	10	70	0	0	**80**
LP_CPX	6	70	4	0	**80**
MZ	0	0	16	0	**16**
Total	**42**	**194**	**20**	**0**	**256**

Table 6.27. – Benchmark “ViDAsExpensive”: General Information about Global Analysis

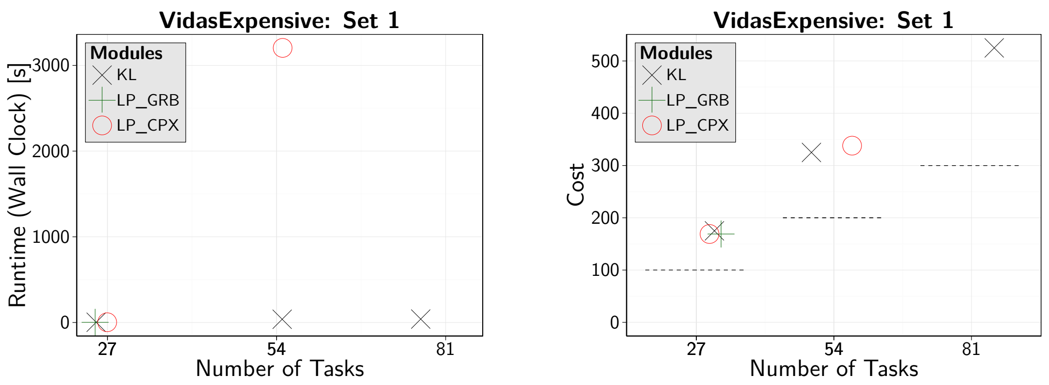


Figure 6.21. – Benchmark “ViDAsExpensive”: Global Analysis, Set 1. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

A textual comparison of the results of all global analysis modules regarding the cost is given in Table 6.28. The data shows that both “LP_GRB” and “LP_CPX” calculate solutions with smaller costs in most of the cases. The differences of their calculated costs

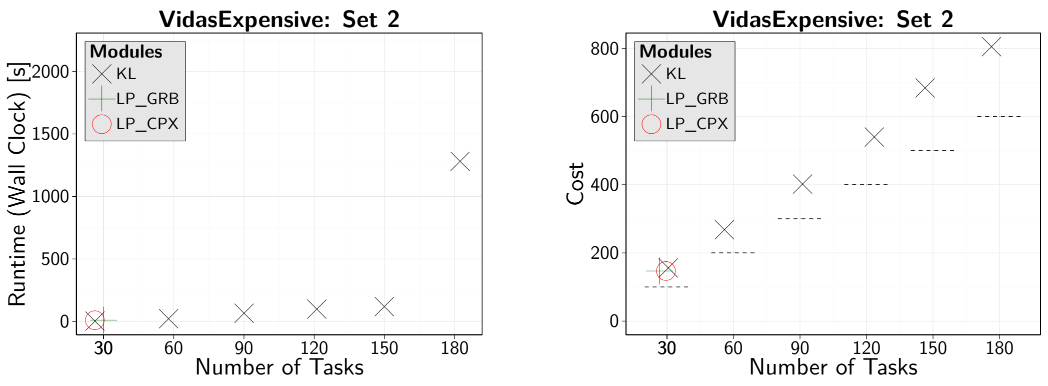


Figure 6.22. – Benchmark "ViDAsExpensive": Global Analysis, Set 2. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

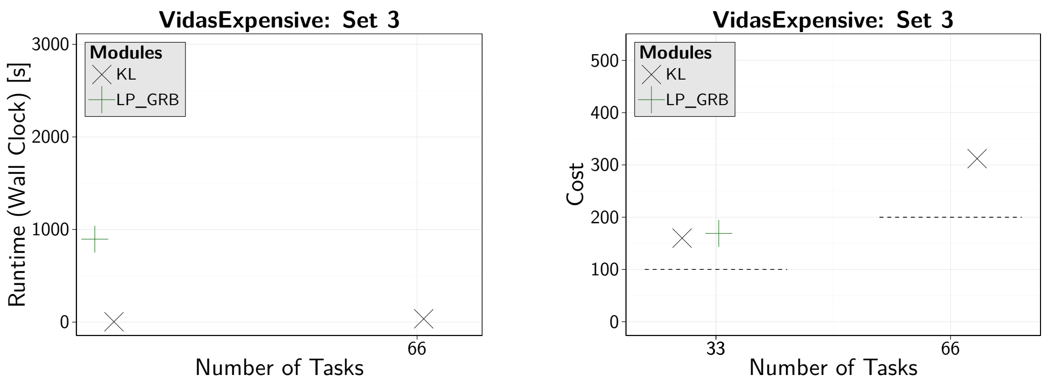


Figure 6.23. – Benchmark "ViDAsExpensive": Global Analysis, Set 3. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

Method	**Comparing Cost**			$\delta = \|cost_{KL} - cost\|$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 22$	$22 < \delta \leq 32$	$32 < \delta$
LP_GRB	7	0	2	7	2	0
LP_CPX	5	0	1	5	1	0

Table 6.28. – Hypothesis 1: Comparing Cost of Successful Runs ("VidasExpensive")

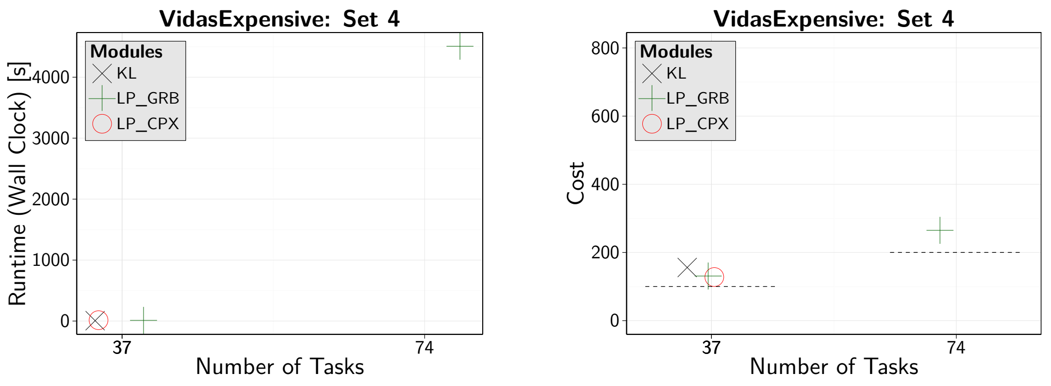


Figure 6.24. – Benchmark "ViDAsExpensive": Global Analysis, Set 4. The dashed line marks the initial hardware cost. Note that for better readability some points have been slightly shifted along the x-axis.

to the costs of the solutions computed by "KL" are in all cases not smaller or equal to the cost of the most expensive ECU-type (which costs 32). Because the number of comparable analysis runs is very small (e.g. only one third of the solutions found by "KL" could be compared to "LP_GRB"), I decided not to evaluate Hypothesis 1a and Hypothesis 1b ("The costs of the solutions are better"/"The costs are similar"). Rejecting the corresponding null hypotheses based on this small data set would be highly arguable.

Method	**Comparing Runtime**			$\delta = \lvert runtime_{KL} - runtime\rvert$		
	Less than KL	**Equal**	**Greater than KL**	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
LP_GRB	1	1	7	5	1	3
LP_CPX	1	1	4	4	1	1

Table 6.29. – Hypothesis 2: Comparing Runtime (Wall Clock) of Successful Runs ("Vidas-Expensive")

The comparison of the runtimes is shown in Table 6.29. Of course the number of comparable results is the same as for the costs. But additionally the data clearly does not allow to reject the null hypothesis H_0^2 ("LP_∗" is at least as fast as "KL"). The "KL" heuristic is faster for most of the models found in benchmark "ViDAsExpensive" and/or has a better chance to find solutions.

6.5.3.2. Results: Local Analysis

For the benchmark "ViDAsOriginal" the local analysis module "Eis" encountered an error during nearly every second analysis run, as can be seen in Table 6.30. Nevertheless half of the analyzed models have been processed successfully and the computed solutions passed the post-checks validating their consistency. The results for the benchmark

"ViDAsExpensive" are much better: An error condition was triggered only in about 5% of the analysis runs.

Module	**Successful**	**Error**	**Total**
Eis	882	702	**1584**
Stm	1582	2	**1584**
Total	**2464**	**704**	**3168**

Table 6.30. – Benchmark "ViDAsOriginal": General Information about Local Analysis Results

ViDAsOriginal The comparable data measured for all successful analysis runs is depicted in Figure 6.25. The left chart shows all results. It can be seen that the "Eis" local analysis module produced lots of outliers varying between 20s up to 4000s. The "Stm" approach behaves much better with only a few outliers very close to 20s. The right chart shows only the relevant parts of the boxplots. The median of the "Stm" approach is at a significantly lower level (approximately at the lower end of the box of the "Eis" results) than the median of the "Eis" approach. The absolute values for the medians are given in Table 6.31. Here, the huge standard deviation calculated for the measured runtimes of the "Eis" analysis runs stands out. Again, the mean of the costs of the computed solutions is slightly better for the "Eis" approach than for the "Stm" approach, but the cost medians are identical. In statistics, an important property of the notion of **median** is its robustness against outliers compared to the notion of **mean**. A lower mean of the costs of the solutions calculated by "Eis" compared to the "Stm" but identical median values might result from the outliers in the data set. The larger standard deviation of the "Eis" results supports this presumption.

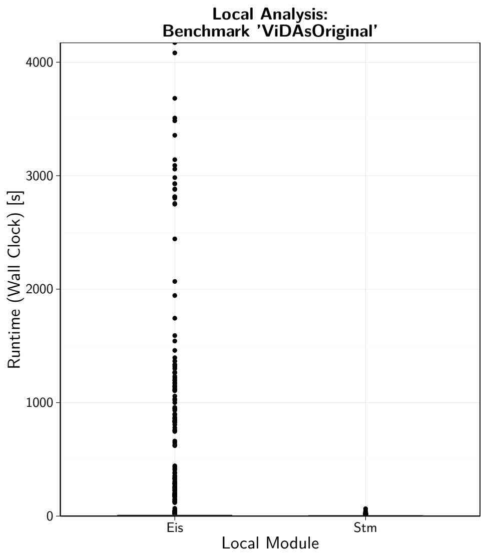
The box of the "Stm" results starts at a lower y-value than the box of the "Eis" results and its height is much smaller. Most of the runtimes measured when using the "Stm" local analysis module on the benchmark "VidasOriginal" are significantly lower than the runtimes measured when using the "Eis" local analysis module.

Module	**Runtime (Wall Clock) [s]**			**Cost**		
	Mean	**Median**	**Std. Dev.**	**Mean**	**Median**	**Std. Dev.**
Eis	582.95	5.00	1658.53	28.53	32.00	9.38
Stm	20.81	3.00	101.50	31.67	32.00	3.61

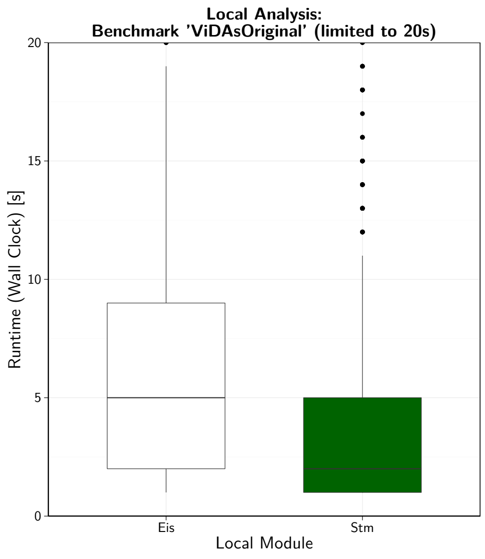
Table 6.31. – Benchmark "ViDAsOriginal": Statistical Summary of Local Analysis Results

A chart where the number of ECUs is plotted against the runtime is not included in this work, because it contained exactly one category with six ECUs and despite of that is identical to Figure 6.25.

Table 6.34 compares the resulting costs of solutions calculated by "Stm" and "Eis". In only 1% of all analysis runs (total number: 8) the costs of the "Eis" solutions have been



(a) All results



(b) Scale truncated at 10 seconds

Figure 6.25. – Benchmark "ViDAsOriginal": Overview of Local Analysis Results

Correlation (Stm)	**Runtime (Wall Clock)**		**Runtime (CPU)**	
	Linear	**Logarithmic**	**Linear**	**Logarithmic**
Number of Tasks	0.61	0.73	0.65	0.74
Unallocated Tasks	0.58	0.68	0.62	0.70
Tasks * Unallocated	0.47	0.58	0.51	0.60
Unallocated Tasks * ECUs	0.58	0.68	0.62	0.70

Table 6.32. – Benchmark "ViDAsOriginal": Analysis of Correlations: Local Module "Stm"

Correlation (Eis)	**Runtime (Wall Clock)**		**Runtime (CPU)**	
	Linear	**Logarithmic**	**Linear**	**Logarithmic**
Number of Tasks	0.73	0.83	0.77	0.85
Unallocated Tasks	0.70	0.78	0.73	0.80
Tasks * Unallocated	0.61	0.68	0.65	0.70
Unallocated Tasks * ECUs	0.70	0.78	0.73	0.80

Table 6.33. – Benchmark "ViDAsOriginal": Analysis of Correlations: Local Module "Eis"

Method	**Comparing Cost**			$\delta = \lvert cost_{Eis} - cost \rvert$		
	Less than Eis	**Equal**	**Greater than Eis**	$\delta \leq 11$	$11 < \delta \leq 37$	$37 < \delta$
Stm	0	874	8	882	0	0

Table 6.34. – Hypothesis 3: Comparing Local Cost of Successful Runs ("ViDAsOriginal")

Method	Comparing Runtime			$\delta = \|runtime_{Eis} - runtime\|$		
	Less than Eis	Equal	Greater than Eis	$\delta \leq 10$	$10 < \delta \leq 60$	$60 < \delta$
Stm	706	161	15	723	34	125

Table 6.35. – Hypothesis 4: Comparing Local Runtime of Successful Runs (“ViDAsOriginal”)

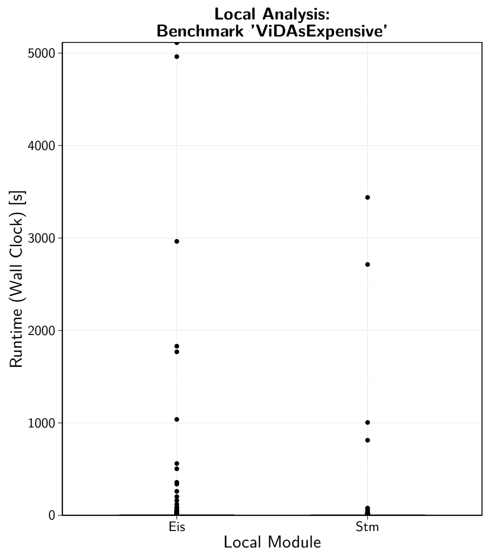
less than the costs of corresponding “Stm” solutions. In all cases the cost difference was smaller or equal to the cost for the cheapest ECU-type (which is 11). Therefore, null hypothesis H_0^3 can be rejected in favor of the alternative hypothesis H_A^3 (“The majority of solutions found using “Stm” have lower or equal hardware costs than those found by “Eis”). The situation for the runtime is quite definite, too. Table 6.35 shows that “Stm” has been faster in 80% of all cases and as fast as “Eis” in 18% of all cases. This allows us to reject null hypothesis H_0^4 in favor of H_A^4 (“Stm” finds the majority of solutions in less time than “Eis”).

ViDAsExpensive The models contained in the benchmark “ViDAsExpensive” did not trigger the bug present in the implementation of the “Eis” approach as frequently as in the previous benchmark. Table 6.36 still reports some errors for the local analysis module “Eis”, but only in 262 (approximately 4%) of all analyzed cases. In 12 cases the local analysis module “Stm” encountered an error, too. An overview of the runtimes is depicted in Figure 6.26. Compared to “Eis” the local analysis module “Stm” is faster in general. The median of the runtime of the “Stm” analysis runs is at the same height as the lower end of the boxplot representing the “Eis” results. The number of outliers is considerably smaller for the “Stm” results than for the “Eis” results. Table 6.37 provides the textual data measured for this benchmark for both local analysis modules. In this benchmark even the mean cost of the solutions calculated by the two analysis modules are almost identical. The table also shows the means and medians of the runtimes of “Stm” and “Eis”. According to both of them, “Stm” is faster than “Eis” in most cases.

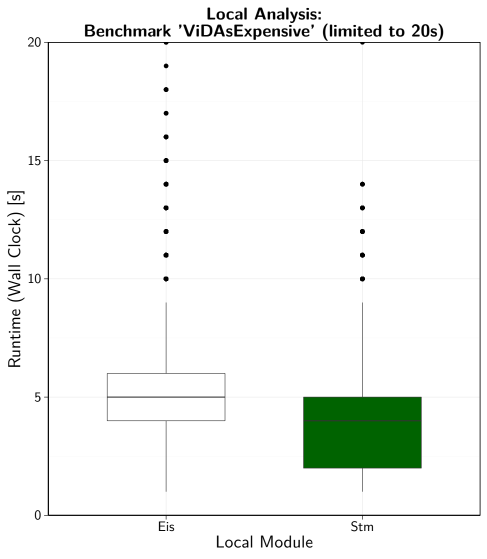
Module	Successful	Error	Total
Eis	5555	262	**5817**
Stm	5805	12	**5817**
Total	**11360**	**274**	**11634**

Table 6.36. – Benchmark “ViDAsExpensive”: General Information about Local Analysis Results

Based on the data in Table 6.38 the null hypothesis H_0^3 can be rejected favoring H_A^3 (“The majority of solutions found using “Stm” have lower or equal hardware costs than those found by “Eis”). Null hypothesis H_0^4 can also be rejected, based on the data presented in Table 6.39, in favor of Hypothesis H_A^4 (“Stm” finds the majority of solutions in less time than “Eis”).



(a) All results



(b) Scale truncated at 10 seconds

Figure 6.26. – Benchmark "ViDAsExpensive": Overview of Local Analysis Results

Module	**Runtime (Wall Clock) [s]**			**Cost**		
	Mean	**Median**	**Std. Dev.**	**Mean**	**Median**	**Std. Dev.**
Eis	66.36	5.00	638.15	59.79	56.00	10.94
Stm	20.83	4.00	318.85	59.70	56.00	10.62

Table 6.37. – Benchmark "ViDAsExpensive": Statistical Summary of Local Analysis Results

Method	**Comparing Cost**			**$\delta = \lvert cost_{Eis} - cost\rvert$**		
	Less than Eis	**Equal**	**Greater than Eis**	**$\delta \leq 22$**	**$22 < \delta \leq 32$**	**$32 < \delta$**
Stm	1092	4345	117	5554	0	0

Table 6.38. – Hypothesis 3: Comparing Local Cost of Successful Runs ("ViDAsExpensive")

Method	**Comparing Runtime**			**$\delta = \lvert runtime_{Eis} - runtime\rvert$**		
	Less than Eis	**Equal**	**Greater than Eis**	**$\delta \leq 10$**	**$10 < \delta \leq 60$**	**$60 < \delta$**
Stm	4633	798	123	5518	19	17

Table 6.39. – Hypothesis 4: Comparing Local Runtime of Successful Runs ("ViDAsExpensive")

Correlation (Stm)	Runtime (Wall Clock)		Runtime (CPU)	
	Linear	Logarithmic	Linear	Logarithmic
Number of Tasks	−0.02	0.69	−0.04	0.61
Unallocated Tasks	−0.02	0.71	−0.04	0.63
Tasks * Unallocated	−0.02	0.65	−0.04	0.58
Unallocated Tasks * ECUs	−0.02	0.71	−0.04	0.63

Table 6.40. – Benchmark "ViDAsExpensive": Analysis of Correlations: Local Module "Stm"

Correlation (Eis)	Runtime (Wall Clock)		Runtime (CPU)	
	Linear	Logarithmic	Linear	Logarithmic
Number of Tasks	−0.02	0.53	−0.03	0.39
Unallocated Tasks	−0.02	0.54	−0.03	0.41
Tasks * Unallocated	−0.03	0.49	−0.03	0.36
Unallocated Tasks * ECUs	−0.02	0.54	−0.03	0.41

Table 6.41. – Benchmark "ViDAsExpensive": Analysis of Correlations: Local Module "Eis"

6.5.3.3. Single Core vs. Multi Core

Again all DSE problems in both benchmarks "ViDAsOriginal" and "ViDAsExpensive" have been analyzed by each of the global analysis modules "KL", "LP_GRB" and "LP_CPX" twice, the first time with a restriction of the analysis software to use maximal 12 CPU cores and the second time with a restriction to use only one core. The results are described below, separately for each benchmark.

ViDAsOriginal The results for "KL" are shown in Figure 6.27. The global analysis module "KL" was responsible for only a small fraction of the total computation time while most of the computation time was spent on the local analysis backend. In some cases the use of 12 cores brought a small improvement of the runtimes. But there are more cases where the use of multiple cores slowed down the analysis process. Surprisingly, "KL" found no solutions at all for some larger models contained in Set 1 and Set 3 when using multiple cores. As "KL" itself does not use more than one core, the local analysis is likely to be responsible for this effect by causing timeouts due to an increased runtime when running with multiple threads.

For the global analysis module "LP_GRB" the situation looks better. For all models, using multiple cores led to better results with respect to the computation time. There are no surprises, where e.g. the use of multiple cores led to complete failures to find a solution. The huge runtime of the global analysis compared to the already very time-consuming local analysis limits the usefulness of the current implementation. The null hypothesis H_0^5 cannot be rejected for "KL". For analysis module "LP_GRB" the null hypothesis H_0^5 can be rejected in favor of the alternative hypothesis H_A^5 ("The use of multiple CPU cores reduces the runtimes of multi-core-enabled global analysis modules").

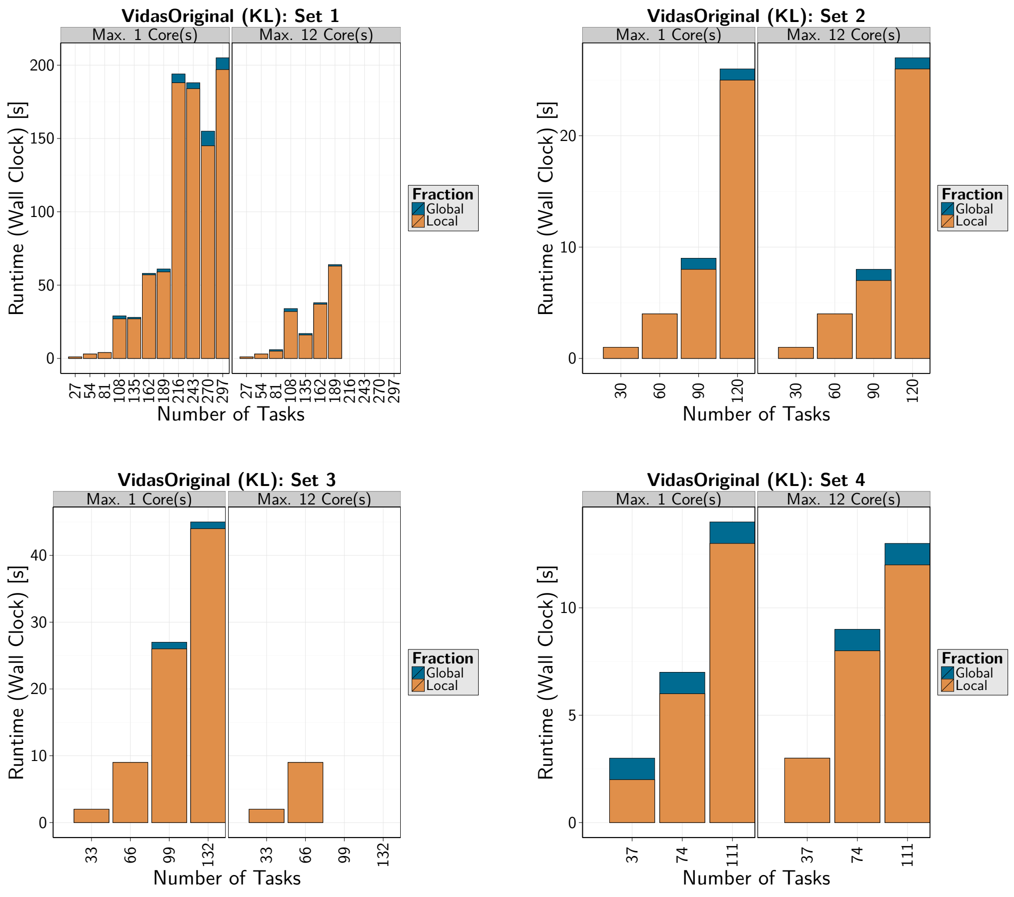


Figure 6.27. – Benchmark “ViDAsOriginal”: Multi-core vs. Single-Core, (“KL”)

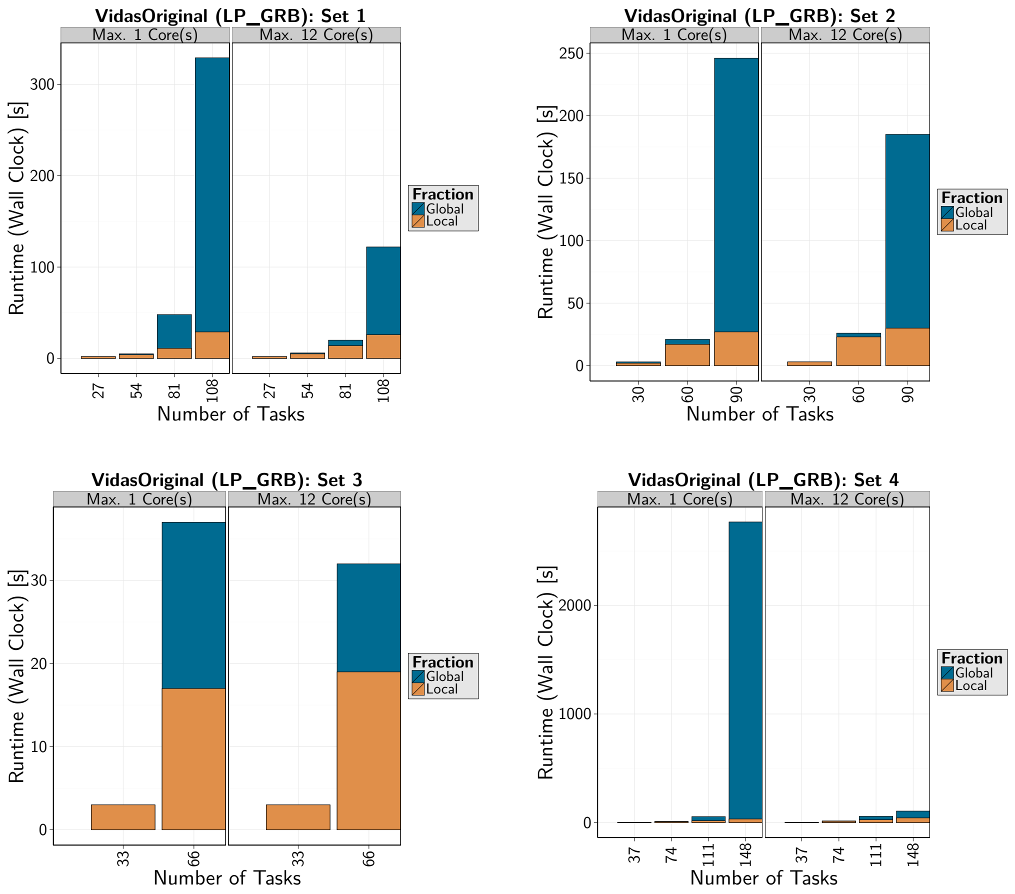


Figure 6.28. – Benchmark "ViDAsOriginal": Multi-core vs. Single-Core ("LP_GRB")

Except for the model with 60 tasks contained in Set 2 the use of multiple cores was beneficial for the module “LP_CPX”, too. For analysis module “LP_CPX” the null hypothesis H_0^5 can be rejected in favor of the alternative hypothesis H_A^5 (“The use of multiple CPU cores reduces the runtimes of multi-core-enabled global analysis modules”).

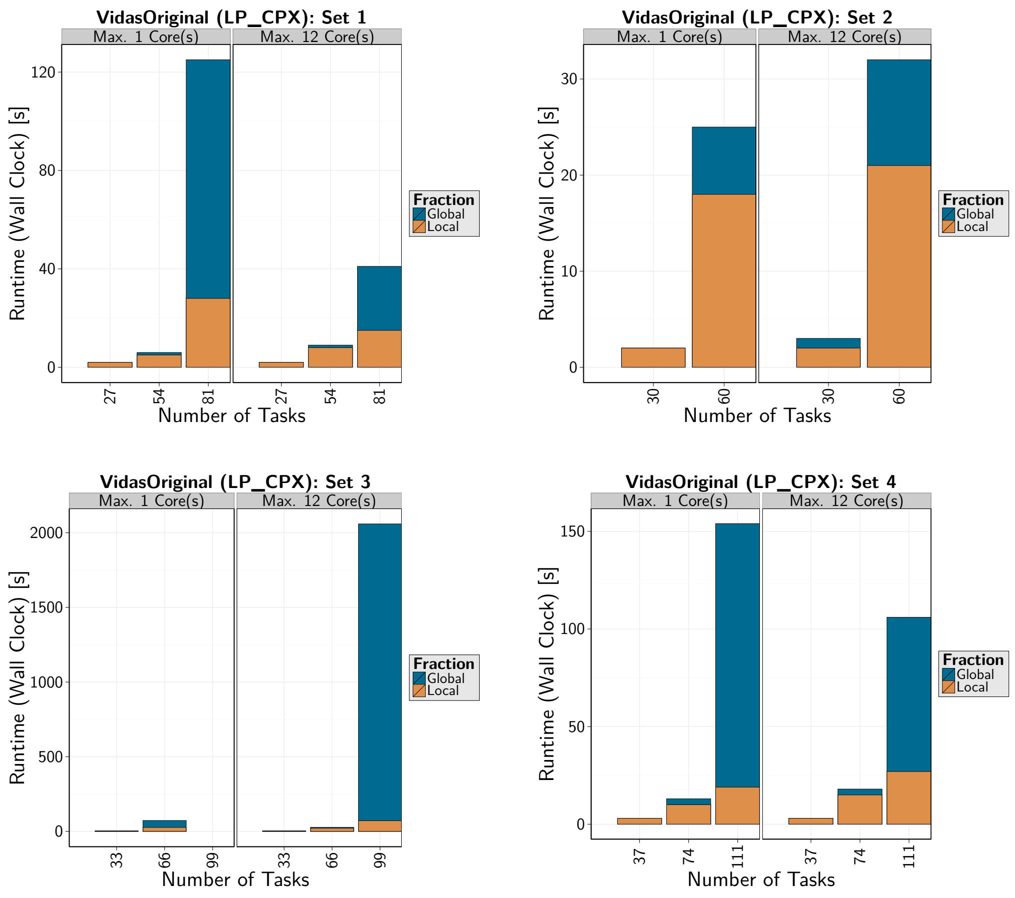


Figure 6.29. – Benchmark “ViDAsOriginal”: Multi-core vs. Single-Core (“LP_CPX”)

ViDAsExpensive The situation for the benchmark “VidasExpensive” regarding multi-threading is by far the most complicated one in this thesis. Even the powerful “KL” heuristic could not find solutions for any model larger than the simplest one contained in Set 4, as shown in Figure 6.30. In two cases the use of multiple cores was beneficial: for the model with 81 tasks in Set 1 and for model with 180 tasks contained in Set 2. For all other cases no effect was visible or the analysis even failed to find solutions when using more than one core. Therefore the null hypothesis H_0^5 cannot be rejected for “KL”.

Figure 6.31 shows again that except for the model with 54 tasks in Set 1 the global analysis module “LP_GRB'' is quite graceful with respect to single core vs. multi core.

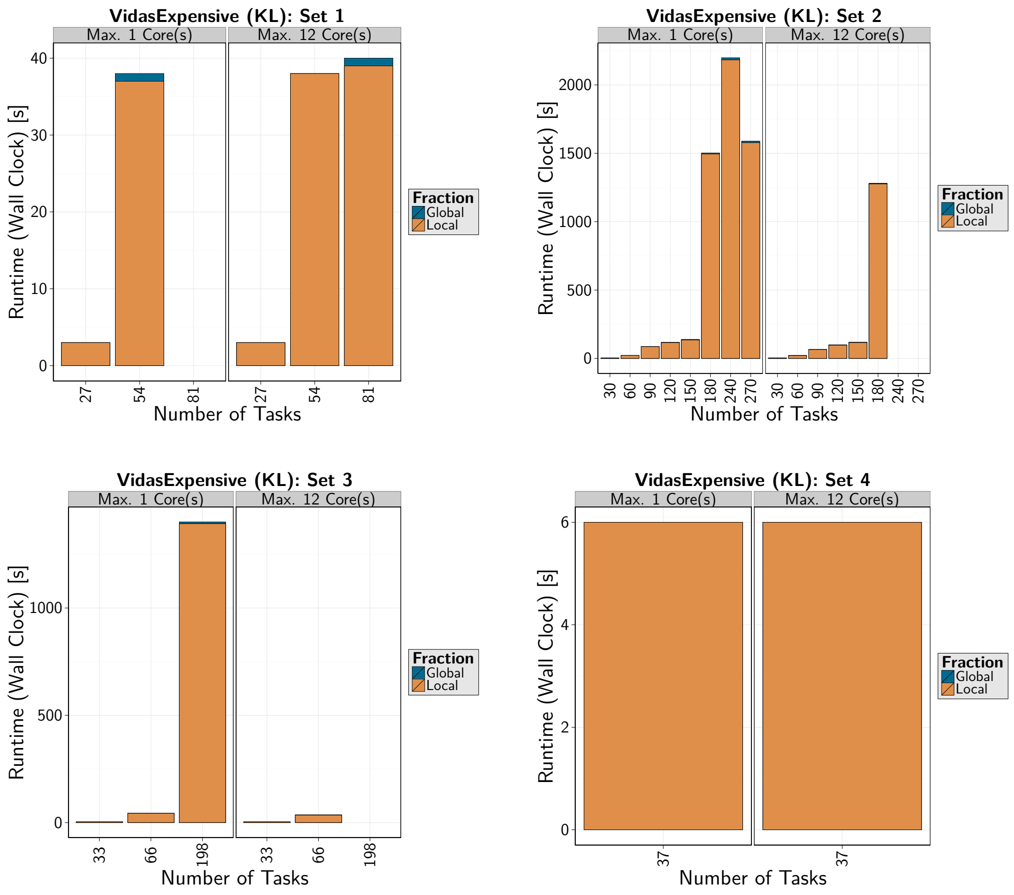


Figure 6.30. – Benchmark “ViDAsExpensive”: Multi-core vs. Single-Core (“KL”)

But even with 12 available cores no solutions could be found for larger models. The small sample size does not allow the rejection of null hypothesis H_0^5 for “LP_GRB”.

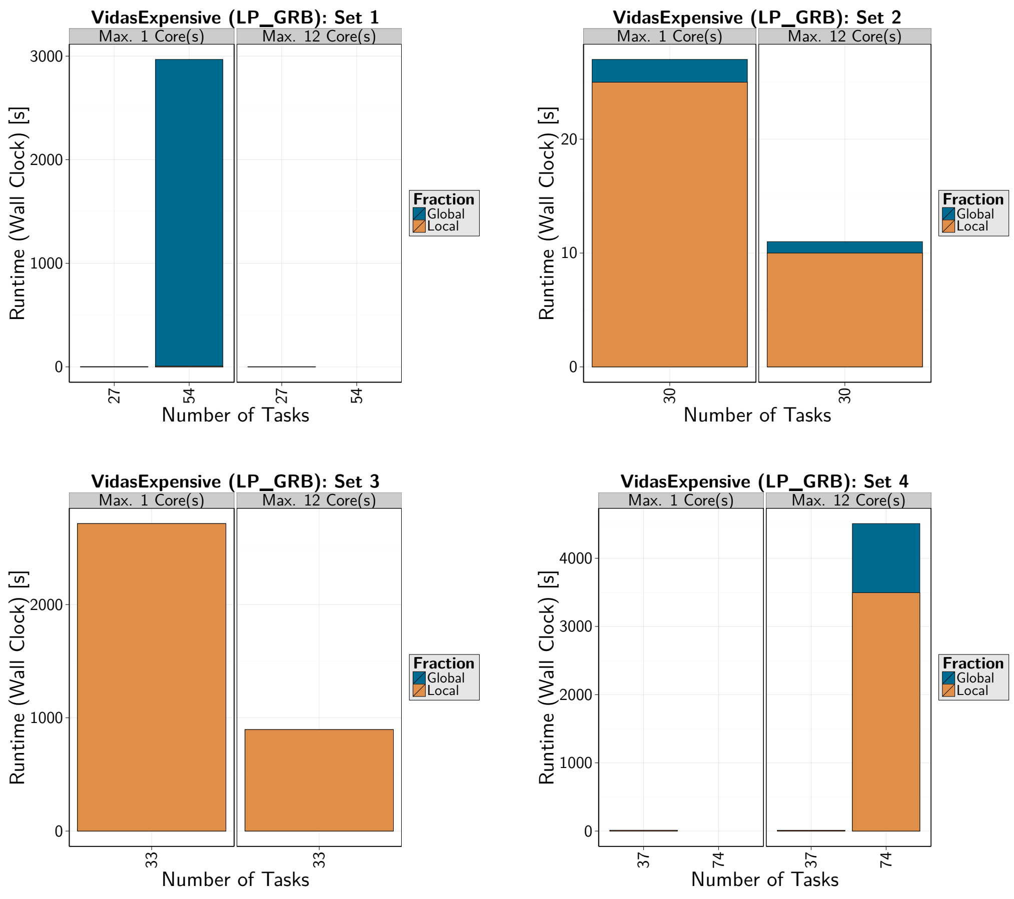


Figure 6.31. – Benchmark “ViDAsExpensive”: Multi-core vs. Single-Core (“LP_GRB”)

The situation is similar for “LP_CPX” as can be seen in Figure 6.32. Note that there is no figure for benchmark “ViDAsExpensive”, Set 3 for the global analysis module “LP_CPX” simply because there is no model in that set for which that analysis module found a solution for both analysis runs (with 12 cores and with one core). Therefore, no comparable results exist for that module in Set 3. While the results look promising when looking at the fact that in the few comparable cases the multi-core runs have been faster than the single-core runs, the small sample size does not allow the rejection of null hypothesis H_0^5 for “LP_CPX”.

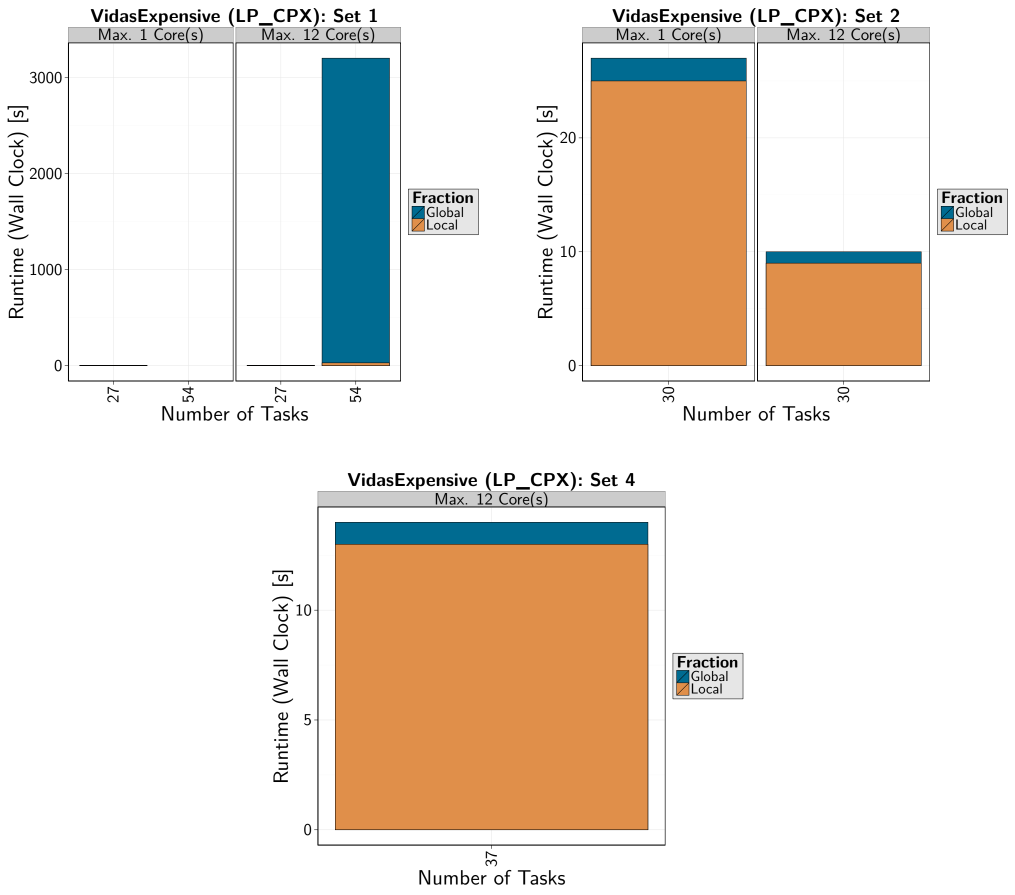


Figure 6.32. – Benchmark “ViDAsExpensive”: Multi-core vs. Single-Core (“LP_CPX”)

6.6. Summary

6.6.1. Summary: Global Analysis

In Table 6.42 some statistics of all analyzed global models are given. In total 3866 global analysis runs have been performed on 540 different models. The "KL" approach was the most successful global analysis module with 719 successfully optimized models. The least successful analysis method was "MZ". This was, however, expected as "MZ" is an exact optimizing method designed to find optimal solutions for DSE problems which inevitably leads to very long runtimes for NP-hard problems. The "LP_∗" methods using commercial backends ("LP_GRB" and "LP_CPX") come close to the results of "KL". The global analysis module "LP_GLP" using the open-source backend GLPK cannot compete with them, but still provides an impressive performance given the fact that it is freely available and developed and maintained by only one programmer.

Module	**Successful**	**Timeout**	**Incomplete**	**Error**	**Total**
KL	719	363	78	0	**1160**
LP_GRB	644	445	71	0	**1160**
LP_CPX	633	446	81	0	**1160**
LP_GLP	22	76	0	0	**98**
MZ	10	0	278	0	**288**
Total	**2028**	**1330**	**508**	**0**	**3866**

Table 6.42. – Global Analysis: General Information

Benchmark / Method	**"LP_GRB"**	**"LP_CPX"**	**"LP_GLP"**
TindellScaled	✓	✓	✓
Generated	✓	✓	
ViDAsOriginal	✓	✓	
ViDAsExpensive	–	–	
Null Hypotheses Rejected	**3 / 4**	**3 / 4**	1 / 4

Table 6.43. – Summary: Null Hypothesis H_0^{1a} ("Cost greater than "KL"")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

The results for the global null hypotheses H_0^{1a} and H_0^{1b} found for all benchmarks are given in Tables 6.43 and 6.44. For "LP_GRB" and "LP_CPX" in three of four cases the null hypotheses could be rejected. The hardware costs of the solutions found by these analysis modules have been equal (in most cases) or even better (in a few cases) than the costs of the solutions found by "KL". For "LP_GLP" the null hypotheses could be rejected only for one of the benchmarks.

With respect to the runtime of the "LP_∗" methods, the picture looks different. Table 6.45 shows that only for the analysis module "LP_GRB" used on the benchmark "Generated" the measured runtimes have been equal or less than the one of "KL". This

Benchmark / Method	"LP_GRB"	"LP_CPX"	"LP_GLP"
TindellScaled	✓	✓	✓
Generated	✓	✓	
ViDAsOriginal	✓	✓	
ViDAsExpensive	–	–	
Null Hypotheses Rejected	**3 / 4**	**3 / 4**	**1 / 4**

Table 6.44. – Summary: Null Hypothesis H_0^{1b} ("Cost not similar to "KL"")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

Benchmark / Method	"LP_GRB"	"LP_CPX"	"LP_GLP"
TindellScaled	–	–	–
Generated	✓	–	
ViDAsOriginal	–	–	
ViDAsExpensive	–	–	
Null Hypotheses Rejected	**1 / 4**	**0 / 4**	**0 / 4**

Table 6.45. – Summary: Null Hypothesis H_0^2 ("Runtime of "KL" is equal to or less")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

also has been expected because (similar to the "MZ" analysis module) the "LP_∗" approach is based on optimal optimization techniques which can hardly compete with heuristics when comparing the runtimes.

The result for the global analysis is that in terms of performance and scalability the approach presented in this thesis cannot compete with the "KL" approach. In terms of the quality of the calculated solutions the global analysis approach presented in this thesis in many cases can provide similar results. But there is still potential to further improve the approach to increase the quality especially when it comes to backtracking whenever one or more tasks could not be allocated by the local analysis (this has been observed while running the evaluation).

6.6.2. Summary: Local Analysis

A huge amount of data has been measured for the local analysis modules "Eis" and "Stm". In total 122932 local analysis runs have been analyzed with 29367 models (however it cannot be guaranteed that each analyzed model is unique). Both analysis modules successfully analyzed most of the local DSE problems, see Table 6.46.

Tables 6.47 and 6.48 aggregate the results of analyzing Hypothesis 3 and 4. Both corresponding null hypotheses have been rejected for all benchmarks. Therefore, the Spare-Time/MaxWCET approach "Stm" has been shown to calculate results with a significantly reduced runtime (up to factor 10) and hardware costs which are almost always equal to the costs of solutions calculated by the "Eis" local analysis module.

Module	**Successful**	**Error**	**Total**
Eis	60187	1279	**61466**
Stm	61226	240	**61466**
Total	**121413**	**1519**	**122932**

Table 6.46. – Local Analysis: General Information

Benchmark / Method	**"Eis"**
TindellScaled	✓
Generated	✓
ViDAsOriginal	✓
ViDAsExpensive	✓
Null Hypotheses Rejected	**4 / 4**

Table 6.47. – Summary: Null Hypotheses H_0^3 ("Cost larger than "Eis"")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

Benchmark / Method	**"Eis"**
TindellScaled	✓
Generated	✓
ViDAsOriginal	✓
ViDAsExpensive	✓
Null Hypotheses Rejected	**4 / 4**

Table 6.48. – Summary: Null Hypotheses H_0^4: ("Runtime of "Eis" is equal to or less")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

6.6.3. Summary: Multi-Core vs. Single Core

As all global analysis runs (and the corresponding local analyses runs) have been executed twice — the first time with maximal 12 CPU cores, the second time with maximal one CPU core — a large amount of data has been recorded. The analysis of Hypothesis 5 shows that "KL" did not experience a significant speed-up for any of the benchmarks when the local analysis module it executed had access to more than one CPU core. For "LP_GRB" and "LP_CPX" a significant speed-up could be shown for two of three benchmarks respectively.

Benchmark / Method	**"KL"**	**"LP_GRB"**	**"LP_CPX"**	**"LP_GLP"**
TindellScaled	–	✓	✓	–
Generated	–	–	–	
ViDAsOriginal	–	✓	✓	
ViDAsExpensive	–	–	–	
Null Hypotheses Rejected	**0 / 4**	**2 / 4**	**2 / 4**	**0 / 4**

Table 6.49. – Summary: Null Hypothesis H_0^5 ("Use of multiple CPU cores does not reduce runtimes of multi-core enabled global analysis modules")
(✓) Null Hypothesis Rejected, (–) Null Hypothesis not Rejected, (Empty) Not Tested

7. Conclusion

7.1. Summary

In this thesis, an optimization method for extending a safety-critical embedded real-time system with additional functionalities implemented as software tasks while minimizing the cost arising from modifications to the hardware architecture is presented. The approach consists of a two-tier optimization which exploits the hierarchical structure of the hardware architectures typically used in embedded systems by iteratively performing coarse-grained optimization steps on the system level called global analysis followed by fine-grained analysis steps separately for each hardware subsystem called local analysis. For both the global and the local analysis exact optimization methods are presented based on mixed integer linear programming.

For a given DSE problem, the global analysis assigns ECU-types to ECUs such that the total hardware architecture cost are minimal, groups for each hardware subsystem all ECUs of the same ECU-type to so-called ECU-types groups, finds a pre-allocation of all unallocated software tasks onto those groups using the notion of utilization to ensure that enough capacity is available, and synthesizes local deadlines where necessary. The proposed algorithm is based on mixed integer linear programming.

Then separately for each hardware subsystem the local analysis is executed for the calculated pre-allocation and with a limit for the hardware cost of that subsystem. The local analysis approach presented in this work introduces the concepts of spare-times and MaxWCET for characterizing the available computation capacity on FPS-scheduled ECUs and CAN buses with strict periodic tasks/messages without release jitter such that the allocation problem is reduced to a variant of the bin-packing problem without the need to perform a schedulability analysis for each ECU. Tasks which have been pre-allocated to a subsystem but could not be allocated during a local analysis run are returned to the global analysis as part of a so-called odd set. The global analysis is then responsible for pre-allocating those tasks again to other subsystems or to the same subsystem but with an increased cost limit. The proposed Spare-Time/MaxWCET is not only capable of guaranteeing that the hardware cost limit is not exceeded, but also of minimizing the required hardware cost as part of the optimization objective. This feature can further reduce the total hardware cost in situations where the cost limit specified by the global analysis is too lax.

An extensive evaluation of the concepts presented in this work has been carried out where the proposed approach is compared to multiple alternative approaches. The results show that the MILP-based global analysis could in many cases provide solutions of similar quality (measured by the total hardware cost) compared to the alternative heuristic approach based on the Kernighan-Lin algorithm. But (as expected) the runtimes of the exact MILP-based approach are huge compared to the heuristic "KL" algorithm. While small examples could successful be solved with the open-source MILP-backend GLPK, intermediate-size examples could only be solved with the (expensive) commercial MILP-solvers. The "KL" algorithm proved to be much faster in general than the MILP-based approach but could also solve "only" approximately 62% of the benchmark models without hitting the time limit compared to 55% successfully solved benchmark models using the MILP-based approach (with the commercial MILP-backend "Gurobi"). The comparison of the heuristic "KL" approach with the exact MILP-based approach in this work provides the first quantitative confirmed statements about the "KL" approach regarding the quality of the solutions (measured in hardware cost) and the runtime of the analysis for a sufficiently large set of small, medium-sized and large benchmark models. Therefore, the results presented in this thesis provide valuable data useful for extending and improving the heuristic global analysis approach in the future with the ability to perform further evaluation runs on demand. Additionally, the MILP-based approach has the advantage of being easily expandable thus boosting the quick implementation of new features. However, in a productive environment the "KL" global analysis should be favored over the MILP-based approach proposed in this thesis to avoid huge runtimes.

The evaluation clearly shows that the proposed Spare-Time/MaxWCET approach for local analysis has significantly better runtimes (it is 1.5 and 10 times faster) than the alternative approach while guaranteeing nearly identical quality (measured by subsystem hardware cost). The effect of slightly more expensive solutions (greater hardware costs) due to the abstraction used in the Spare-Time/MaxWCET approach was significantly smaller than expected: During the evaluation only a very small number of cases have been observed where the hardware costs of the solution calculated by the Spare-Time/MaxWCET approach has been greater than the costs of the solution found by the alternative approach. As for the global analysis approach, using a commercial MILP-solver as backend is highly recommended, at least for larger instances. During the evaluation, a significant lower variance of the measured analysis runtimes has been observed for the Spare-Time/MaxWCET approach compared to the alternative approach. The drawback of the proposed approach is that release jitter cannot be handled (because tasks and messages could be allocated independently anymore). But release jitter is not supported by the alternative approaches as well, for the same reason.

The evaluation also showed that the built-in automatic multi-threading support of the commercial MILP-solvers was not beneficial for the design space exploration problem defined in this thesis. Explicit measures for parallelizing the optimization process could overcome this, as implicitly enabled by the design of the proposed two-tier optimization approach where the decoupled local analysis steps can be run simultaneously.

Finally, the comparison of the presented two-tier optimization approach with an exact optimization approach for the whole system ("DSEOverallAnalysisMainz") shows that the two-tier approach is able to solve significant larger DSE problem instances. This comes at the cost that despite both the global and the local analysis approaches are based on exact algorithms, due to the way both are combined in this work the calculated solutions are not guaranteed to be optimal. The proposed approach is a trade-off between a pure exact algorithm and a pure heuristic algorithm. The modular design based on two tiers — the global and the local tier — has a good potential to fit for academic DSE problem as well as for industrial-sized problems because different modules can be combined as required for the particular model size.

7.2. Discussion of Related Work

Most publications related to this work describe optimization approaches focusing either on other levels of abstraction (for example the optimization of digital hardware components) or solve only a subset of the problems incorporated in the approach presented in this thesis. The tools and case studies presented in those publications are in general not publicly available which makes the comparison difficult. Even in cases where the full benchmark models are described detailed enough, the quickly evolving computer hardware market makes any measurements of e.g. runtime taken more than a few years ago obsolete.

Therefore the discussion of related work in the next subsections compares only the provided features of the approaches and tools as presented in those publications. The individual publications are classified into four categories: publications which describe general approaches for design space exploration without explicitly targeting safety-critical embedded real-time systems (see Section 7.2.1), publications which solve the problem of allocating task networks to hardware architectures (see Section 7.2.2), publications which focus on the design space exploration on the hardware level (see Section 7.2.3), and finally publications presenting approaches for design space exploration which combine the allocation problem with the hardware modification/extension problem (see Section 7.2.4, and thus are very similar to the approach presented here.

7.2.1. General Frameworks for Design Space Exploration

The main contributions of [Kue06] are a new method for performance evaluation of embedded systems, a user-controlled evolutionary algorithm for performing design space exploration and a corresponding implementation named EXPO. The evolutionary approach is demonstrated using the example of a hardware architecture consisting of a communication bus and one or more packet processors to which a given software application shall be allocated. While the proposed evolutionary approach is general enough to be used not only for calculating optimal allocations of the software but also for extending and optimizing the hardware architecture itself, this is not done as part of thesis. That approach is solely based on heuristic methods, in contrary to the concepts presented in this PhD thesis.

A design space exploration method called Pareto-Front Arithmetics is proposed in [HT03]. The method is explicitly designed for being applied hierarchically, similar to the approach presented in this thesis. An extended presentation of this work has been published in [Hau+03]. The case study presented in both publication consists of an MPEG4 encoder to be realized with optimal cost, power consumption and flexibility of the design. For the case study, a fixed hardware architecture has been used, but it should be possible to extend the approach to support more complex hardware architectural patterns. In contrary to this thesis the approach is based solely on a (heuristic) evolutionary optimization algorithm.

The previous publications refer to [ARS00] where the idea of exploiting the hierarchical structure of large systems is promoted and a Pareto optimization approach is defined which can efficiently handle such systems. The approach is applied to perform a design space exploration for finding optimal configuration parameters for an embedded hardware system for example for the used memory hierarchy. Multiple modules ("Walkers") are specified which evaluate special aspects of the hardware architecture. But there is no explicit support for explicit hard real-time systems as in this thesis. However, the paper supports the presumption of this thesis that it is beneficial to exploit the hierarchical structures found in (larger) embedded system.

7.2.2. Restriction to Allocation Problem

The following publications have in common that they solve the problem of allocating a given software application to a given hardware architecture often combined with finding additional parameters for configuring the ECUs (e.g. settings for the schedulers) and buses. In all publications the hardware architecture is not modified.

An early publication on the allocation of a task network of hard real-time tasks onto a fixed hardware architecture such that all tasks are schedulable has been published in [Tin96]. Their example task network consists of 43 tasks, the hardware architecture is composed of eight ECUs connected by a token-ring bus. The proposed algorithm is based on simulated annealing. This publication is especially interesting for this thesis because it is one of the rare papers where the complete model used for the case study is unveiled. Consequently, this example has been used in this thesis as a basis for creating most of the benchmarks (see Chapter 6 on page 121). In the paper, modifications to the hardware architecture are not allowed, and it is assumed that all ECUs have the same ECU-type.

In [HRE06] a design space exploration process based on a genetic algorithm is described. The approach calculates Pareto-optimal solutions for an arbitrary number of system configuration parameters such as task priorities, power consumption, buffer sizes, etc. The underlying hardware architecture is not subject to change.

In [Pop+04], Pop et al. proposed a very promising approach with many commonalities to this thesis. Their approach aims for extended an existing software application with new functionalities. The hardware architecture remains unchanged, but the existing application can be modified inducing modification cost (for example because tasks have to be reimplemented). The approach aims for allocating all new tasks while minimizing those modification cost. The proposed algorithm is a heuristic method.

The same authors describe in [Pop+06; Pop+08] a strategy for design optimization capable of allocating software tasks to processors of a given hardware architecture, choosing/modifying their scheduling configurations and scheduling policies (FPS and EDF), and determining valid bus configurations. The approach aims for calculating feasible solutions but leaves the hardware architecture untouched.

In [PEP06; Erb06] the authors present the framework SESAME for system-level performance evaluation based on Kahn process networks. He uses that framework to perform a multi-objective design space exploration for finding optimal allocations of the application software to a fixed hardware architecture. As case study a Motion-JPEG encoder application is used which does not have hard deadlines. Some extensions towards real-time systems with hard timing requirements are proposed but still based on a simulating annealing heuristic and without modifying the target hardware.

Another approach for solving allocation problems has been presented in [Zhu+10]. Based on the Metropolis framework a combination of linear programming and heuristic algorithms is used to initially allocate tasks, assign signals to messages, allocate the message to buses and choose appropriate priorities for tasks and messages. The tasks might then be re-allocated initiating another iteration of the optimization process. The hardware architecture remains unmodified in the whole process.

The earlier publication [Zen+06] is also based on the Metropolis framework. It provides lots of details of the factors to consider during design space exploration in the automotive domain — mainly concerning CAN buses — but does not explain how to actually perform the design space exploration in a systematic way.

An approach which is similar to our global analysis has been proposed in [AS00]. The authors describe an algorithm which combines tasks into task clusters and processors into processor clusters using a k-way cut heuristic. Their approach assumes that there is only one processor type. It does not minimize the number of processors or the hardware architecture cost. Tasks are grouped by their period (assuming that the number of different periods is small for most embedded systems). The communication load between task clusters is minimized. The heuristic is approximating.

In [EB10] the authors discuss algorithms for initial and maintenance task allocation and evaluate those using generated scenarios. Their approach is based on simulated annealing and does not modify the hardware architecture.

7.2.3. Design Space Exploration for Hardware Design

The concept of DSE is also used on other levels of abstraction, for example on the hardware level where design space exploration is applied to determine good architectural parameters while designing new hardware components (processors, system-on-chips, etc.).

In [GR00] a design space exploration approach is presented for optimizing digital circuit designs on a high abstraction level. The result of the optimization process is then used as input for high-level synthesis. Optimization objectives are the area required for the digital circuits and their power consumption which clearly differs from the goals of this thesis.

A newer project is the MULTICUBE project, aiming for "an efficient and automatic exploration of parallel embedded architectures in terms of several design parameters such as available parallelism (e.g. number of cores, processor issue width), cache-subsystem (e.g. cache size and associativity) and Network-on-Chip (NoC) related parameters (e.g., channel buffer size)", see [Sil+11]. Similar to the approach presented in this thesis the user (namely an "exploration architect" together with a "use case and simulator provider") drives the optimization process by providing appropriate DSE strategies and parameters, assessing the (intermediate) results and reconfiguring the DSE process until a good solution has been found. The project focuses on the hardware level and specifically on many-core architectures where lots of different parameters are considered during the optimization phase. The goals for that project are quite different compared to this thesis which does not aim for designing new hardware components but on extending large distributed systems by integrating additional components as required to allocate new software tasks.

7.2.4. DSE of Allocation and Hardware Architecture

The publications mentioned in this section solve the optimization problem of allocating a software application onto a hardware architecture while simultaneously exploring the hardware design space described by a specified set of allowed modifications to the hardware architecture.

The author of [Dor+08] present an exact branch-and-bound search algorithm for finding allocations of tasks onto a set of identical ECUs using the minimal number of required ECU. The approach does only handle periodic tasks without considering communication between tasks. The possibilities for defining the hardware design space are quite limited compared to the approach presented in this thesis.

In [Mad+07] the authors distinguish between the allocation of software tasks to a fixed hardware architecture and the allocation to a flexible hardware architecture where the types and/or numbers of ECUs can be modified. They address both cases using formalisms for describing task networks and hardware architectures which are close to the ones proposed in this thesis. A genetic algorithm based on the PISA framework [Ble+03] is used for the design space exploration process which can modify the types of ECUs and buses, add/remove ECUs to/from the hardware architecture, and (re-)allocate tasks which includes finding static schedules by utilizing a basic *list scheduling algorithm*. In contrary to that approach, the approach proposed in this thesis distinguishes between a global and a local optimization tier which allows the use of exact and more detailed optimization methods (with support for FPS scheduled ECUs and a subsystem-local CAN bus) on the local tier (see Chapter 5 on page 85). On the other hand a strength of the approach presented in [Mad+07] is the support for multi-objective optimization based on the PISA framework. Due to the differences between that approach and the one proposed in this thesis it is not possible to predict which of the approaches would perform better for a given problem instance (assuming that such an instance is fully supported by both approaches).

7.3. Future Work

This section names some of the remaining open questions requiring further investigation.

Firstly, some limitations have been described for the proposed optimization algorithms of which at least some could be overcome in the future. These limitations include the focus on strictly periodic tasks where support for more complex activation patterns should be considered. Another limitation is the lack of support for release jitter. Release jitter could potentially be handled as part of the local analysis approach by calculating over-approximations of the (input) jitter for each task and each signal/message and considering this data for the Spare-Time/MaxWCET pre-analysis. The focus on FPS-scheduled ECUs can be expanded to other scheduling methods for ECUs including time-triggered and hierarchical scheduling methods. The already existing heuristic for calculating feasible schedules for the global bus could be incorporated in the MILP-based global analysis. Alternatively an existing MILP-based approach for schedule synthesis such as the one presented in [Zen+11] could be added to the global analysis formulation.

Secondly, a few technical improvements may help to extend the scalability of the analysis tools. This includes a re-implementation of the Zerg modules used for Spare-Time/MaxWCET local analysis, which is work in progress already. The re-implementation will be based on the Coin-OR Osi abstraction layer (as is the global analysis module) and will thus make the analysis module more independent from specific MILP-backends. The abstraction layer itself could be extended to allow to interrupt the optimization phase at specific states, e.g. whenever an integral solution has been found. This would help to decrease the excessive runtimes for large models, of course at the cost of the quality of the found solutions.

Finally, the methods presented in this thesis can be tested and further extended and improved in industrial-focused research projects. This is scheduled already for the project SPES_XT (see [Böh] for the project's website).

A. Mathematical Foundations

A.1. Fixed-point equations

This section presents some of the basic mathematical foundations used in this paper as invented by Alfred Tarski in [Tar55]. Note that this section does not contain any new findings but is included as a help for the reader to better understand the underlying concepts.

Definition A.1 (Lattice)
A lattice is a partially ordered set where every two elements of this set have a supremum and an infimum. A lattice $\langle A, \leq \rangle$ is called complete if for every subset $B \subseteq A$ a supremum and infimum exists. □

Remark A.2 (Natural Numbers Lattice)
The natural numbers with the usual partial order relation $\leq$ constitute a lattice which is not complete. □

Definition A.3 (Fixed-point)
For any function f a value x out of the domain of f is a fixed-point if and only if $f(x) = x$. □

There may exist multiple fixed-points for a given function. Especially for real-time schedulability we are interested in the lowest fixed-point.

Definition A.4 (Lowest (Greatest) Fixed-point)
Let L be a lattice with ordering function $\leq$ and $f : L \to L$ be a function with at least one fixed-point. Let F_f be the set of all fixed-points of function f. Obviously, the lowest value in F_f (according to $\leq$) is the ***lowest fixed-point of*** *f and the largest value is the* ***largest fixed-point of*** *f.* □

Theorem A.5 (Lowest Fixed-point of a monotonically increasing function)
Let $\mathcal{L} = \langle A, \leq \rangle$ be a lattice with lowest element $0 \in A$. Let $f : A \to A$ be a monotonically increasing function with at least one fixed-point in A. The lowest fixed-point l_{min} of f is then $l_{min} = lim_{n\to\infty} f^n(0)$.

Proof There is no fixed-point $l = lim_{n\to\infty} f^n(0)$ with $l < l_{min}$, because l_{min} is already the lowest fixed-point of f.

l also cannot be larger than the lowest fixed-point because f is monotonically increasing. Therefore for every $a, b \in A$ with $a \leq b$ it holds $f(a) \leq f(b)$. With $0 \leq l_{min}$ for every $n \in \mathbb{N}$ it holds that $f^n(0) \leq f^n(l_{min}) = l_{min}$.

It follows that $l_{min} = lim_{n\to\infty} f^n(0)$. ■

B. Examples

B.1. MILP Example encoded with MathProg

The following MathProg model implements the example linear program found in Section 2.3.1.2 on page 23.

Listing B.1 – MILP Example Model

```
# For testing this example please install the GNU Linear Programming Kit (
    http://www.gnu.org/software/glpk/)
# Run: "glpsol --math production.mod"

# Two constants for specifying the price of product type 1 and product type
    2, respectively.
param price1 := 10000;
param price2 := 15000;

# Two constants for specifying the production time in hours for each product
    type
param time1 := 23;
param time2 := 42;

# Two free integer variables representing the number of products to produce
    for each product type
var x1, integer, >=0;
var x2, integer, >=0;

# Constraint: Only maximal 200 hours of production time are available per
    month
s.t. maxTime: x1*time1 + x2*time2 <= 200;

# Two constraint specifying that at leats one product of each type has to be
    produced
s.t. minOneProduct1: x1 >= 1;
s.t. minOneProduct2: x2 >= 1;

# The optimization objective is to maximize the total profit
maximize maxprice: x1*price1 + x2*price2;

# Start solving the problem
solve;

# The next lines print the result (if any)
printf "Number of products of type 1: %d\n", x1;
printf "Number of products of type 2: %d\n", x2;
printf "Profit per month: %d\n", x1*price1+x2*price2;
```

C. Hardware/Software Configuration for Evaluation

The following tables provide details on the hardware (Table C.1) and software (Table C.2) used for performing the evaluation described in Chapter 6.

Processor Model	AMD Opteron™ Processor 6282 SE
Number of Processors	4
Cores per Processor	16
Total number of Cores	64
Memory	512 GB

Table C.1. – Evaluation Server: Hardware Configuration

Tool/Library	**Version**	**License**	**Vendor**
libc	2.11	LGPL	Open Source
Qt	4.6.3	LGPL	Open Source
GNU Scientific Library	1.14	GPL	Open Source
GNU Linear Programming Kit	4.43	GPL v3	Open Source
HySAT	HySAT-0.8.5-reentrant	Proprietary	University of Oldenburg
Coin-OR Osi	1.6.0	EPL v1.0	Open Source
Gurobi Optimizer	5.0.0	Proprietary	Gurobi Optimization, Inc.
IBM ILOG CPLEX Optimization Studio V12.4	12.4	Proprietary	IBM

Table C.2. – Evaluation Server: Software Configuration

D. Additional Benchmark Information

In this section of the appendix that part of the data measured during the evaluation is presented which is considered too comprehensive to be printed in Chapter 6 directly.

D.1. Benchmark "TindellScaled"

This section

Figure D.1 shows the hardware architecture of the smallest models contained in the benchmark. The corresponding task network is depicted in Figure D.2. Details on the properties of the available ECU-types are given in Table D.1.

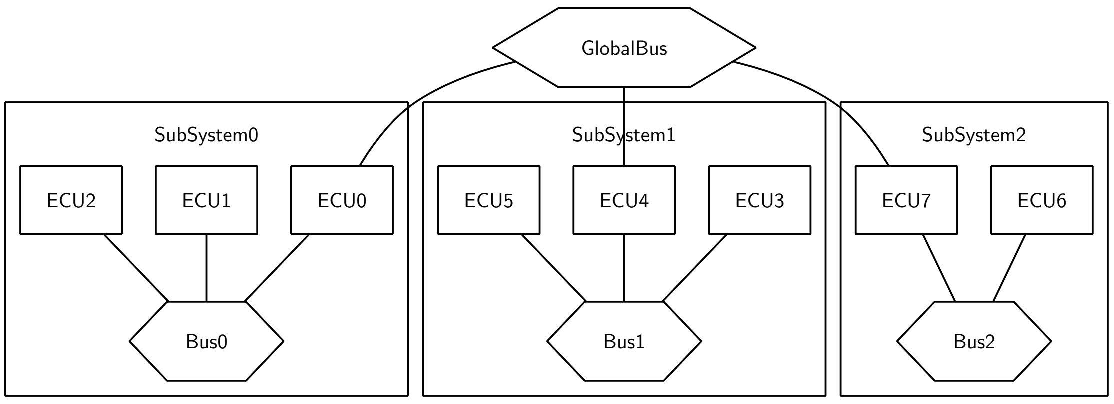


Figure D.1. – Benchmark "TindellScaled": Schematic Drawing Hardware of the Architecture of the Smallest Model

Name	Cost	Memory [kByte]
ECUType0	19	10000
ECUType1	37	12000
ECUType2	11	7000
ECUType3	21	10000

Table D.1. – Benchmark "TindellScaled": ECU Types

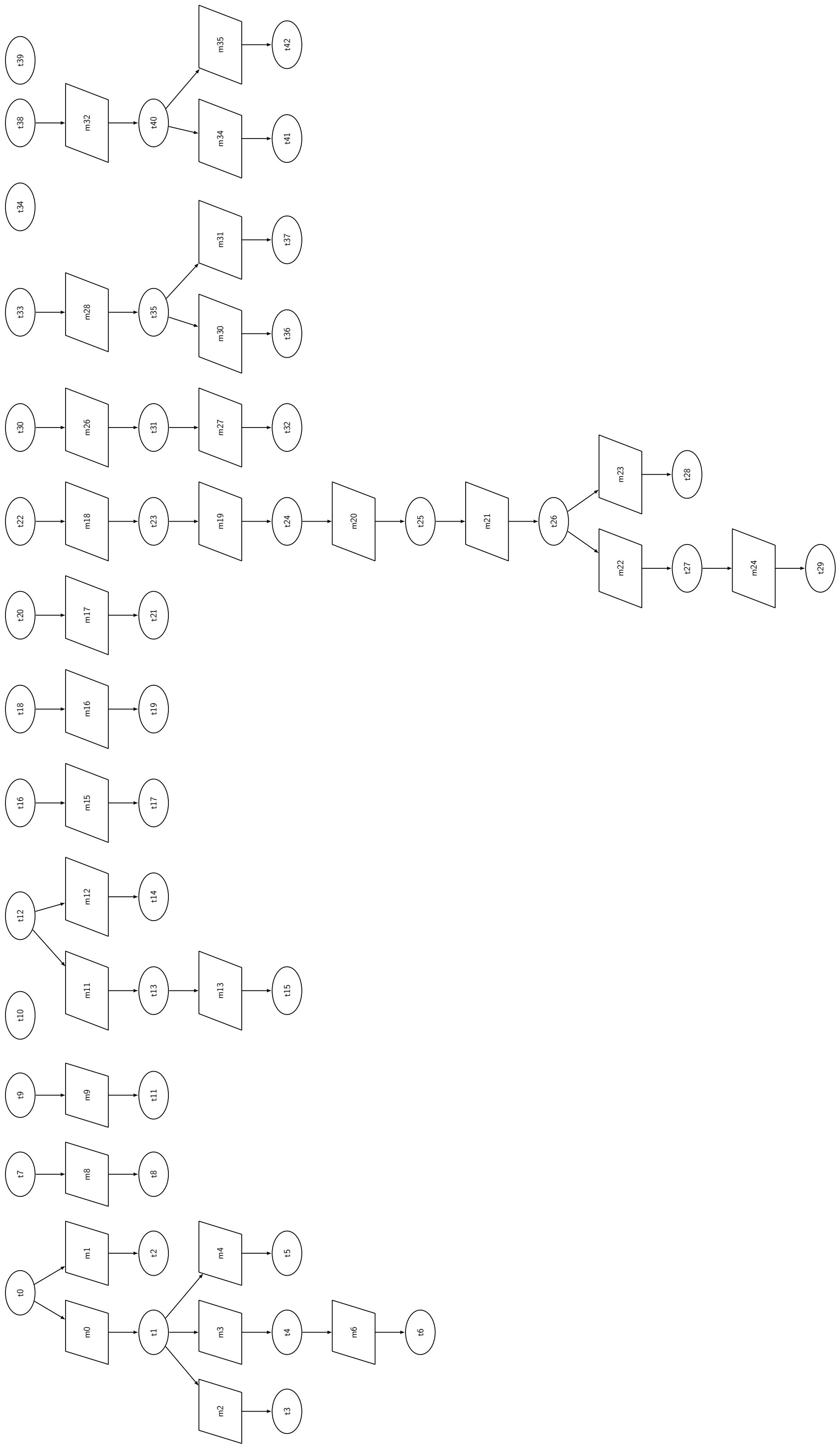


Figure D.2. – Benchmark "TindellScaled": Schematic Drawing of the Task Network of the Smallest Model

In Table D.2 the models contained in this benchmark are classified according to the total number of tasks and the fraction of unallocated tasks. For some combinations there are no models contained in the benchmark.

#Tasks	x **Unallocated Tasks [%]**		
	$x \approx 14$	$x \approx 20$	$x \approx 37$
43	✓	✓	✓
86	✓	✓	✓
129	✓	✓	✓
172	✓	✓	✓
215	✓	✓	✓
258	✓	✓	✓
301	✓	✓	✓
344	–	–	✓
387	–	–	✓
430	–	–	✓
473	–	–	✓
516	–	–	✓
559	–	–	✓
602	–	–	✓
645	–	–	✓
688	–	–	✓
731	–	–	✓
774	–	–	✓
817	–	–	✓
860	–	–	✓

Table D.2. – Benchmark “TindellScaled”: Combinations of Parameters in the Benchmark

D.1.1. Results of Benchmark “TindellScaled”

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
TindellScaled_1, Set 1, 43 Tasks, 6 Unallocated Tasks (14%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	6	34	0/2	0/2	125
LP_GRB	✓	5	35	0/1	0/1	125
LP_CPX	✓	6	35	0/1	0/1	125
LP_GLP	✓	5	35	1/2	1/2	125
MZ	✓	0	16	11/11	9/9	125
TindellScaled_1, Set 1, 43 Tasks, 6 Unallocated Tasks (14%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	6	34	0/2	0/2	125
LP_GRB	✓	5	35	0/1	1/2	125
LP_CPX	✓	6	35	1/2	1/2	125
LP_GLP	✓	5	35	1/2	1/2	125
MZ	✓	0	16	12/12	9/9	125
TindellScaled_1, Set 2, 43 Tasks, 6 Unallocated Tasks (14%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	8	35	0/2	0/2	125
LP_GRB	✓	7	35	1/2	1/2	125
LP_CPX	✓	7	35	1/2	0/2	125
LP_GLP	✓	5	35	1/2	1/2	125
MZ	✓	0	15	10/10	7/7	125
TindellScaled_1, Set 2, 43 Tasks, 6 Unallocated Tasks (14%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	8	35	0/2	0/2	125
LP_GRB	✓	7	35	0/2	0/2	125
LP_CPX	✓	7	35	0/2	0/2	125
LP_GLP	✓	5	35	1/2	1/2	125
MZ	✓	0	15	9/9	7/7	125
TindellScaled_1, Set 1, 43 Tasks, 9 Unallocated Tasks (21%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	10	34	0/3	0/3	125
LP_GRB	✓	7	36	0/2	0/2	125
LP_CPX	✓	7	36	0/2	0/2	125
LP_GLP	✓	7	34	0/2	0/2	125

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
MZ	✓	0	19	291/291	117/117	125
TindellScaled_1, Set 1, 43 Tasks, 9 Unallocated Tasks (21%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	10	34	0/3	0/3	125
LP_GRB	✓	7	36	0/2	0/2	125
LP_CPX	✓	7	36	0/2	0/2	125
LP_GLP	✓	7	34	0/2	0/2	125
MZ	✓	0	19	314/314	102/102	125
TindellScaled_1, Set 2, 43 Tasks, 9 Unallocated Tasks (21%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	8	36	0/2	0/2	125
LP_GRB	✓	7	36	0/2	0/2	125
LP_CPX	✓	6	35	0/2	0/2	125
LP_GLP	✓	7	36	0/2	0/2	125
MZ	Inc	0	0	–/–	–/–	–
TindellScaled_1, Set 2, 43 Tasks, 9 Unallocated Tasks (21%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	8	36	0/2	0/2	125
LP_GRB	✓	7	36	0/2	0/2	125
LP_CPX	✓	6	35	0/2	0/2	125
LP_GLP	✓	7	36	0/2	0/2	125
MZ	Inc	0	0	–/–	–/–	–
TindellOriginal_1, Set 1, 43 Tasks, 16 Unallocated Tasks (38%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	21	37	0/4	0/4	125
LP_GRB	✓	8	42	0/3	0/3	143
LP_CPX	✓	9	40	1/4	1/4	133
LP_GLP	✓	8	43	6/9	6/9	125
MZ	Inc	0	0	–/–	–/–	–
TindellOriginal_1, Set 1, 43 Tasks, 16 Unallocated Tasks (38%), 3 Subsystems, 8 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	21	37	0/4	1/5	125
LP_GRB	✓	8	41	2/5	0/3	125
LP_CPX	✓	9	43	1/4	0/3	133
LP_GLP	✓	8	43	6/9	6/9	125
MZ	Inc	0	0	–/–	–/–	–
TindellScaled_1, Set 2, 43 Tasks, 16 Unallocated Tasks (38%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 1 Core(s)						

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
KL	✓	23	39	0/5	0/5	143
LP_GRB	✓	11	39	0/3	0/3	133
LP_CPX	✓	7	40	0/3	0/3	125
LP_GLP	✓	11	39	4/7	4/7	133
MZ	Inc	0	0	–/–	–/–	–
TindellScaled_1, Set 2, 43 Tasks, 16 Unallocated Tasks (38%), 3 Subsystems, 8 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	23	39	0/5	0/5	143
LP_GRB	✓	11	39	2/5	0/3	133
LP_CPX	✓	7	39	0/3	0/3	125
LP_GLP	✓	11	39	5/8	4/7	133
MZ	Inc	0	0	–/–	–/–	–
TindellScaled_2, Set 1, 86 Tasks, 12 Unallocated Tasks (14%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	21	41	0/5	0/5	250
LP_GRB	✓	17	44	1/5	1/5	250
LP_CPX	✓	17	44	0/4	0/4	250
LP_GLP	✓	14	45	60/64	60/64	250
MZ	✓	0	6	358/358	45/45	239
TindellScaled_2, Set 1, 86 Tasks, 12 Unallocated Tasks (14%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	21	41	0/5	0/5	250
LP_GRB	✓	14	45	5/9	2/7	250
LP_CPX	✓	17	44	3/7	2/9	250
LP_GLP	✓	14	45	67/71	71/75	250
MZ	✓	0	6	372/372	47/47	239
TindellScaled_2, Set 2, 86 Tasks, 12 Unallocated Tasks (14%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	26	42	1/6	1/6	250
LP_GRB	✓	16	45	1/4	1/4	250
LP_CPX	✓	16	46	1/4	1/4	250
LP_GLP	✓	18	45	70/74	72/77	250
MZ	✓	0	3	529/529	111/111	239
TindellScaled_2, Set 2, 86 Tasks, 12 Unallocated Tasks (14%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	26	42	0/6	0/6	250

Analysis Module	**Result**	**Local Runs**	**Free Global Bus Slots**	**Global / Total CPU Runtime [s] ◀ Better**	**Global / Total Wall-Clock Runtime [s] ◀ Better**	**Total Cost ◀ Better**
LP_GRB	✓	16	45	8/13	3/9	250
LP_CPX	✓	16	46	4/8	2/7	250
LP_GLP	✓	18	45	74/78	78/82	250
MZ	✓	0	3	530/530	108/108	239
TindellScaled_2, Set 1, 86 Tasks, 18 Unallocated Tasks (21%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	31	40	0/7	0/7	250
LP_GRB	✓	18	46	2/6	2/6	250
LP_CPX	✓	18	46	0/4	0/4	250
LP_GLP	✓	19	43	37/41	37/41	258
MZ	Time	0	0	–/–	–/–	–
TindellScaled_2, Set 1, 86 Tasks, 18 Unallocated Tasks (21%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	31	40	0/7	0/7	250
LP_GRB	✓	19	43	5/9	1/6	258
LP_CPX	✓	18	46	2/7	1/7	250
LP_GLP	✓	19	43	43/48	53/58	258
MZ	Time	0	0	–/–	–/–	–
TindellScaled_2, Set 2, 86 Tasks, 18 Unallocated Tasks (21%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	34	44	0/7	0/7	250
LP_GRB	✓	19	47	1/5	1/5	250
LP_CPX	✓	20	47	2/6	2/6	258
LP_GLP	✓	17	46	211/215	229/233	250
MZ	Time	0	0	–/–	–/–	–
TindellScaled_2, Set 2, 86 Tasks, 18 Unallocated Tasks (21%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	34	44	0/7	0/7	250
LP_GRB	✓	19	47	5/9	2/6	250
LP_CPX	✓	20	47	2/7	1/6	258
LP_GLP	✓	17	46	200/204	201/205	250
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_2, Set 1, 86 Tasks, 32 Unallocated Tasks (38%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	50	44	1/14	1/14	261
LP_GRB	✓	28	57	33/42	33/42	296
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
TindellOriginal_2, Set 1, 86 Tasks, 32 Unallocated Tasks (38%), 6 Subsystems, 16 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	50	44	1/14	1/14	261
LP_GRB	✓	34	56	258/267	23/32	268
LP_CPX	✓	22	60	17/25	5/13	268
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_2, Set 2, 86 Tasks, 32 Unallocated Tasks (38%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	75	49	0/18	0/18	275
LP_GRB	✓	25	54	26/33	26/33	258
LP_CPX	✓	27	54	8/15	8/15	268
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_2, Set 2, 86 Tasks, 32 Unallocated Tasks (38%), 6 Subsystems, 16 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	75	49	0/18	0/18	275
LP_GRB	✓	25	53	84/92	9/17	258
LP_CPX	✓	26	54	22/32	7/15	268
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 1, 129 Tasks, 18 Unallocated Tasks (14%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	39	31	1/12	0/12	375
LP_GRB	✓	32	36	33/40	33/40	383
LP_CPX	✓	30	32	10/18	11/19	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 1, 129 Tasks, 18 Unallocated Tasks (14%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	39	31	0/12	0/12	375
LP_GRB	✓	31	36	116/124	12/20	383
LP_CPX	✓	33	32	16/24	4/13	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 2, 129 Tasks, 18 Unallocated Tasks (14%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	60	29	0/13	1/14	375
LP_GRB	✓	38	34	5/12	5/12	375
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	✓	32	33	1402/1409	1420/1428	383
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
TindellScaled_3, Set 2, 129 Tasks, 18 Unallocated Tasks (14%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	60	29	1/14	1/14	375
LP_GRB	✓	38	34	45/52	6/13	375
LP_CPX	✓	39	34	23/30	4/12	375
LP_GLP	✓	32	33	1389/1397	1389/1397	383
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 1, 129 Tasks, 27 Unallocated Tasks (21%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	60	22	1/16	1/16	375
LP_GRB	✓	35	31	17/25	17/25	401
LP_CPX	✓	34	37	4/12	4/12	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 1, 129 Tasks, 27 Unallocated Tasks (21%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	60	22	1/16	1/16	375
LP_GRB	✓	35	35	111/120	11/20	375
LP_CPX	✓	33	37	19/27	5/13	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 2, 129 Tasks, 27 Unallocated Tasks (21%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	64	29	1/16	1/16	375
LP_GRB	✓	33	35	4/11	4/11	375
LP_CPX	✓	36	34	4/12	4/12	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 2, 129 Tasks, 27 Unallocated Tasks (21%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	64	29	1/16	1/16	375
LP_GRB	✓	33	35	31/39	5/13	375
LP_CPX	✓	37	34	13/22	4/13	375
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_3, Set 1, 129 Tasks, 48 Unallocated Tasks (38%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	103	32	2/54	6/143	404
LP_GRB	✓	70	63	1363/1388	1382/1408	411
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◄ Better	Global / Total Wall-Clock Runtime [s] ◄ Better	Total Cost ◄ Better
TindellOriginal_3, Set 1, 129 Tasks, 48 Unallocated Tasks (38%), 9 Subsystems, 24 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	103	32	4/72	8/161	404
LP_GRB	✓	58	65	14832/14894	2032/2119	429
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 2, 129 Tasks, 48 Unallocated Tasks (38%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	49	46	100/115	108/125	383
LP_CPX	✓	47	52	1836/1851	1839/1855	386
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_3, Set 2, 129 Tasks, 48 Unallocated Tasks (38%), 9 Subsystems, 24 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	51	49	1272/1289	110/126	386
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 1, 172 Tasks, 24 Unallocated Tasks (14%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	57	29	1/19	0/19	507
LP_GRB	✓	55	38	53/65	53/65	500
LP_CPX	✓	53	36	21/35	28/45	508
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 1, 172 Tasks, 24 Unallocated Tasks (14%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	57	29	1/19	0/19	507
LP_GRB	✓	54	36	191/203	21/33	500
LP_CPX	✓	55	36	41/56	9/24	508
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 2, 172 Tasks, 24 Unallocated Tasks (14%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	89	32	1/24	1/24	489
LP_GRB	✓	61	37	141/154	153/169	500
LP_CPX	✓	63	36	92/105	93/106	500
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 2, 172 Tasks, 24 Unallocated Tasks (14%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 12 Core(s)						

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
KL	✓	89	32	1/25	1/26	489
LP_GRB	✓	60	37	934/947	82/95	500
LP_CPX	✓	61	36	278/292	37/51	500
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 1, 172 Tasks, 36 Unallocated Tasks (21%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	86	21	2/35	1/35	489
LP_GRB	✓	58	37	623/636	643/656	508
LP_CPX	✓	60	35	98/111	98/111	500
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 1, 172 Tasks, 36 Unallocated Tasks (21%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	86	21	2/37	2/38	489
LP_GRB	✓	58	33	1321/1334	114/127	516
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 2, 172 Tasks, 36 Unallocated Tasks (21%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	92	31	1/25	1/25	489
LP_GRB	✓	57	34	36/51	40/58	500
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 2, 172 Tasks, 36 Unallocated Tasks (21%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	92	31	1/25	1/25	489
LP_GRB	✓	56	35	151/165	17/31	500
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_4, Set 1, 172 Tasks, 64 Unallocated Tasks (38%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_4, Set 1, 172 Tasks, 64 Unallocated Tasks (38%), 12 Subsystems, 32 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	**Result**	**Local Runs**	**Free Global Bus Slots**	**Global / Total CPU Runtime [s] ◀ Better**	**Global / Total Wall-Clock Runtime [s] ◀ Better**	**Total Cost ◀ Better**
TindellScaled_4, Set 2, 172 Tasks, 64 Unallocated Tasks (38%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	79	55	950/974 ▮	955/979 ▮	508 ▮
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_4, Set 2, 172 Tasks, 64 Unallocated Tasks (38%), 12 Subsystems, 32 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 1, 215 Tasks, 30 Unallocated Tasks (14%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	76	24	1693/1711 ▮	1694/1712 ▮	633 ▮
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 1, 215 Tasks, 30 Unallocated Tasks (14%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	75	23	8054/8171 ▮	700/822 ▮	625 ▮
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 30 Unallocated Tasks (14%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	147	12	3/88 ▮	3/89 ▮	625 ▮
LP_GRB	✓	80	18	333/351 ▮	333/351 ▮	641 ▮
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 30 Unallocated Tasks (14%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	147	12	4/94 ▮	3/95 ▮	625 ▮
LP_GRB	✓	81	23	1551/1582 ▮	140/172 ▮	625 ▮
LP_CPX	✓	112	25	4352/4403 ▮	629/681 ▮	625 ▮
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 1, 215 Tasks, 45 Unallocated Tasks (21%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	✓	83	17	677/718 ▮	677/718 ▮	641 ▮
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
TindellScaled_5, Set 1, 215 Tasks, 45 Unallocated Tasks (21%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	✓	82	25	1546/1595	140/190	625
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 45 Unallocated Tasks (21%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	143	17	5/253	7/356	614
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 45 Unallocated Tasks (21%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	143	17	8/310	8/355	614
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_5, Set 1, 215 Tasks, 80 Unallocated Tasks (38%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	✓	224	10	11/501	12/560	622
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_5, Set 1, 215 Tasks, 80 Unallocated Tasks (38%), 15 Subsystems, 40 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	224	10	12/549	12/608	622
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 80 Unallocated Tasks (38%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_5, Set 2, 215 Tasks, 80 Unallocated Tasks (38%), 15 Subsystems, 40 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 1, 258 Tasks, 36 Unallocated Tasks (14%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–

Analysis Module	**Result**	**Local Runs**	**Free Global Bus Slots**	**Global / Total CPU Runtime [s]** ◀ **Better**	**Global / Total Wall-Clock Runtime [s]** ◀ **Better**	**Total Cost** ◀ **Better**
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 1, 258 Tasks, 36 Unallocated Tasks (14%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 36 Unallocated Tasks (14%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	213	41	6/281 ▎	7/290 ▎	739 █
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 36 Unallocated Tasks (14%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	213	41	4/227 ▎	4/232 ▎	739 █
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 1, 258 Tasks, 54 Unallocated Tasks (21%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 1, 258 Tasks, 54 Unallocated Tasks (21%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 54 Unallocated Tasks (21%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	240	22	7/325 ▎	7/342 ▎	739 █
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 54 Unallocated Tasks (21%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	240	22	9/370 ▎	9/382 ▎	739 █
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_6, Set 1, 258 Tasks, 96 Unallocated Tasks (38%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
KL	✓	316	4	14/212	28/528	768
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_6, Set 1, 258 Tasks, 96 Unallocated Tasks (38%), 18 Subsystems, 48 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	✓	316	4	13/254	29/534	768
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 96 Unallocated Tasks (38%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_6, Set 2, 258 Tasks, 96 Unallocated Tasks (38%), 18 Subsystems, 48 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 1, 301 Tasks, 42 Unallocated Tasks (14%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 1, 301 Tasks, 42 Unallocated Tasks (14%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 42 Unallocated Tasks (14%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	271	109	15/331	15/363	864
LP_GRB	✓	143	153	2521/2575	2522/2576	883
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 42 Unallocated Tasks (14%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	271	109	17/313	18/320	864
LP_GRB	✓	151	167	9262/9390	941/1090	875
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 1, 301 Tasks, 63 Unallocated Tasks (21%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 1, 301 Tasks, 63 Unallocated Tasks (21%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Inc	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 63 Unallocated Tasks (21%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	✓	305	25	10/359	10/369	864
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 63 Unallocated Tasks (21%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	✓	305	25	7/294	8/298	864
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_7, Set 1, 301 Tasks, 112 Unallocated Tasks (38%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_7, Set 1, 301 Tasks, 112 Unallocated Tasks (38%), 21 Subsystems, 56 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 112 Unallocated Tasks (38%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_7, Set 2, 301 Tasks, 112 Unallocated Tasks (38%), 21 Subsystems, 56 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_8, Set 1, 344 Tasks, 128 Unallocated Tasks (38%), 24 Subsystems, 64 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_8, Set 1, 344 Tasks, 128 Unallocated Tasks (38%), 24 Subsystems, 64 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_8, Set 2, 344 Tasks, 128 Unallocated Tasks (38%), 24 Subsystems, 64 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_8, Set 2, 344 Tasks, 128 Unallocated Tasks (38%), 24 Subsystems, 64 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_9, Set 1, 387 Tasks, 144 Unallocated Tasks (38%), 27 Subsystems, 72 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_9, Set 1, 387 Tasks, 144 Unallocated Tasks (38%), 27 Subsystems, 72 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_9, Set 2, 387 Tasks, 144 Unallocated Tasks (38%), 27 Subsystems, 72 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_9, Set 2, 387 Tasks, 144 Unallocated Tasks (38%), 27 Subsystems, 72 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
TindellOriginal__10, Set 1, 430 Tasks, 160 Unallocated Tasks (38%), 30 Subsystems, 80 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__10, Set 1, 430 Tasks, 160 Unallocated Tasks (38%), 30 Subsystems, 80 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__10, Set 2, 430 Tasks, 160 Unallocated Tasks (38%), 30 Subsystems, 80 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__10, Set 2, 430 Tasks, 160 Unallocated Tasks (38%), 30 Subsystems, 80 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__11, Set 1, 473 Tasks, 176 Unallocated Tasks (38%), 33 Subsystems, 88 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__11, Set 1, 473 Tasks, 176 Unallocated Tasks (38%), 33 Subsystems, 88 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__11, Set 2, 473 Tasks, 176 Unallocated Tasks (38%), 33 Subsystems, 88 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__11, Set 2, 473 Tasks, 176 Unallocated Tasks (38%), 33 Subsystems, 88 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__12, Set 1, 516 Tasks, 192 Unallocated Tasks (38%), 36 Subsystems, 96 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__12, Set 1, 516 Tasks, 192 Unallocated Tasks (38%), 36 Subsystems, 96 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__12, Set 2, 516 Tasks, 192 Unallocated Tasks (38%), 36 Subsystems, 96 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__12, Set 2, 516 Tasks, 192 Unallocated Tasks (38%), 36 Subsystems, 96 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__13, Set 1, 559 Tasks, 208 Unallocated Tasks (38%), 39 Subsystems, 104 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__13, Set 1, 559 Tasks, 208 Unallocated Tasks (38%), 39 Subsystems, 104 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__13, Set 2, 559 Tasks, 208 Unallocated Tasks (38%), 39 Subsystems, 104 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__13, Set 2, 559 Tasks, 208 Unallocated Tasks (38%), 39 Subsystems, 104 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__14, Set 1, 602 Tasks, 224 Unallocated Tasks (38%), 42 Subsystems, 112 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◀ Better	Global / Total Wall-Clock Runtime [s] ◀ Better	Total Cost ◀ Better
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_14, Set 1, 602 Tasks, 224 Unallocated Tasks (38%), 42 Subsystems, 112 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_14, Set 2, 602 Tasks, 224 Unallocated Tasks (38%), 42 Subsystems, 112 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_14, Set 2, 602 Tasks, 224 Unallocated Tasks (38%), 42 Subsystems, 112 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_15, Set 1, 645 Tasks, 240 Unallocated Tasks (38%), 45 Subsystems, 120 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_15, Set 1, 645 Tasks, 240 Unallocated Tasks (38%), 45 Subsystems, 120 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_15, Set 2, 645 Tasks, 240 Unallocated Tasks (38%), 45 Subsystems, 120 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled_15, Set 2, 645 Tasks, 240 Unallocated Tasks (38%), 45 Subsystems, 120 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal_16, Set 1, 688 Tasks, 256 Unallocated Tasks (38%), 48 Subsystems, 128 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP_GRB	Time	0	0	–/–	–/–	–
LP_CPX	Time	0	0	–/–	–/–	–
LP_GLP	Time	0	0	–/–	–/–	–

Analysis Module	**Result**	**Local Runs**	**Free Global Bus Slots**	**Global / Total CPU Runtime [s] ◀ Better**	**Global / Total Wall-Clock Runtime [s] ◀ Better**	**Total Cost ◀ Better**
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__16, Set 1, 688 Tasks, 256 Unallocated Tasks (38%), 48 Subsystems, 128 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__16, Set 2, 688 Tasks, 256 Unallocated Tasks (38%), 48 Subsystems, 128 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Inc	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__16, Set 2, 688 Tasks, 256 Unallocated Tasks (38%), 48 Subsystems, 128 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__17, Set 1, 731 Tasks, 272 Unallocated Tasks (38%), 51 Subsystems, 136 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__17, Set 1, 731 Tasks, 272 Unallocated Tasks (38%), 51 Subsystems, 136 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__17, Set 2, 731 Tasks, 272 Unallocated Tasks (38%), 51 Subsystems, 136 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__17, Set 2, 731 Tasks, 272 Unallocated Tasks (38%), 51 Subsystems, 136 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__18, Set 1, 774 Tasks, 288 Unallocated Tasks (38%), 54 Subsystems, 144 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Analysis Module	Result	Local Runs	Free Global Bus Slots	Global / Total CPU Runtime [s] ◂ Better	Global / Total Wall-Clock Runtime [s] ◂ Better	Total Cost ◂ Better
TindellOriginal__18, Set 1, 774 Tasks, 288 Unallocated Tasks (38%), 54 Subsystems, 144 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__18, Set 2, 774 Tasks, 288 Unallocated Tasks (38%), 54 Subsystems, 144 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__18, Set 2, 774 Tasks, 288 Unallocated Tasks (38%), 54 Subsystems, 144 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__19, Set 1, 817 Tasks, 304 Unallocated Tasks (38%), 57 Subsystems, 152 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__19, Set 1, 817 Tasks, 304 Unallocated Tasks (38%), 57 Subsystems, 152 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__19, Set 2, 817 Tasks, 304 Unallocated Tasks (38%), 57 Subsystems, 152 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__19, Set 2, 817 Tasks, 304 Unallocated Tasks (38%), 57 Subsystems, 152 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__20, Set 1, 860 Tasks, 320 Unallocated Tasks (38%), 60 Subsystems, 160 ECUs, WCET Regularity 0.807979%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellOriginal__20, Set 1, 860 Tasks, 320 Unallocated Tasks (38%), 60 Subsystems, 160 ECUs, WCET Regularity 0.807979%, max. 12 Core(s)						

Analysis Module	**Result**	**Local Runs**	**Free Global Bus Slots**	**Global / Total CPU Runtime [s] ◀ Better**	**Global / Total Wall-Clock Runtime [s] ◀ Better**	**Total Cost ◀ Better**
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__20, Set 2, 860 Tasks, 320 Unallocated Tasks (38%), 60 Subsystems, 160 ECUs, WCET Regularity 1%, max. 1 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–
TindellScaled__20, Set 2, 860 Tasks, 320 Unallocated Tasks (38%), 60 Subsystems, 160 ECUs, WCET Regularity 1%, max. 12 Core(s)						
KL	Time	0	0	–/–	–/–	–
LP__GRB	Time	0	0	–/–	–/–	–
LP__CPX	Time	0	0	–/–	–/–	–
LP__GLP	Time	0	0	–/–	–/–	–
MZ	Time	0	0	–/–	–/–	–

Glossary

Allocation
Defines how a Task Network is allocated onto a Hardware Architecture. *38*

Bus Slot
A bus slot represents one time slice of a TDMA-bus where messages can be allocated. **8**

Bus-Type
A bus type describes all properties of a hardware communication bus. It is used to type a bus. **7**, *28*

Controller Area Network (CAN)
Controller Area Network. 7, 33

Design Space Exploration (DSE)
Design Space Exploration refers to the process of exploring the solution space of possible hardware/software architectures subject to problem specific constraints usually with an optimization objective (e.g. minimize cost). 1, 2, 43

Digital Signal Processor (DSP)
Digital Signal Processor. 8, 9

ECU-Type
Specification of the type of an ECU containing all relevant information like processor type, size and type of memory including caches, I/O ports, etc. **7**, 27

Fixed-Priority Preemptive Scheduling (FPS)
Fixed-Priority Preemptive Scheduling is an approach for scheduling tasks on (single-core) processors where a fixed priority is assigned to each task a-priori with the consequence that during execution of a task with higher priority all tasks with lower priority which are ready for execution or already executed are preempted. **16**

FlexRay
A field bus specifically developed for automotive applications based on the TDMA bus access protocol. 7, 30

GNU Compiler Collection (GCC)
GNU Compiler Collection: An open source collection of compilers for various programming languages and target platforms. 8

GNU Linear Programming Kit (GLPK)
The GNU Linear Programming Kit is an open source solver for mixed integer linear programming (MILP) problems. 184

Hardware Architectural Pattern
A hardware architectural pattern determines a set of possible hardware architectures (the hardware architecture design space) by specifying mandatory and optional hardware components. **6**

Hardware Architecture
All hardware components of an embedded systems. **5**

Hardware Design Space
The hardware design space of a given optimization problem is defined by specifying an hardware architectural pattern. **6**, *27*

Linear Programming
Linear Programming is an optimization technique for the efficient minimization/-maximization of a given objective function whose free variables (only reals) are subject to constraints specified as linear inequations. **24**

Logical Bus
A logical bus represents a communication bus in a hardware architectural pattern. A logical bus is typed by a bus-type. **7**, *28*

Logical Electronic Control Unit (ECU)
Electronic Control Units (ECU) are used in embedded systems and consist of a microprocessor together with memory, I/O ports, etc. In this thesis the term ECU is used to name an element of a hardware architectural pattern where an electronic control unit of a specific ECU-type could be inserted. **7**, 13–16, 18, 20, 27

MaxWCET
Characterization of available capacity for additional tasks/signals on priority-scheduled ECUs/buses. Ensures that as long as MaxWCET capacity is not exceeded none of the newly allocated tasks/signals on the resource will miss its deadline. **90**

Message
In this work, messages are used to transmit signals over communication buses. **11**, *33*

Mixed Integer (Linear) Programming (MILP)
Mixed Integer (Linear) Programming is based on linear programming but additional allows some or all free variables to be integers instead of reals. **24**

Satisfiabiliy Modulo Theories (SMT)
Satisfiabiliy Modulo Theories is a research area where classical satisfiability solving methods are extended using a theory module, e.g. for handling linear constraints. See e.g. [Mis] for current activities on that subject. **23**

Signal

In this work, signals abstractly represent data flow between tasks. **11**, *32*

Software Task

Abstract representation of a specified functionality implemented in a programming language and the binary code for one or more execution platform derived by compiling the source code. May be the receiver of one signal and may be the sender of one or more signals. Is characterized by properties like activation period, local hard deadline and worst case execution times for each of the execution platforms. **8**

Spare-Time

Characterization of available capacity for additional tasks/signals on priority-scheduled ECUs/buses. Ensures that as long as Spare-Time capacity is not exceeded no existing tasks/signals on the resource will miss its deadline. **90**

Task Network

A task network is a directed graph consisting of vertices which are either software tasks or signals. **11**, *36*

Time Division Multiple Access (TDMA)

Time Division Multiple Access is a method for organizing the communication of multiple devices over a shared physical communication channel. 7, 28, 30, 33

Worst Case Execution Time (WCET)

The Worst Case Execution Time (WCET) of a software task on a specified execution platform is a safe upper bound for the maximal time required for the isolated execution of the platform-specific binary code derived by compilation of the source code of the software task on that platform. 10, 11, 32, 40

Worst Case Response Time (WCRT)

The Worst Case Response Time (WCRT) of a software task is a safe upper bound on the time after the arrival (activation) of any instance task until its completion considering all possible delays which might be e.g. caused by the presence of other tasks on the same processor scheduled by a specified scheduling policy. **15**

Bibliography

[Abs] AbsInt Angewandte Informatik GmbH. *aiT Worst-Case Execution Time Analyzers*. URL: `http://www.absint.com` (visited on 06/14/2013).

[Alt+12] Ernst Althaus, Sebastian Hoffmann, Joschka Kupilas, and Eike Thaden. "A Column Generation Approach to Scheduling of Real-Time Networks". In: *Proceedings of the World Congress on Engineering and Computer Science (WCECS)*. Vol. 1. Best Paper Award of International Conference on Computer Science and Applications 2012. IAENG. San Francisco, USA: IAENG, Oct. 2012, pp. 224–229. ISBN: 978-988-19251-6-9. DOI: `10.1007/978-3-642-20662-7_29`. URL: `http://www.iaeng.org/publication/WCECS2012/WCECS2012\_pp224-229.pdf`.

[Anoa] Anonymous. *Embedded Systems Guide. Common Technical Baseline*. URL: `http://www.embedded-systems-portal.com` (visited on 01/11/2013).

[Anob] Anonymous. *OFFIS – Institute for Information Technology*. URL: `http://www.offis.de` (visited on 01/27/2013).

[Ano12] Anonymous. *The R Project for Statistical Computing*. 2012. URL: `http://www.r-project.org/` (visited on 11/25/2012).

[Ano13] Anonymous. *Wikibook Embedded Systems*. Jan. 11, 2013. URL: `http://en.wikibooks.org/wiki/Embedded_Systems` (visited on 01/11/2013).

[ANT11] Ernst Althaus, Rouven Naujoks, and Eike Thaden. "A Column Generation Approach to Scheduling of Periodic Tasks". In: *Experimental Algorithms – 10th International Symposium, SEA 2011, Proceedings*. Ed. by Panos M. Pardalos and Steffen Rebennack. Vol. 1. Lecture Notes in Computer Science / Programming and Software Engineering 6630. Kolimpari, Crete: Springer Berlin, May 2011, pp. 340–351. ISBN: 978-3-642-20661-0. DOI: `10.1007/978-3-642-20662-7_29`.

[ARS00] Santosh G. Abraham, B. Ramakrishna Rau, and Robert Schreiber. *Fast Design Space Exploration through Validity and Quality Filtering of Subsystem Designs*. Tech. rep. Packard, Compiler and Architecture Research, HP Laboratories Palo Alto, 2000.

[AS00] Tarek F. Abdelzaher and Kang G. Shin. "Period-Based Load Partitioning and Assignment for Large Real-Time Applications". In: *IEEE Trans. Comput.* 49 (1 Jan. 2000), pp. 81–87. ISSN: 0018-9340. DOI: `10.1109/12.822566`. URL: `http://dl.acm.org/citation.cfm?id=330322.330337`.

[Aud90] Neil C. Audsley. *Deadline Monotonic Scheduling.* Tech. rep. University of York, Department of Computer Science, 1990.

[Bao+10a] J. Bao et al. *Projektgruppe ViDAs.* 2010. URL: `http://vidas.informatik.uni-oldenburg.de/` (visited on 10/28/2012).

[Bao+10b] J. Bao et al. *Projektgruppe ViDAs - Endbericht.* Tech. rep. Carl von Ossietzky Universität Oldenburg, 2010. URL: `http://vidas.informatik.uni-oldenburg.de/autobuild/doc/Endbericht.pdf` (visited on 06/14/2013).

[Bec+12] B. Becker, W. Damm, B. Finkbeiner, M. Fränzle, E. Olderog, and A. Podelski (Members of the Board). *Automatic Verification And Analysis of Complex System (AVACS).* 2012. URL: `http://www.avacs.org/` (visited on 10/29/2012).

[Ble+03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. "PISA — A Platform and Programming Language Independent Interface for Search Algorithms". English. In: *Evolutionary Multi-Criterion Optimization.* Ed. by CarlosM. Fonseca, PeterJ. Fleming, Eckart Zitzler, Lothar Thiele, and Kalyanmoy Deb. Vol. 2632. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 494–508. ISBN: 978-3-540-01869-8. DOI: `10.1007/3-540-36970-8_35`. URL: `http://dx.doi.org/10.1007/3-540-36970-8_35`.

[BMS09] Matthias Büker, Alexander Metzner, and Ingo Stierand. "Testing Real-Time Task Networks with Functional Extensions Using Model-Checking". In: *14th International Conference on Emerging Technologies and Factory Automation.* 2009.

[Böh] Wolfgang Böhm. *SPES_XT.* URL: `http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html` (visited on 01/27/2013).

[Bro06] Manfred Broy. "Challenges in automotive software engineering". In: *Proceedings of the 28th international conference on Software engineering.* ICSE '06. Shanghai, China: ACM, 2006, pp. 33–42. ISBN: 1-59593-375-1. DOI: `http://doi.acm.org/10.1145/1134285.1134292`. URL: `http://doi.acm.org/10.1145/1134285.1134292`.

[Bük+11a] Matthias Büker, Werner Damm, Günter Ehmen, Alexander Metzner, Ingo Stierand, and Eike Thaden. "Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too". In: *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on.* June 2011, pp. 156–165. DOI: `10.1109/SIES.2011.5953658`.

[Bük+11b] Matthias Büker, Werner Damm, Günter Ehmen, Alexander Metzner, Ingo Stierand, and Eike Thaden. *Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too.* Tech. rep. 69. SFB/TR 14 AVACS, 2011. URL: `http://www.avacs.org/`.

[Bük12] Matthias Büker. “An Automated Semantic-based Approach for Creating Task Structures”. PhD thesis. Carl von Ossietzky Universität Oldenburg, 2012. Submitted.

[BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. 3rd ed. Addison Wesley, 2001. ISBN: 0-201-72988-1.

[Can] Canonical. *Ubuntu*. URL: `http://www.ubuntu.com` (visited on 09/06/2012).

[Cha+12] Sudipta Chattopadhyay, Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. *Chronos*. National University of Singapore, 2012. URL: `http://www.comp.nus.edu.sg/~rpembed/chronos/`.

[CST11] Brian Clark, Ingo Stierand, and Eike Thaden. “Cost-Minimal Pre-Allocation of Software Tasks Under Real-Time Constraints”. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS 2011)*. Ed. by Rex Earl Gantenbein and Tei Wei Wei Kuo. RACS ’11. Miami, Florida, Nov. 2011, pp. 77–83. ISBN: 978-1-4503-1087-1. DOI: `10.1145/2103380.2103395`.

[Dan63] George Dantzig. *Linear Programming and Extensions*. Princeton University Press, Aug. 3, 1963. ISBN: 0691059136. URL: `http://www.worldcat.org/isbn/0691059136`.

[Dav+07] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. “Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised”. In: *Real-Time Syst.* 35 (3 Apr. 2007), pp. 239–272. ISSN: 0922-6443. DOI: `10.1007/s11241-007-9012-7`. URL: `http://portal.acm.org/citation.cfm?id=1227689.1227696`.

[Deb] Debian Project. *http://www.debian.org/*. URL: `http://www.debian.org` (visited on 09/06/2012).

[DG00] Raymond Devillers and Joel Goossens. “Liu and Layland’s Schedulability Test Revisited”. In: *Information Processing Letters*. 2000, pp. 157–161.

[Dor+08] François Dorin, Michael Richard, Emmanuel Grolleau, and Pascal Richard. “Minimizing the number of processors for real-time distributed systems”. In: *16th International Conference on Real-Time and Network Systems*. Ed. by Giorgio Buttazzo and Pascale Minet. Isabelle Puaut. Rennes, France, 2008. URL: `http://hal.inria.fr/inria-00336511/en/`.

[Dut93] Shantanu Dutt. “New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms”. In: *In Proc. IEEE Intl. Conf. Computer-Aided Design*. 1993, pp. 370–377.

[EB10] Paul Emberson and Iain Bate. “Stressing Search with Scenarios for Flexible Solutions to Real-Time Task Allocation Problems”. In: *IEEE Trans. Softw. Eng.* 36 (5 Sept. 2010), pp. 704–718. ISSN: 0098-5589. DOI: `http://dx.doi.org/10.1109/TSE.2009.58`. URL: `http://dx.doi.org/10.1109/TSE.2009.58`.

[Eis+06] F. Eisenbrand, W. Damm, A. Metzner, G. Shmonin, R. Wilhelm, and S. Winkel. “Mapping Task-Graphs on Distributed ECU Networks: Efficient Algorithms for Feasibility and Optimality”. In: *Proceedings of the 12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, 2006.

[Erb06] Cagkan Erbas. “System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures”. PhD thesis. University of Amsterdam, 2006.

[FH07] Martin Fränzle and Christian Herde. “HySAT: An efficient proof engine for bounded model checking of hybrid systems”. English. In: *Formal Methods in System Design* 30 (3 2007), pp. 179–198. ISSN: 0925-9856. DOI: `10.1007/s10703-006-0031-0`. URL: `http://dx.doi.org/10.1007/s10703-006-0031-0`.

[GR00] J. Gerlach and W. Rosenstiel. “A methodology and tool for automated transformational high-level design space exploration”. In: *Computer Design, 2000. Proceedings. 2000 International Conference on.* 2000, pp. 545 –548. DOI: `10.1109/ICCD.2000.878337`.

[Hau+03] C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, and A. Tyagi. *Hierarchical Synthesis of Embedded Systems Using Evolutionary Algorithms – A Multi-Objective Approach.* 2003.

[Her09] Christian Herde. *HySAT Quick Start Guide*. Tech. rep. Carl von Ossietzky Universität Oldenburg, 2009. URL: `http://hysat.informatik.uni-oldenburg.de/user_guide/hysat-user-guide.pdf`.

[Her10] Christian Herde. “Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure: Proof Engines for the Analysis of Hybrid Discrete–Continuous Systems”. PhD thesis. Carl von Ossietzky Universiät Oldenburg, 2010.

[Her12] Christian Herde. *HySAT - A bounded model checker for hybrid systems.* Sept. 6, 2012. URL: `http://hysat.informatik.uni-oldenburg.de/`.

[HRE06] Arne Hamann, Razvan Racu, and Rolf Ernst. “Formal Methods for Automotive Platform Analysis and Optimization”. In: *Proc. Future Trends in Automotive Electronics and Tool Integration Workshop*. 2006.

[HT03] C. Haubelt and J. Teich. “Accelerating design space exploration using Pareto-front arithmetics”. In: *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific.* Jan. 2003, pp. 525 –531. DOI: `10.1109/ASPDAC.2003.1195073`.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)*. 1990. ISBN: 1-55937-067-X.

[INC12] INCHRON GmbH. *Real-Time Simulator ChronSIM*. 2012. URL: `http://www.inchron.de` (visited on 05/16/2012).

[JP86] M. Joseph and P. K. Pandya. “Finding response times in a real-time system.” In: *The Computer Journal* 29 (1986), pp. 390–395.

[Kar84] N. Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. STOC ’84. New York, NY, USA: ACM, 1984, pp. 302–311. ISBN: 0-89791-133-4. DOI: 10.1145/800057.808695. URL: http://dx.doi.org/10.1145/800057.808695.

[Kha79] L. G. Khachiyan. “A polynomial algorithm in linear programming”. In: *Doklady Akademii Nauk SSSR* 244 (1979), pp. 1093–1096.

[KL70] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *The Bell system technical journal* 49.1 (1970), pp. 291–307.

[Kop98] H. Kopetz. “A Comparison of CAN and TTP”. In: *Proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS 98)*. 1998.

[Kue06] Simon Kuenzli. “Efficient Design Space Exploration for Embedded Systems”. PhD thesis. Swiss Federal Institute of Technology, 2006.

[KV10] Bernhard Korte and Jens Vygen. *Combinatorial Optimization – Theory and Algorithms*. 4th ed. Vol. 1. Springer, 2010. ISBN: 978-3-642-09092-9. DOI: 10.1007/978-3-540-71844-4.

[LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *Journal of the ACM* 20 (1973), pp. 46–61. DOI: http://doi.acm.org/10.1145/321738.321743.

[LLC12] AMPL Optimization LLC. *AMPL – A Modeling Language for Mathematical Programming*. Sept. 4, 2012. URL: http://www.ampl.com/.

[Luk+09] M. Lukasiewycz, Michael Glaß, Jürgen Teich, and Paul Milbredt. “FlexRay schedule optimization of the static segment”. In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2009, pp. 363–372. ISBN: 978-1-60558-628-1. DOI: http://doi.acm.org/10.1145/1629435.1629485. URL: http://doi.acm.org/10.1145/1629435.1629485.

[LY03] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003. ISBN: 978-1-57820-124-2.

[Mad+07] Jan Madsen, Thomas K. Stidsen, Peter Kjærulf, and Shankar Mahadevan. “Multi-Objective Design Space Exploration of Embedded System Platforms.” In: *DIPES*. Ed. by Bernd Kleinjohann, Lisa Kleinjohann, Ricardo Jorge Machado, Carlos Eduardo Pereira, and P. S. Thiagarajan. Vol. 225. IFIP. Springer, July 4, 2007, pp. 185–194. ISBN: 978-0-387-39361-2. DOI: 10.1.1.83.6204.

[Mak10] Andrew Makhorin. *GNU Linear Programming Kit*. Sept. 2010. URL: http://www.gnu.org/software/glpk/ (visited on 06/14/2013).

[Mar10] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems.* 2nd Edition. Springer, Dec. 2010. ISBN: 978-94-007-0256-1. DOI: `10.1007/978-94-007-0257-8`.

[Mat12] Mathworks Inc. *Simulink. Simulation and Model-Based Design.* Tech. rep. 2012. URL: `http://www.mathworks.de/products/datasheets/pdf/simulink.pdf` (visited on 06/14/2013).

[Met+06] Alexander Metzner, Martin Fränzle, Christian Herde, and Ingo Stierand. "An optimal approach to the task allocation problem on hierarchical architectures". In: *Proceedings of the 20th international conference on Parallel and distributed processing.* IPDPS'06. Rhodes Island, Greece: IEEE Computer Society, 2006. ISBN: 1-4244-0054-6.

[Mis] Misc Authors. *Satisfiability Modulo Theories Competition.* URL: `http://smtcomp.sourceforge.net` (visited on 09/04/2012).

[PEP06] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels". In: *IEEE Transactions on Computers* 55 (2006), pp. 99–112. DOI: `10.1109/TC.2006.16`. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1566572&tag=1#`.

[Pop+04] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop. "Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems". In: *IEEE Trans. Very Large Scale Integr. Syst.* 12 (8 Aug. 2004), pp. 793–811. ISSN: 1063-8210. DOI: `http://dx.doi.org/10.1109/TVLSI.2004.831467`. URL: `http://dx.doi.org/10.1109/TVLSI.2004.831467`.

[Pop+06] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop. "Analysis and optimization of distributed real-time embedded systems". In: *ACM Trans. Des. Autom. Electron. Syst.* 11.3 (2006), pp. 593–625. ISSN: 1084-4309. DOI: `http://doi.acm.org/10.1145/1142980.1142984`.

[Pop+08] Traian Pop, Paul Pop, Petru Eles, and Zebo Peng. "Analysis and Optimisation of Hierarchically Scheduled Multiprocessor Embedded Systems". In: *International Journal of Parallel Programming* 36.1 (2008), pp. 37–67.

[Sil+11] Cristina Silvano et al. "MULTICUBE: Multi-Objective Design Space Exploration of Multi-Core Architectures". English. In: *VLSI 2010 Annual Symposium.* Ed. by Nikolaos Voros, Amar Mukherjee, Nicolas Sklavos, Konstantinos Masselos, and Michael Huebner. Vol. 105. Lecture Notes in Electrical Engineering. Springer Netherlands, 2011, pp. 47–63. ISBN: 978-94-007-1487-8. DOI: `10.1007/978-94-007-1488-5_4`. URL: `http://dx.doi.org/10.1007/978-94-007-1488-5_4`.

[Sta12] William Stallings. *Operating Systems: Internals and Design Principles.* International Edition of the 7th Revised Edition. Pearson Education, May 2012. ISBN: 978-0-273-75150-2.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. 2nd ed. Prentice Hall, 2001.

[Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications”. In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309. URL: `http://projecteuclid.org/euclid.pjm/1103044538`.

[TB94] Ken Tindell and Alan Burns. “Guaranteeing Message Latencies On Control Network (CAN)”. In: *In Proceedings of the 1st International CAN Conference*. CiA, 1994, pp. 1–2.

[TBW92] Ken Tindell, Alan Burns, and Andy Wellings. “Allocating Hard Real Time Tasks (An NP-Hard Problem Made Easy)”. In: *Journal of Real-Time Systems* 4 (1992), pp. 145–165.

[TBW95] K. Tindell, A. Burns, and A. Wellings. “Calculating Controller Area Network (CAN) Message Response Times”. In: *Control Engineering Practice* 3 (1995), pp. 1163–1169.

[Tea12] SRI Team. *Yices: An SMT Solver*. Sept. 6, 2012. URL: `http://yices.csl.sri.com/`.

[Tha+10] Eike Thaden, Henrik Lipskoch, Alexander Metzner, and Ingo Stierand. “Exploiting Gaps in Fixed-Priority Preemptive Schedules for Task Insertion”. In: *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE Computer Society, 2010, pp. 212–217. ISBN: 978-1-4244-8480-5. DOI: `10.1109/RTCSA.2010.27`.

[THW94] K.W. Tindell, H. Hansson, and A.J. Wellings. “Analysing real-time communications: controller area network (CAN)”. In: *Real-Time Systems Symposium, 1994., Proceedings*. Dec. 1994, pp. 259 –263. DOI: `10.1109/REAL.1994.342710`.

[Tin96] Kenneth William Tindell. “Fixed Priority Scheduling of Hard Real-Time Systems”. PhD thesis. Department of Computer Science, University of York, 1996.

[Wil+08] R. Wilhelm et al. “The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: `10.1145/1347375.1347389`. URL: `http://doi.acm.org/10.1145/1347375.1347389`.

[Zen+06] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. “Design Space Exploration of Automotive Platforms in Metropolis”. In: *Society of Automotive Engineers Congress*. Apr. 2006.

[Zen+11] Haibo Zeng, M. Di Natale, A. Ghosal, and A. Sangiovanni-Vincentelli. "Schedule Optimization of Time-Triggered Systems Communicating Over the FlexRay Static Segment". In: *Industrial Informatics, IEEE Transactions on* 7.1 (Feb. 2011), pp. 1 –17. ISSN: 1551-3203. DOI: `10.1109/TII.2010.2089465`.

[Zhu+10] Qi Zhu, Yang Yang, Marco Di Natale, Eelco Scholte, and Alberto L. Sangiovanni-Vincentelli. "Optimizing the Software Architecture for Extensibility in Hard Real-time Distributed Systems." In: *IEEE Trans. Industrial Informatics* 6.4 (2010), pp. 621–636. URL: `http://dblp.uni-trier.de/db/journals/tii/tii6.html#ZhuYNSS10`.